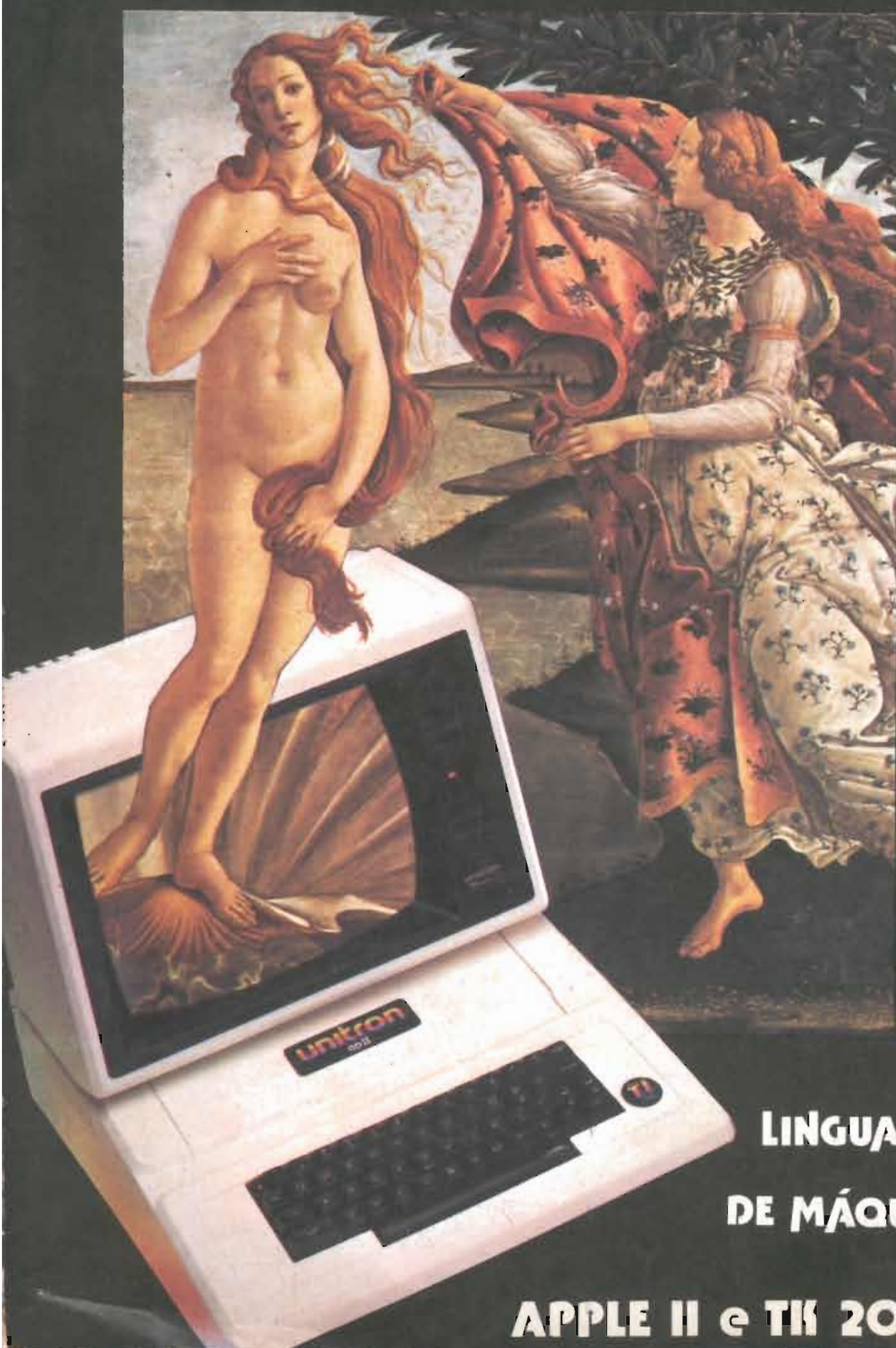
assembly 6502

BERNHARD
WOLFGANG
SCHÖN



LINGUAGEM DE MÁQUINA

APPLE II e TK 2000

05)

assembly 6502

ALEXANDRE FERNANDES
Departamento Técnico

bernhard wolfgang schön

assembly 6502

2ª EDIÇÃO

Coordenação Editorial:
Pierluigi Piazzi
Coordenação didática:
Betty Fromer Piazzi
Avaliação, editoração e revisão técnica:
Glauter Fabiano Mikahil
Organização editorial:
Rosana de Angelo
Revisão e copy-desk:
Lúcia Kairovsky
Arte:
Álvaro Correia
Capa:
Moretti, Piazzi e Botticelli
Produção:
Rosa Kogan Fromer

ALEPH
Publicações e Ass. Pedag. Ltda.
Av. Brig. Faria Lima 1451 c. 31
01451 S. Paulo SP
(011) 813-4555

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

S392a

Schön, Bernhard Wolfgang, 1954-
Assembly 6502 / Bernhard Wolfgang Schön.
São Paulo : Aleph, 1985.
(Segredos do Apple)

1. Apple (Computador) 2. Assembler (Linguagem de programação para computadores) 3. Linguagem de programação (Computadores) 4. Microprocessador 6502 5. Microprocessadores — Programação I. Título.

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

85-1468

Índices para catálogo sistemático:

1. Apple : Computadores eletrônicos : Processamento de dados 651.8 (17.) 001.642 (18.)
2. Linguagem de programação : Computadores : Processamento de dados 651.8 (17.) 001.6424 (18.)
3. Microprocessadores 6502 : Programação : Processamento de dados 651.8 (17.) 001.642 (18.)
4. Programação : Microprocessadores 6502 : Processamento de dados 651.8 (17.) 001.642 (18.)

sumário

Introdução 7
As vantagens da Linguagem de Máquina

Capítulo 1. 9
A CPU 6502: *Seu interior, seu status*

Capítulo 2 13
LDA, LDX, LDY: *Os modos imediato, absoluto, página zero e indireto, Como carregar um programa em Linguagem de Máquina*

Capítulo 3. 37
TAX, TAY, TXA, TYA: *As instruções de transferência de dados da memória para os registradores*

Capítulo 4. 41
INC, INX, INY: *As instruções de incremento*

Capítulo 5. 47
DEC, DEX, DEY: *As instruções de decremento*

Capítulo 6. 51
RTS, BRK, NOP: *A volta ao BASIC e as instruções de "reserva"*

Capítulo 7. 55
STA, STX, STY: *A transferência de dados de registradores para a memória*

Capítulo 8. 63
ADC, SBC: *Adição e subtração*

Capítulo 9. 73
CLC, CLD, CLV, SEC, SED: *Resetando os flags*

Capítulo 10. 75
BCC, BCS, BEQ, BMI, BNE, BPL, BVC, BVS: *Desvios condicionais comandados pelo estado dos flags*

Capítulo 11. 85
JMP, JSR: *Desvios incondicionais*

Capítulo 12. 89
AND, EOR, ORA: *Instruções lógicas*

Capítulo 13. 99
ASL, ROL, ROR, LSR: *Manipulação de bits: translação e rotação*

Capítulo 14. 105
CMP, CPX, CPY, BIT: *As instruções de comparação*

Capítulo 15. 111
PHA, PHP, PLA, PLP: *Transferência de registradores ou de status para o stack e vice-versa*

Capítulo 16. 115
TSX, TXS: *Transferência do conteúdo de um registrador para o stack-pointer e vice-versa*

Capítulo 17. 117
SEI, CLI, RTI: *Manipulando o flag de interrupção*

Capítulo 18. 119
Etiquetas

Capítulo 19. 127
Fluxogramas

Apêndice A. 133
Sistemas de Numeração

Apêndice B 145
Mensagem aos programadores de Z80

Apêndice C 147
Relação das instruções da CPU 6502

Respostas dos Exercícios. 163

introdução

Ao abrir este livro, o leitor está quebrando uma barreira criada pelos limites da linguagem de programação usada até agora — o BASIC. Provavelmente você já ouviu falar em linguagem de máquina, mas talvez até agora nunca tenha se preocupado em saber o que realmente a linguagem de máquina pode fazer por nós. Será que dominando o BASIC, torna-se desnecessário o emprego de programas escritos em linguagem de máquina?

A melhor maneira de responder a esta pergunta é questionarmo-nos sobre o que é o BASIC. Conhecemos esta linguagem através dos comandos PRINT, GOTO etc., bastante compreensíveis para nós. Mas será que o computador realmente sabe o que é um PRINT, um GOTO? Pesquisando na literatura técnica, percebemos logo que cada comando do BASIC nada mais é do que uma rotina complexa escrita em linguagem de máquina, escondida dos nossos olhos. Assim, ao digitar um PRINT, o micro localiza no seu arquivo fixo (ROM) a rotina em linguagem de máquina equivalente ao PRINT. Ora — se cada comando de BASIC no fundo é uma rotina escrita em linguagem de máquina — não seria lógico estudar-se esta linguagem para criar novos comandos jamais sonhados em BASIC?

Além da possibilidade de obter funções novas e rotinas diferentes, a linguagem de máquina proporciona ainda outras vantagens marcantes como:

— velocidade de execução muito superior à velocidade do BASIC;

– economia considerável da memória do usuário: uma determinada rotina escrita em BASIC ocupa muito mais memória do que a mesma rotina escrita em linguagem de máquina.

Neste livro trataremos da CPU 6502, das operações de cada instrução, bem como dos exemplos de aplicação de uma maneira simples e direta. Todos os exercícios requerem uma participação prática do leitor no próprio computador, o que além de possibilitar uma melhor visualização do exposto, ainda cria uma certa intimidade e familiaridade com o programa Assembler já existente no seu micro.

Aconselhamos ao leitor que porventura já conheça a CPU Z80, a dar uma olhada no apêndice B, que explica com poucas palavras as principais diferenças entre esta CPU e a 6502. No apêndice A são discutidos o sistema binário e o sistema hexadecimal, pois seu conhecimento é essencial para um programador em linguagem de máquina.

No apêndice C temos um resumo de todas as instruções da CPU 6502 na forma de várias tabelas, incluindo uma que serve para calcular o tempo de processamento de uma rotina escrita em linguagem de máquina.

No apêndice D temos as respostas aos exercícios propostos.

O leitor perceberá, trabalhando ordenadamente, que linguagem de máquina não é um bicho de sete cabeças, mas sim uma nova opção para ampliar a capacidade do seu microcomputador até ao limite de sua criatividade e imaginação.

capítulo 1

a cpu 6502

O que significa afinal "CPU 6502"?

As letras CPU são a abreviação para "Central Processing Unit", ou em português, "Unidade Central de Processamento". O número a seguir representa o tipo da CPU em questão (ex.: CPU 8080, CPU 6800 etc.). Mas, o que faz esta unidade central no nosso microcomputador?

Bem, comparando microcomputadores de diversas marcas, percebemos que sempre existem algumas diferenças entre eles. Estas diferenças se fazem presentes quando analisamos as linguagens de programação adotadas, principalmente quando se trata de linguagem de máquina. Abrindo computadores com tipos de CPU's diferentes, encontramos também diferenças fundamentais quanto a hardware, ou seja, a maneira como o computador é construído. Em resumo, podemos dizer que o tipo da CPU empregado define as características das montagens e da programação dos computadores em questão.

No caso dos micros da linha APPLE, a CPU empregada é a 6502, objetivo deste livro. Também encontramos esta CPU nos micros da linha ATARI e COMMODORE. Já, por exemplo, os micros da linha MSX ou Sinclair utilizam a CPU Z80, portanto incompatível com os micros anteriormente citados.

O interior da CPU 6502

Apesar da estrutura física da CPU 6502 não ser de grande interesse do programador em geral, não custa dar uma breve olhada no seu interior. A Fig. 1 mostra os diversos módulos, que constituem a CPU 6502:

- *Gerador de Clock/Controle de temporização*
  nesta parte são gerados todos os sinais de sincronismos necessários para um perfeito funcionamento da CPU

- *Decodificador de instruções*
  aqui o dado presente nas linhas DØ até D7 é testado, para saber se trata-se de uma instrução ou de uma informação simples

- *Lógica de interrupção*
  é a unidade responsável para manipular os sinais de interrupção como RES, NMI e IRQ.

- *Registrador de flags (bandeiras)*
  mantém informações importantes quanto ao resultado da última operação efetuada, indispensável para decisões lógicas. Este registrador será bastante analisado nas páginas seguintes.

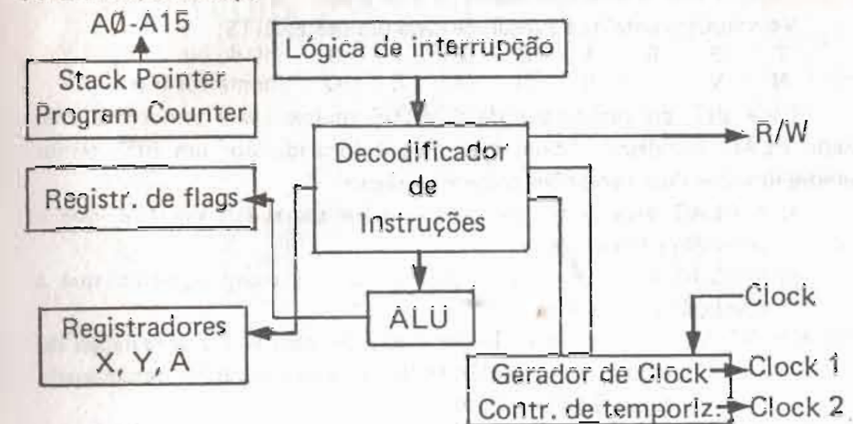
- *registrador de índex X e Y*
  podem ser manipulados como variáveis comuns, ou ainda como índex para instruções envolvendo endereços indexados.

- *Acumulador*
  é uma variável constantemente usada para operações lógicas e aritméticas

- *Stack pointer*
  armazena o endereço onde se encontra o topo da pilha do sistema

- *ALU = Aritmetic Logic Unit (Unidade Aritmética e Lógica)*
  todas as operações lógicas e aritméticas são realizadas por este módulo.

- *Program Counter*
  executando uma rotina, esta variável contém o endereço da próxima instrução a ser executada



EXERCÍCIOS

Assinale as afirmações corretas e complete onde for necessário:

( ) *A CPU 6502 pode ser substituída pela CPU Z80.*
( ) *A linha de dados é formada por 8 BITS.*
( ) *Os endereços são formados com 16 BITS.*
( ) *16 BITS formam o Clock do sistema.*

Nos micros APPLE, COMMODORE e ATARI a CPU empregada é a . . . .
O registrador de flags mantém . . . . importantes.
A variável mais usada é o . . . .

O STATUS da CPU 65Ø2

Antes de nos preocuparmos com as instruções da CPU 6502 em si, temos de falar um pouco do registrador de STATUS desta CPU.

Como você sabe, existem na linguagem BASIC comandos envolvendo decisões que variam de acordo com algum resultado ou condição obtida. Podemos por exemplo ter uma decisão como 'IF A = Ø THEN M = 1ØØ'. Estamos, neste caso, testando primeiro se a variável A contém o valor Ø. Somente se esta condição for verdadeira, a variável M assumirá o valor 1ØØ.

Este tipo de testes, fundamentais para obter decisões também são possíveis em linguagem de máquina. Para armazenar as informações referentes aos últimos resultados (ex. se o resultado é zero, negativo, ou positivo), a CPU utiliza uma memória especial capaz de manter 8 BITS. Esta memória, ou seja, registrador, é denominada STATUS do 65Ø2. Cada um dos oito BITS

representa uma condição, isto é, uma informação específica de acordo com o resultado da última operação lógica ou aritmética.

Veremos,em detalhe,a função de cada um destes BITS:

7	6	5	4	3	2	1	Ø	nº do bit
N	V	–	B	D	I	Z	C	nome dos bits

Cada BIT do registrador de STATUS muitas vezes também é chamado FLAG (bandeira). Como cada flag é formado por um BIT, temos sempre uma das duas condições possíveis a seguir:

a) o FLAG está com nível Ø (bandeira abaixada): significa que a condição é falsa

b) o FLAG está com nível 1 (bandeira levantada): significa que a condição é verdadeira

Flag N = BIT 7: indica o sinal do resultado. A letra N é a abreviação do termo inglês NEGATIVE, às vezes também denominado como flag S (= SIGN)

Flag V = BIT 6: sinaliza um "Overflow" ocorrido

— = BIT 5: este BIT do STATUS não está sendo usado

Flag B = BIT 4: indica quando uma instrução BREAK (vide Estudo das instruções) foi executada

Flag D = BIT 3: quando o modo decimal for selecionado, este flag assumirá o nível lógico 1

Flag I = BIT 2: flag responsável para modos de interrupção

Flag Z = BIT 1: sinaliza quando o último resultado é zero ou não

Flag C = BIT Ø: sinaliza um "CARRY" ocorrido (vide Estudo das instruções)

A utilização dos flags acima fica mais clara durante o estudo das instruções da CPU 65Ø2, e não deve preocupar por enquanto o leitor. É porém importante ter em mente, que após cada operação lógica ou aritmética realizada, os flags são reajustados conforme o resultado obtido. Esta informação permanece inalterada até a próxima operação ou aplicando instruções específicas para manipular os flags diretamente, como veremos nas próximas páginas.

EXERCÍCIOS

Assinale as afirmações corretas e complete onde for necessário:

( ) *O STATUS é uma variável do BASIC no APPLE.*

( ) *O flag é uma bandeira, que pode ter dois níveis diferentes.*

( ) *O STATUS tem uma função secundária na CPU 6502.*

Cada . . . . é um flag do registrador de STATUS.

O nível lógico 1 indica uma condição. . . . .

Quando o último resultado é zero, o flag. . . . . ficará com nível 1.

A CPU 6502 tem. . . . . bandeiras à nossa disposição.

capítulo 2

lda, ldx, ldy

Preparando o microcomputador

Chegou o grande momento: a partir de agora iremos conhecer todas as instruções da CPU 6502 – uma a uma – levando você aos poucos a ponto de poder desenvolver seus próprios programas em linguagem de máquina.

Para entrar no monitor Assembler do seu Apple, digite agora

```
CALL - 151
```

Aparecerá um novo cursor (*), indicando que já estamos no monitor Assembler. Para os usuários do TK 2000, basta digitar CALL-159 ou simplesmente LM, e apesar do cursor ser diferente (?), prevalecem os mesmos comandos para ambos os computadores. Exceções são indicadas quando for necessário.

Para retornar ao BASIC, basta digitar a tecla CONTROL e a tecla C ao mesmo tempo, seguido de RETURN.

Neste capítulo falaremos em detalhes sobre "como" e "onde" programar em linguagem de máquina, dando informações válidas para todas as demais instruções abordadas neste livro.

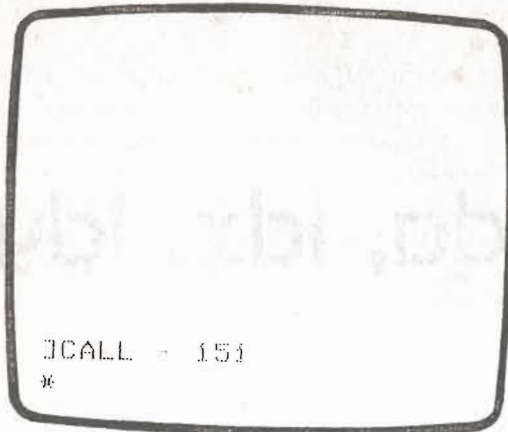


Figura 1 – O cursor do Assembler do Apple.

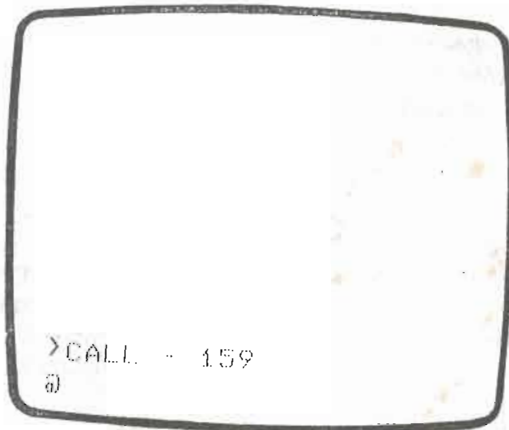
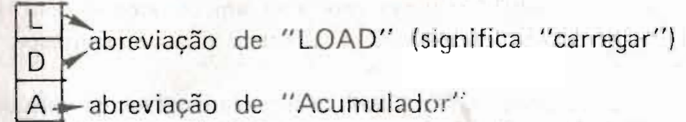


Figura 2 – O cursor do Assembler do Tk 2000

LDA: carregar o acumulador com um valor

As letras LDA formam o *mnemônico* desta instrução, isto é, o símbolo oficial para a operação de carregar o acumulador com um valor qualquer. Não se trata de letras escolhidas aleatoriamente, mas sim de uma abreviação desta operação em inglês:



Na linguagem BASIC temos uma instrução equivalente, que é o "LET".

Nesta altura você talvez já imagine o que realmente acontece num LDA. Mas, para termos certeza, vamos a um exemplo:

Exemplo: Carregar o acumulador A com o valor $3F. (O símbolo $ indica que se trata de um valor hexadecimal.)

Solução: Neste caso, trata-se de um LDA imediato, isto é, o valor $3F será colocado imediatamente no acumulador. O mnemônico completo para este tipo de operação é

LDA # n

O valor n, no nosso exemplo, vale $3F enquanto o símbolo // indica que se trata de uma operação imediata.

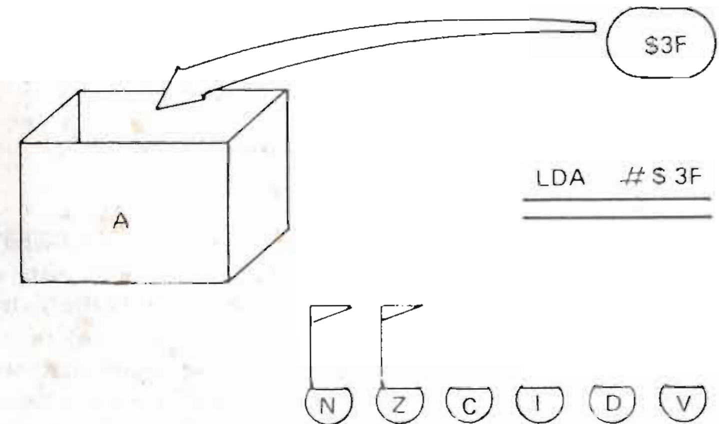


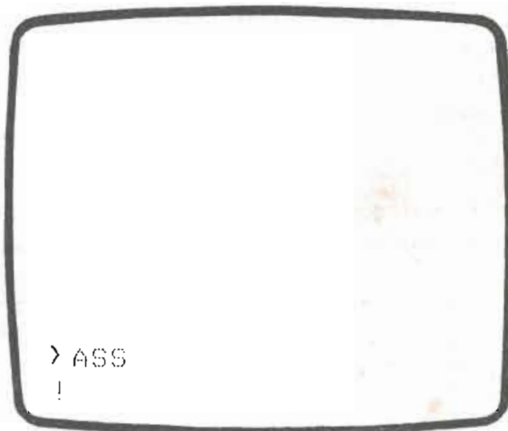
Figura 4.1 – Transferência do valor $3F para o acumulador

Unindo o valor n ao mnemônico, teremos:

```
LDA #$3F
```

No nosso Apple não podemos programar digitando simplesmente LDA #$3F sem o uso de um programa especialmente desenvolvido para este fim (exemplo: LISA). Mas, provavelmente, poucos usuários estão em posse deste ou de um programa similar, motivo pelo qual vamos analisar como "introduzir" esta instrução no computador.

Nota para os usuários do TK 2000: neste computador já existe um programa capaz de programar diretamente com mnemônicos. O acesso a este programa é feito mediante o comando ASS. Consulte o seu manual técnico para maiores informações.



No apêndice C encontramos na tabela I os "OP-CODES" (Códigos) de cada operação da CPU 6502. No nosso caso, para o mnemônico LDA imediato, encontramos o valor A9 (confere?). Isto significa o seguinte:

Cada rotina escrita em linguagem de máquina é composta por valores numéricos, onde cada valor representa uma instrução ou um dado. Usando um programa Assembler (LISA ou mediante o comando ASS no TK 2000), este faz nada mais do que traduzir o mnemônico para seu respectivo OP-Code.

Assim, tendo o valor A9 na memória, a CPU já sabe que deve ser realizado um LDA imediato, porém ainda não sabe o valor de n. Para fornecer à CPU também o valor n é fácil: é simplesmente o próximo valor, após A9.

Desta maneira, os valores numéricos para a operação LDA #$3F são:

```
A9 3F
```

Se o nosso exemplo fosse LDA #$37, então os respectivos valores na memória seriam A9 37.

Como colocar estes valores na memória

Em primeiro lugar, devemos estar no monitor Assembler. Depois é necessário escolher uma parte da memória livre onde serão guardados estes valores. Este endereço, para todas as rotinas apresentados neste livro, será $0300.

Digite agora simplesmente 0300, seguido de RETURN. O computador irá listar agora o conteúdo do endereço $0300:

```
0300 - 00
```

Isto significa que atualmente temos o valor 0 na memória na posição $0300. Digitando RETURN novamente, podemos verificar os demais endereços:

```
0300 - 00
*
AD 00 E0 48 AD 81 C0
*
```

A cada toque da tecla RETURN, serão indicados os conteúdos dos próximos endereços. Se no seu computador não aparecem os mesmos valores acima, não se preocupe. Trata-se de uma parte da memória livre, e provavelmente os valores encontrados são aleatórios sem nenhum sentido.

Vamos agora colocar o nosso primeiro valor $A9 na memória. Para isto digite o endereço seguido de dois pontos (:) e depois o valor A9:

```
*300: A9
```

Confira se este valor realmente foi colocado no endereço 0300:

```
*300
```

e, após o RETURN, encontramos:

```
0300 - A9
```

Simples, não? Resta ainda o segundo valor, no nosso exemplo o $3F. Como o OP-Code A9 já está na posição 0300, o dado 3F ocupará a posição 0301. Portanto, digitamos:

```
*0301: 3F
```

Ao invés de digitar cada valor independentemente, podemos também introduzir todos os valores de uma só vez. Para isto é suficiente indicar o endereço inicial, e após os dois pontos (:) todos os valores hexadecimais:

```
*0300: A9 3F
```

É importante que os valores sejam digitados com *um espaço* entre si, caso contrário a programação não se efetuará.

Vamos ver se realmente temos agora a seqüência A9 3F na memória:

```
*300
```

e, após dois RETURNs, temos na tela:

```
0300 - A9
*
3F 00 E0 48 AD B1 C0
*
```

Os computadores APPLE e TK2000 são capazes de listar os *mnemônicos* equivalentes aos valores numéricos contidos na memória. Para isto, você deve digitar o endereço seguido da letra L:

```
*300L
```


Após o RETURN, temos na tela alguns mnemônicos, entre eles, também a nossa instrução LDA #$3F. Isto é um controle nosso para verificar se digitamos tudo corretamente.

Vamos agora verificar como a CPU coloca o valor $3F no acumulador, pois o que fizemos até agora foi apenas escrever esta operação. Falta "rodar" esta instrução para que o valor $3F seja colocado no acumulador.

Antes de rodar o nosso programa, é preciso colocar o valor $00 após a seqüência A9 3F. Este valor representa o mnemônico BRK, do qual trataremos mais adiante. A vantagem do BRK após a nossa rotina é que o computador lista os conteúdos dos registradores da CPU na tela, permitindo desta maneira a verificação se o acumulador realmente assumiu o valor $3F. Assim, a rotina completa deverá ser digitada da seguinte forma:

```
*0300: A9 3F 00
```

Verificando os mnemônicos com 0300L, temos agora (entre outros) os seguintes mnemônicos na tela:

```
0300 - A9 3F      LDA #$3F
0302 - 00         BRK
   .               .
   .               .
   .               .
```

Para rodar, devemos indicar o endereço inicial, que é no nosso caso o endereço 0300 seguido da letra G:

```
*300G
```

Após o RETURN, o programa será executado e listados os conteúdos dos registradores. Teremos portanto na tela algo assim:

```
0304 - A=3F  X=B4  Y=D8  P=30  S=F0
```

O que nos interessa no momento é o conteúdo do acumulador, que de fato contém o valor 3F, de acordo com a nossa programação.

Tente agora escrever uma rotina que carrega o acumulador com o valor hexadecimal 19.

Semelhante é o procedimento com os registradores de índex X e Y:

LDX: carregar o registrador de índex X com um valor

LDY: carregar o registrador de índex Y com um valor

Exemplo: Colocar em X o valor FF.

Solução: Trata-se de um LDX imediato, cujo OP-CODE é A2. Temos portanto para a instrução LDX # $FF a seguinte seqüência numérica:

```
A2 FF
```

Programamos esta instrução também a partir do endereço 0300:

```
*0300: A2 FF
```

Confira o mnemônico LDX # $FF com:

```
*0300L
```

Para rodar, usamos novamente a letra G após o endereço inicial:

```
*0300G
```

e teremos na tela o seguinte resultado:

```
0304 - A=3F  X=FF  Y=D8  P=B0  S=EE
```

Percebemos que X assumiu o valor FF, enquanto que o acumulador permaneceu inalterado.

Exemplo: Escrever uma rotina que efetue o seguinte:

a) carregar o acumulador com o valor 11;
b) carregar o registrador de índex X com o valor 22;
c) carregar o registrador de índex Y com o valor 33.

Solução: Os mnemônicos para o exemplo acima são:

a) LDA # $11 – o código é A9 11
b) LDX # $22 – o código é A2 22
c) LDY # $33 – o código é A0 33

Realizando várias operações ao mesmo tempo, basta colocar na memória todos os códigos equivalentes, um após o outro. No final colocaremos novamente o valor $00, que possibilita fazer uma leitura dos conteúdos dos registradores.

Programamos portanto da seguinte maneira:

```
*0300: A9 11 A2 22 A0 33 00
```

Verificamos os mnemônicos usando o modo L:

```
*300L
```

Confere tudo? Ótimo, então vamos rodar a nossa rotina:

```
*0300G
```

Se tudo estiver certo, encontramos em A o valor 11, em X o valor 22 e em Y o valor 33. Os conteúdos de P e S não nos interessam no momento.

Tudo claro até agora? Então resolva os seguintes exercícios:

1. Carregue o acumulador com o valor 5A e o registrador de índex Y com o valor 13. Após rodar o programa, os conteúdos destes registradores devem aparecer na tela.
2. Carregue o registrador de índex X com o valor F8, o registrador de índex Y com o valor 00, e finalmente o acumulador com o valor 24.

LDA página zero

LDX página zero

LDY página zero

Pela primeira vez temos diante de nós a expressão "página zero". Veremos que em muitas instruções da CPU 6502 há referências a esta página, que em particular é uma das principais características desta CPU.

Quando falamos em página zero, estamos nos referindo a memória do computador entre os endereços $0000 e $00FF. Temos nesta "página" da memória um total de 256 posições, que na maioria são usadas pela CPU para organizar o sitema operacional do nosso APPLE. Portanto, qualquer alteração nesta página poderá provocar uma interrup-

ção no funcionamento normal do micro, exigindo um RESET ou até mesmo a necessidade de desligar e ligar o computador para que o sistema operacional volte ao normal.

Todavia, uma leitura dos valores contidos nesta página zero não tem conseqüências.

A leitura desta página, em linguagem de máquina, é feita mediante a instrução LDA página zero. Neste caso, o valor de um determinado endereço da página zero será copiado no acumulador. Igualmente, ao usar LDX página zero ou LDY página zero, os valores lidos serão copiados em X ou Y.

Exemplo: Carregar no acumulador o valor contido no endereço $00B3.

Solução: Em primeiro lugar, como estamos operando com a página zero, que está contida entre $0000 e $00FF, indicamos apenas os *últimos* dois números do endereço, já que os primeiros dois valores sempre estão com o valor 0.

A instrução correta é LDA $B3. Observe que a CPU nunca se confunde com a operação de carregar o acumulador com o valor $B3, já que para esta operação usamos o símbolo # , que agora foi omitido.

O código para LDA página zero é A5, e como queremos trabalhar com o endereço B3, a seqüência correta será:

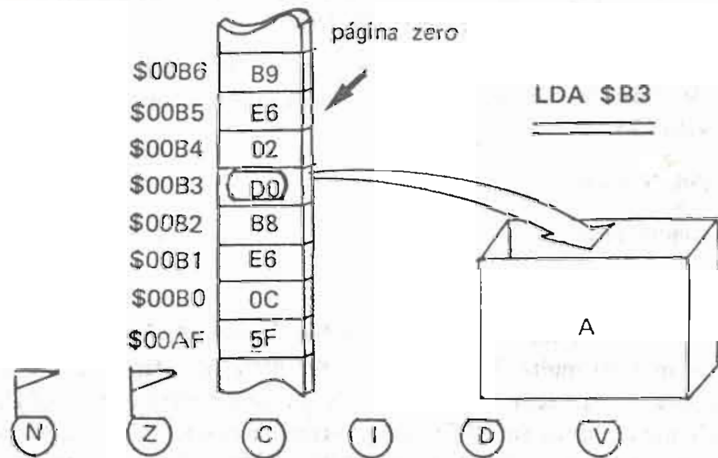


Figura 4.2 – Transferência do valor contido no endereço $00B3 da página zero para o acumulador.

Vamos programar o micro com esta seqüência:

```
*0300: A5 B3 00
```

(o valor 00 permite ler os registradores)

Confira o mnemônico LDA $B3:

```
*0300L
```

Rodando esta rotina com 0300G, teremos na tela o seguinte:

```
*0300G
0304 - A=D0  X=22  Y=33  P=B0  S=EA
*
```

Resta saber se no endereço B3 da página zero realmente se encontra o valor D0, que está agora no acumulador. Para isto, digite:

```
*00B3
```

seguido de RETURN. De fato, teremos:

```
00B3 - D0
```

Se você encontrar um outro valor no acumulador, este também deve estar contido no endereço B3 da página zero.

Vamos ver se o mesmo procedimento se aplica também aos registradores de índex X e Y?

Exemplo: Carregar X com o conteúdo do endereço $00 e Y com o conteúdo do endereço $05.

Solução: Para o LDX página zero encontramos na tabela I do apêndice C o código A6, e para LDY página zero o código A4. A nossa rotina portanto será da seguinte maneira:

```
LDX $00 = A6 00
LDY $05 = A4 05
```

Programamos a partir do endereço $0300:

```
*0300: A6 00 A4 05 00
```

Não se esqueça de conferir os mnemônicos usando 0300L!
Rodando agora esta rotina, devemos ter na tela:

```
*0300G
0306 - A=D0  X=4C  Y=DB  P=B0  S=EB
*
```

Conferindo os conteúdos dos endereços 00 e 05, temos:

```
*00
00 - 4C
*05
05 - DB
```

EXERCÍCIOS:

1. *Assinale as respostas corretas:*

   ( ) *A instrução LDA #$45 carrega tanto o acumulador como também os registradores de índex X e Y com o valor $45.*

   ( ) *O valor de A após a instrução LDA #$00 sempre será 0.*

   ( ) *Programando a seqüência A9 A6, é equivalente ao mnemônico LDA $A6.*

2. *Escreva uma rotina que carregue X com o valor $73, e Y o conteúdo do endereço $004A da página zero.*

3. *Qual será o valor de A após a instrução LDA $45?*

4. *Escreva uma rotina que carregue o acumulador com o valor 00, e os registradores de índex X e Y com o valor contido no endereço $0099.*

LDA página zero,X

Aqui temos uma instrução que envolve tanto o acumulador como o registrador de índex X. Neste caso, a instrução é semelhante ao LDA página zero, só que agora deve ser *somado* ao endereço da página zero o valor de X.

Exemplo:
```
LDX #$02
LDA $42,X
```

O valor de X é 02, e o endereço definido é $42. Portanto será copiado no acumulador não o conteúdo do endereço $42, mas sim o conteúdo do endereço $42 + $02 (valor de X) = $44.

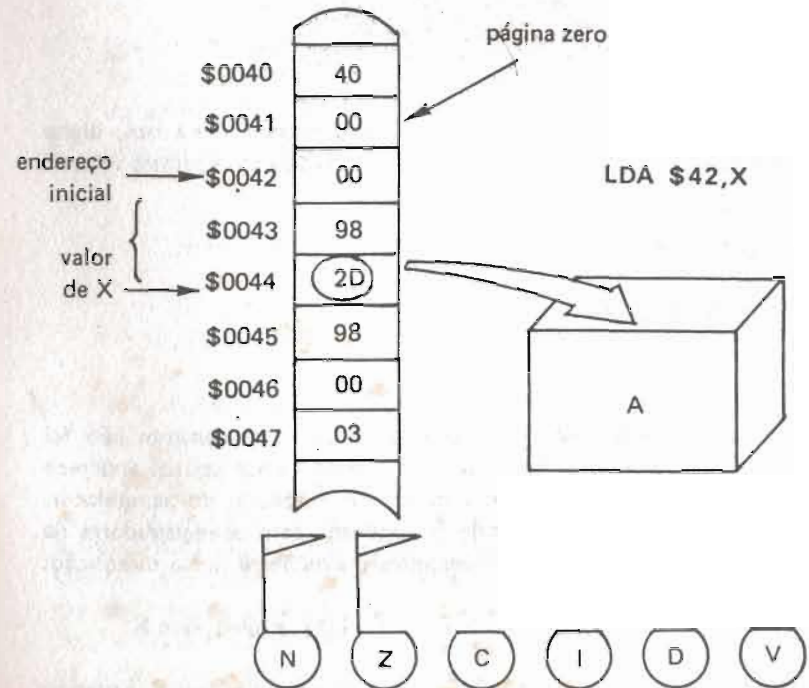


Figura 4.3 – Transferência do valor $3F para o acumulador.

Vamos ver isso na prática?

O código para LDA página zero,X é B5, sendo a rotina completa conforme segue:

```
LDX #$02 = A2 02
LDA $42,X = B5 42
```

```
*0300: A2 02 B5 42 00
```

Confira os mnemônicos digitando:

```
*0300L
```

Vamos rodar agora a nossa rotina e verificar o conteúdo do acumulador:

```
*0300G
0306 - A=2D  X=02  Y=DB  P=30  S=E6
*
```

Vamos verificar os conteúdos da página zero. Para isto, digite 0042 seguido de dois RETURNs. Teremos na tela os seguintes valores:

```
*0042
0042 - 00
*
98 2D 2D 02 DB
*
```

No endereço $42 temos o valor 00, que obviamente não foi copiado no acumulador. Verificando o segundo valor após o endereço 0042, encontramos 2D, de acordo com o conteúdo do acumulador.

A mesma sistemática pode ser aplicada para os registradores de índex X e Y, tendo assim as seguintes instruções à nossa disposição:

LDX página zero,Y e LDY página zero,X

Observe que ao usar X, o registrador de índex deve ser Y, e usando o Y, o registrador de índex passa a ser o X. Já, usando o acumulador, somente o registrador X poderá ser usado.

Exemplo: Carregar X com o conteúdo do endereço $0055 + $03.

Solução:

```
LDY #$03 = A0 03
LDX $55,Y = B6 55
```

```
*0300: A0 03 B6 55 00
```

Tente conferir você mesmo se o valor encontrado em X é realmente o mesmo valor encontrado no endereço $0058 ($55 + valor de Y).

O código para LDX página zero,Y é B6
e para LDY página zero,X é B4

EXERCÍCIOS:

1. *Carregue X com o valor $32 e o acumulador com o valor contido no endereço $0056.*

2. *Realize um LDX página zero,Y, onde o valor de Y deve ser igual ao valor contido no endereço $0013 e o endereço inicial da página zero é $0000.*

3. *Assinale as afirmações corretas:*

   ( ) *Somente os registradores X e Y podem ser usados para representar registradores de índex.*

   ( ) *Se existir a instrução LDA página zero,X, então também deve existir a instrução LDA página zero,Y.*

   ( ) *Se nós temos a instrução LDX $68,Y, onde Y contém o valor 00, então esta instrução é equivalente a instrução LDX $68.*

Mas os LD's não param por aqui. Até agora vimos LDA's imediatos, LDA's página zero e LDA's página zero com índex. Mas como faremos para transferir o conteúdo de um endereço da memória acima do limite da página zero? Pois usando o LDA página zero com índex, o maior endereço possível é quando tanto o valor do registrador de índex como o endereço base forem "FF".

LDX # $FF

LDA $FF,X = endereço máximo real 01FE!

Para estes casos, temos o

LDA absoluto

Exemplo: transferir o conteúdo do endereço $ 3FC9 para o acumulador.

Solução: `LDA $3FC9 (digite 0300: AD C9 3F)`

Simples também, não? Talvez a esta altura, você esteja pensando que um LDA página zero (Ex. LDA $34) poderia ser substituído por um LDA absoluto (LDA $0034). Examinando estas duas instruções, percebemos que realmente é realizada a mesma operação, com a diferença que o LDA absoluto requer *sempre* três endereços na memória, enquanto a instrução LDA página zero requer apenas dois endereços. Com isto, ao trabalhar com a página zero, ganhamos não apenas mais memória, mas também uma velocidade de execução maior.

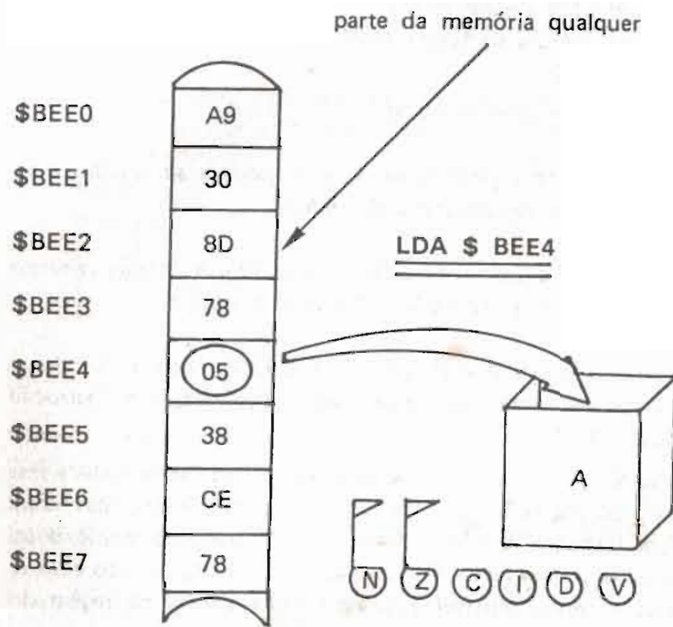


Figura 4.4 – Transferência de um valor contido numa parte qualquer da memória para o acumulador.

Como para o acumulador, o LD absoluto também funciona para os registradores de índex X e Y:

LDA absoluto – *OP-Code:AD*

LDX absoluto – *OP-Code:AE*

LDY absoluto – *OP-Code:AC*

Importante

Pela primeira vez estamos usando endereço de 4 Bytes nas nossas instruções. Você já deve ter percebido que no exemplo anterior (LDA $3FC9) digitamos a seqüência AD C9 3F.

Quanto ao valor AD não temos dúvida, pois representa o OP-Code (código) para a instrução LDA absoluto. Os dois bytes a seguir definem o endereço, no nosso caso $3FC9. Por que então escrevemos após o código AD os valores C9 3F e não 3F C9?

Tente você mesmo, programando a partir do endereço $0300 a seqüência aparentemente certa:

```
0300: AD 3F C9 00
```

A seguir, vamos dar uma olhada no respectivo mnemônico através do comando 0300L:

```
*0300L
0300 - AD 3F C9        LDA $C93F
0303 - 00              BRK
```

Percebemos que ao programar endereços, a ordem dos dois bytes na memória *sempre* estão invertidos. Assim sendo, ao escrever uma rotina em linguagem de máquina com OP-Codes, nunca se esqueça desta ordem inversa! Aliás, esta inversão é válida para todas as CPU's no mercado, não apenas para o 6502.

Escrevendo as rotinas com mnemônicos, o endereço deve constar de acordo com a realidade (Escreve-se LDA $1234, mas programa-se AD 34 12).

EXERCÍCIOS:

1. *Carregue no acumulador o valor $23 e no registrador de índex Y o conteúdo do endereço $F550.*

2. *Coloque em X o valor contido no endereço $003F. Indique as duas maneiras possíveis.*

3. *Substitua a operação LDX $01BB (que é um LDX absoluto) por um LDX página zero,Y (o registrador de índex Y deve valer $FF).*

4. *Assinale as afirmativas corretas:*

   ( ) Todo LDA página zero e LDA página zero,X podem ser substituídos por um LDA absoluto.

   ( ) Todo LDA absoluto pode ser substituído por um LDA página zero,X.

   ( ) Somente alguns LDA absoluto podem ser substituídos por um LDA página zero,X.

   ( ) Como o modo absoluto requer três posições na memória, trata-se de uma operação mais poderosa e é portanto mais rápida na execução.

   ( ) Todos os endereços nas instruções são programados na ordem inversa.

Também para o LD absoluto podemos usar os registradores de índex, que somados ao endereço dado na instrução resultam num endereço efetivo desta operação.

LDA absoluto,X
LDA absoluto,Y

O código para o LDA absoluto,X é BD, e para o LDA absoluto,Y é B9.

Exemplo: Colocar no acumulador o valor contido no endereço $F300 + o valor $34, que deve estar em X.

Solução:
```
LDX $34      = A2 34
LDA $F300,X  = BD 00 F3
```
(lembre da ordem inversa do endereço)

```
*0300: A2 34 BD 00 F3 00
```


Rodando a rotina com 0300G, temos os seguintes valores em A e X:

```
*0300G
0307 - A=AF  X=34  Y=00  P=B0  S=F0
*
```

Listando o conteúdo do endereço F300, achamos o valor F4, que de acordo com a nossa instrução, não deverá aparecer no acumulador. Tente descobrir em que endereço a instrução localizou o valor AF (conteúdo de A). Uma dica: a instrução LDA página zero,X é semelhante neste ponto

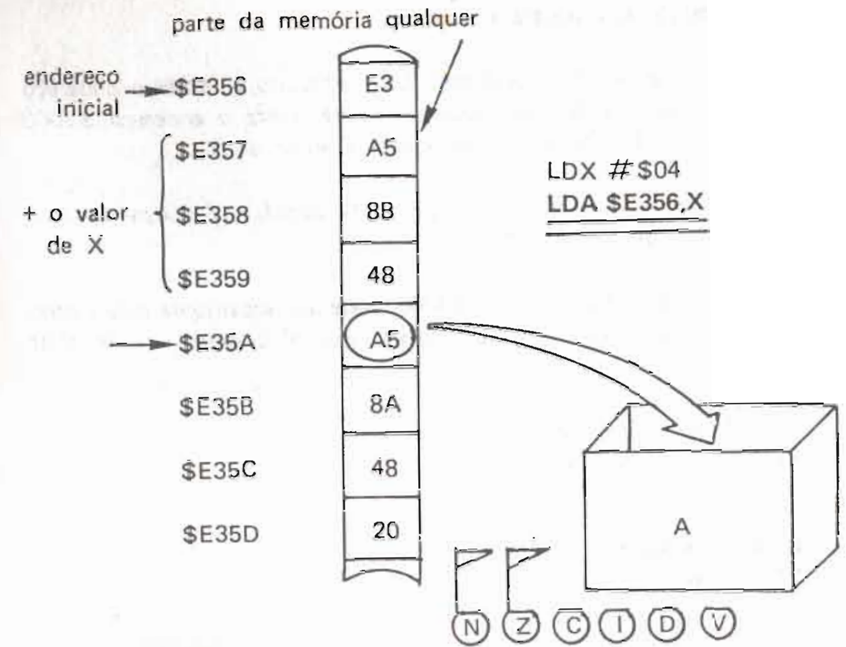


Figura 4.5. – Transferência de um valor contido numa parte qualquer da memória para o acumulador, no modo absoluto.

Este modo absoluto com um registrador de índex também é válido para o X e o Y:

LDX absoluto,Y – *OP-Code: BE*

LDY absoluto,X – *OP-Code: BC*

EXERCÍCIOS:

1. *A instrução LDA página zero,X pode ser substituída pela instrução LDA absoluto,X. Esta afirmação está correta?*

2. *Resolva o seguinte problema:*

   "A" deve ser carregado com o valor $20, e o registrador de índex Y com o conteúdo do endereço $A320. A seguir, deve ser aplicada a seguinte operação: LDA $E555,Y. Qual é o valor final de "A"?

3. *Assinale as afirmações corretas:*

   ( ) Se temos a operação LDX absoluto,Y, onde o endereço base é $1000, podemos operar entre o endereço $1000 e $10FF mudando apenas o valor de Y.

   ( ) Como existe a instrução LDX absoluto,Y, existe também a instrução LDX absoluto,X.

   ( ) A instrução LDA $4000 pode ser substituída pela instrução LDA $3FF0,Y, sendo que Y contém o valor $10.

E, para finalizar os LD's, veremos ainda um modo exclusivo para o acumulador, que é o

LDA (indireto,X)
LDA (indireto),Y

Observe que há uma forma *diferente* de anotação dos LDA's indiretos. Em primeiro lugar usamos parênteses depois do mnemônico LDA. Depois, de acordo com o registrador de índex utilizado, temos um comportamento diferente do modo indireto, como veremos agora:

LDA (indireto,X) — *OP-Code:* A1

Observe que ao usar o X, este registrador de índex está dentro dos parênteses, junto com o endereço da página zero. Não é possível colocar nesta instrução um endereço acima de $00FF, motivo pelo qual escrevemos apenas os últimos dois valores deste endereço (de $00 até $FF).

Ao aplicar um LDA (indireto,X), a CPU soma em primeiro lugar ao endereço dado o valor de X, e a seguir coloca em A o valor *contido no endereço da página zero definida.*

Confuso, não? Vamos ver isto na figura a seguir, que representa a instrução LDA ($004D,X), onde X contém o valor $03:

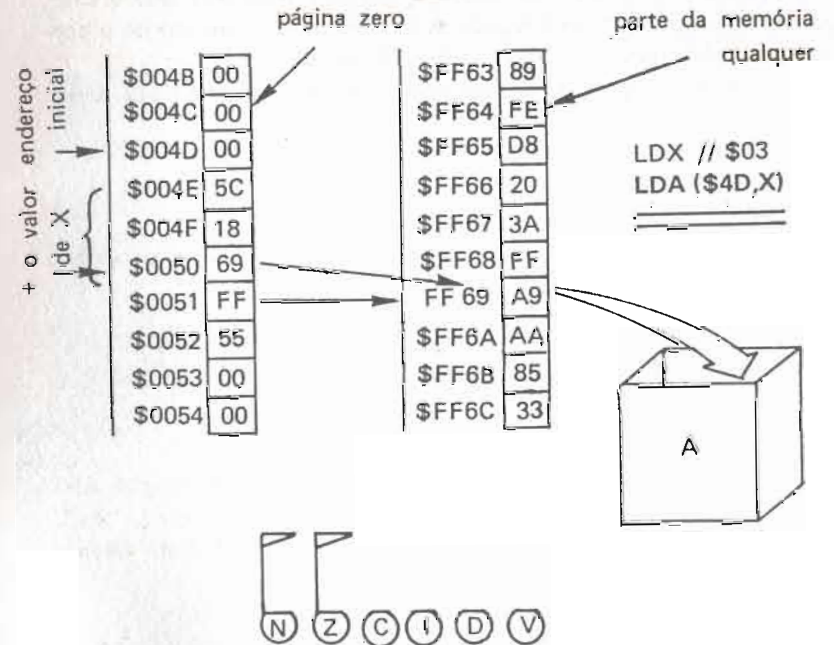


Figura 4.6 — Transferência de um valor contido na página zero para o acumulador no modo indireto.

Observando atentamente a figura 4.6 vemos que existem dois trechos da memória do computador. O trecho à esquerda representa a página zero, e o trecho à direita uma parte qualquer da memória (que poderia ser inclusive a própria página zero).

Na nossa instrução definimos o endereço base $4D, e somando o valor de X, temos o endereço final $50. Neste endereço encontramos o valor $69.

Ao contrário da instrução LDA $4D,X, *onde o valor encontrado $69 entraria no acumulador,* agora, através do modo indireto, o endereço contido será lido a partir do endereço $50. Ora, como cada endereço na memória é formado por dois bytes, deve ser lido pela CPU também o valor contido no endereço seguinte ($51). Temos portanto os valores $69 e $FF na página zero.

Você recorda-se da ordem inversa dos bytes de um endereço na memória? Pois bem, na memória temos a seqüência $69 e $FF, portanto o endereço encontrado será $FF69. E é justamente o conteúdo deste endereço $FF69 que será copiado em A.

Tente você mesmo, escrevendo estas instruções no seu micro:

```
*0300: A2 03 A1 4D 00
```

Confira os mnemônicos com 0300L.

A seguir, rodamos este programa com 0300G, e teremos na tela algo assim:

```
*0300G
0306 - A=A9  X=03  Y=00  P=B0  S=F0
*
```

Vamos conferir? Para listar os conteúdos dos endereços $50 e $51, podemos digitar 0050.0051, onde o ponto significa um "até", isto é, serão listados os conteúdos dos endereços $0050 até $0051:

```
*0050.0051
0050 - 69 FF
*
```

E, olhando rapidamente para o endereço FF69, achamos o valor A9, que após rodar a nossa rotina é encontrado em A.

O modo indireto é um pouco mais complicado do que os modos vistos até agora, e se ainda tiver dúvida sobre o seu funcionamento, estude com calma a figura 4.6.

Utilizando o registrador de índex Y no modo indireto, temos a seguinte anotação:

LDA (indireto),Y — *OP-Code:* B1

Reparou que o Y está fora dos parênteses, enquanto na instrução com X, este ficava dentro dos parênteses? Bem, isto significa que após definir o endereço base (dentro da página zero), primeiro será lido o endereço apontado pelo endereço base, e a seguir somado ao novo endereço encontrando o valor de Y.

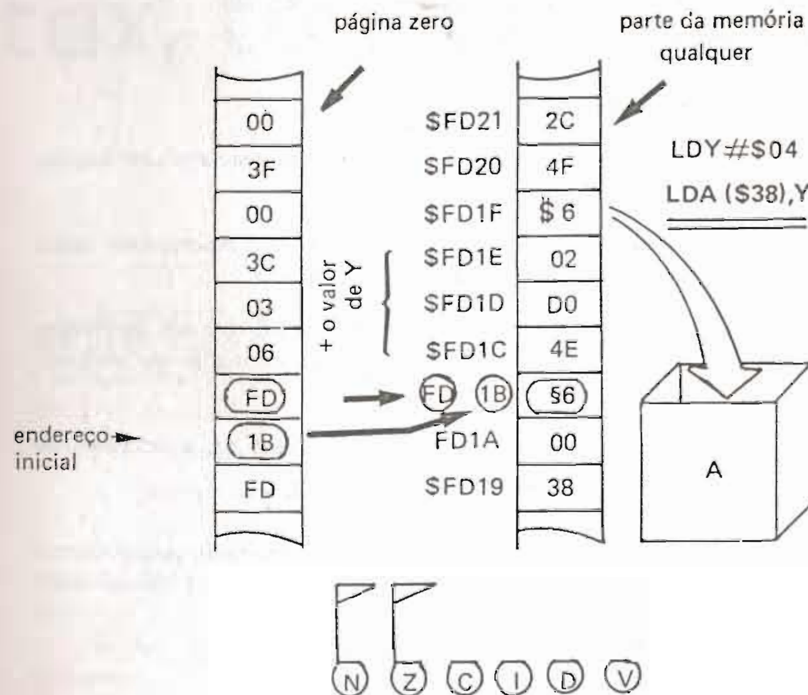


Figura 4.7 — Transferência de um valor contido numa parte qualquer da memória para o acumulador.

Na figura 4.7, podemos verificar o que acontece após:

```
LDY #$04     = A0 04
LDA ($38),Y = B1 38
```

Definimos o endereço base $0038, que é um endereço dentro da página zero. Lá encontramos o valor 1B, e no endereço seguinte o valor FD, formando desta maneira o endereço FD1B. A este endereço encontrado, a CPU soma o valor de Y, que é 04, definindo assim o endereço final $FD1F ($FD1B + $04).

O valor contido neste endereço final ($E6) é colocado no acumulador. Se nós quisermos usar o modo indireto, sem que haja uma soma a um endereço final diferente do endereço encontrado na página zero, devemos simplesmente carregar o registrador de índex Y com o valor $00.

EXERCÍCIOS:

1. *Assinale as afirmações corretas:*

   ( ) *o endereço $AA31 na memória é programado pela seqüência AA31.*

   ( ) *o modo indireto funciona tanto com o acumulador como com os registradores do índex X e Y.*

   ( ) *no modo indireto, onde X aparece dentro dos parênteses, o endereço final é encontrado no conteúdo do endereço base + valor de X.*

   ( ) *não existe a instrução LDA (indireto,Y), e também não existe a instrução LDA (indireto),X.*

2. *Coloque em A o conteúdo do endereço apontado pelo endereço $45 da página zero (uso como endereço base o endereço $40).*

3. *Coloque em X o conteúdo do endereço $FD1B.*

capítulo 3

tax, tay, txa, tya

Temos nesta parte as transferências de dados de um registrador para outro. A letra "T" nos mnemônicos acima significa "Transfere", e as duas letras a seguir, os registradores envolvidos. No caso da instrução TAX, o critério é o seguinte:

T transfere

A o conteúdo de A

X para o registrador de índex X

Em BASIC, o equivalente para a instrução TAX seria LET X = A.

Dentro desta lógica, o que será que a instrução TYA realiza? Exatamente, o TXA transfere (copia) o conteúdo de Y em A.

Os códigos das operações de transferência entre os registradores são os seguintes:

TAX – OP-Code: AA	(transfere o conteúdo de A para X)
TAY – OP-Code: A8	(transfere o conteúdo de A para Y)
TXA – OP-Code: 8A	(transfere o conteúdo de X para A)
TYA – OP-Code: 98	(transfere o conteúdo de Y para A)

Exemplo de uma transferência do conteúdo de A para o registrador de index X.

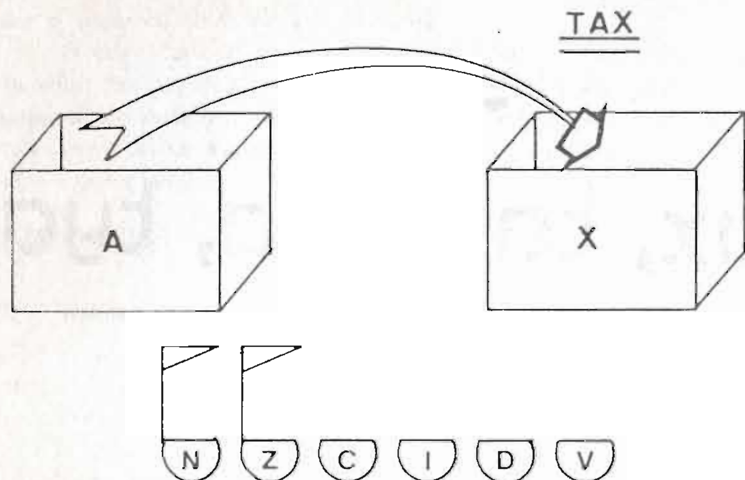


Figura 5.1 – Transferência de um valor contido no acumulador para o registrador de index X.

Exemplo: Escrever uma rotina que coloca em A o conteúdo do endereço $F0B0. A seguir, este valor encontrado deve ser copiado em X e Y.

Solução: a) colocar em A o conteúdo do endereço $F0B0 . . . . LDA $F0B0 AD B0 F0

b) copiar este valor em X . . . . . TAX AA

c) copiar este valor em Y . . . . . TAY A8

d) para consultar os registradores, colocamos um BRK . . . . BRK 00

Na prática, programamos conforme segue:

```
*300: AD B0 F0 AA A8 00
```

e conferimos os mnemônicos com 0300L. Se tudo estiver em ordem, podemos rodar a nossa rotina:


```
*0300G
0307 - A=E9 X=E9 Y=E9 P=B0 S=EA
*
```

Temos portanto o valor $E9 tanto em A como também em X e em Y. Será que no endereço $F0B0 também existe este valor? Certamente, sim. Tente você mesmo:

```
*F0B0
F0B0 - E9
*
```

Pelo que vimos, nas operações de transferência sempre usamos o acumulador. Qual seria o procedimento para copiar em X o conteúdo de Y?

Como não existe a operação TYX, devemos usar o acumulador para atingir o resultado desejado. Inicialmente, o valor que deve ser colocado em X, está em Y. Devemos então primeiramente colocar o conteúdo de Y no acumulador com a instrução TYA. Agora, com o valor em A, a transferência para X fica fácil: TAX.

Assim sendo, para copiar Y em X, devemos programar:

```
TYA 98
TAX AA
```

e, copiando o conteúdo de X em Y, temos respectivamente:

```
TXA 8A
TAY A8
```

Além destas instruções de transferência, existem ainda o TAS e o TSA, que trataremos mais tarde por envolverem o "registrador" S.

Observamos que após a transferência, o conteúdo do registrador que forneceu a informação permanece inalterado!

EXERCÍCIOS:

1. Assinale as afirmações corretas

( ) O conteúdo de A pode ser copiado tanto em X como em Y.

( ) Podemos transferir o conteúdo de X para A sem auxílio do Y.

( ) É possível transferir o conteúdo de Y para X sem o uso de A.

2. *Escreva uma rotina que coloque em A o conteúdo do endereço $0032. A seguir, este mesmo valor deve ser transferido para X.*

3. **Desafio:** *Sabemos que não existe o modo indireto para os registradores de índex X e Y. Como deve ser o programa para colocar em X o conteúdo do endereço apontado pelo endereço $73 da página zero?*

capítulo 4

inc, inx, iny

As operações cujos mnemônicos iniciam com as letras "IN" modificam o conteúdo de um determinado registrador ou endereço, somando ao valor atual o valor 1.

INX – OP-Code: E8 (incrementa o conteúdo de X por 1)

INY – OP-Code: C8 (incrementa o conteúdo de Y por 1)

Assim, se o conteúdo inicial de X era $45, após a operação INX o valor passa a ser $46. Programando INX novamente, o conteúdo de X será $47.

Em BASIC, o equivalente seria LET X = X + 1.

Vamos a um exemplo prático: Colocar em Y o valor $10 e realizar cinco vezes a operação INY:

Solução:

```
LDY #$10    A010
INY         C8
INY         C8
INY         C8
INY         C8
INY         C8
```

Programamos a partir do endereço 0300:

```
*0300: A0 10 C8 C8 C8 C8 C8 00
*
```

Após 0300G, teremos o valor $15 no registrador de índex Y. Confere? Infelizmente esta operação não pode ser usada com o acumulador. Assim, para incrementar o conteúdo do acumulador, devemos primeiro transferir seu conteúdo para o X ou para o Y, aplicando a seguir um INXou INY, e devolvendo depois o valor incrementado ao acumulador.

INC – incrementa conteúdos da memória

Além dos registradores de índex X e Y, também podemos incrementar diretamente os valores contidos na memória do registrador sem a necessidade de carregá-los primeiramente num registrador. Para estes tipos de incrementos temos as seguintes variações:

INC página zero – OP-Code: E6

INC página zero,X – OP-Code: F6

INC absoluto – OP-Code: EE

INC absoluto,X – OP-Code: FE

Exemplo: Devemos incrementar o conteúdo do endereço $7F duas vezes.

Solução: Como o endereço $7F encontra-se na página zero, vamos usar a instrução INC página zero:

```
INC $7F        E6 7F
INC $7F        E6 7F
```

Uma outra solução seria carregar primeiro o valor do endereço $7F para um registrador de índex (por exemplo o Y), e realizar dois incrementos antes de devolver este valor ao endereço $7F. Será que vale a pena? Vamos ver:


```
LDY $7F        A4 7F
INY            C8
INY            C8
```

(aqui entra a instrução que coloca o valor de Y na memória).

Mesmo ainda não conhecendo a instrução para devolver o valor ao endereço $7F, já vimos que a segunda solução ocupa mais bytes do que a primeira, portanto inviável.

Exemplo: O registrador X contém o valor 10 e o endereço base $55 da página zero. Qual é o endereço cujo conteúdo será incrementado após a instrução INC $55,X?

Solução: Já sabemos o que significa "página zero,X". Se não souber, volte algumas páginas e consulte no capítulo 2 a instrução LDA página zero,X.
Como X contém o valor $10, e o endereço base é $55, temos portanto o endereço final $65. Também aqui vamos ver isto na prática:

A nossa rotina será:

```
LDX #$10       A2 10
INC $55,X      F6 55
```

E programando:

```
*0300: A2 10 F6 55 10
```

Não rode ainda! Verifique primeiro o conteúdo do endereço $65:

```
*0065
0065 - 00
*
```

Agora, digite 0300G e verifique novamente o conteúdo do endereço $65:

```
*0300G
0306 - A=00  X=10  Y=00  P=30  S=EE
*0065
0065 - 01
*
```

Realmente, incrementamos o valor em $65 que antes era $00, e agora é 01. Repetindo o comando 0300G, teremos o valor $02 no endereço $0065. Verifique!

Às vezes precisamos incrementar o conteúdo de um endereço que não está na página zero. Neste caso, teremos de usar o INC absoluto.

Exemplo: Incrementar o conteúdo do endereço $A100.

Solução: `INC $A100        EE 00 A1`

(Nunca esqueça de inverter os dois bytes que formam o endereço).

Semelhante ao INC página zero,X, temos para endereços absolutos também a possibilidade de usar o registrador de índex X:

`INC $2000,X` (o conteúdo de X é $3F)

Nesta instrução, iremos incrementar o valor contido no endereço $203F (somando o endereço base $2000 com o valor de X).

Exemplo: Escrever uma rotina que coloque no acumulador o 1º valor contido na página zero. A cada comando 0300G, o acumulador deve mostrar o próximo valor na página zero.

Solução: Para colocar em A o conteúdo de um endereço da página zero, em princípio poderíamos usar a instrução LDA página zero. Todavia, no exemplo pede-se sempre o conteúdo do próximo endereço da página zero, a cada comando 0300G. A solução é usar o registrador de índex X, que deverá ser incrementado cada vez que se roda o nosso programa. Desta maneira, a nossa rotina fica assim:

1ª rotina (define X)

```
LDX #$00        A2 00

*0300: A2 00 00
*
```

Seguido de 0300G. Esta rotina preparatória era necessária para levar o registrador de índex para $00. Não podemos incluir esta operação na nossa rotina principal, pois então a operação INX não terá o efeito esperado.

2ª rotina (rotina principal)

```
LDA $00,X       B5 00
INX             E8
BRK             00
```

Como X vale inicialmente $00, o acumulador será carregado com o conteúdo do endereço $00. A seguir incrementamos X, que passa a ter o valor $01. Assim, rodando esta rotina novamente, A será carregado com o conteúdo do endereço $01, e assim por diante.

Programamos então o nosso computador:

```
*0300: B5 00 E8 00
*
```

Confira os mnemônicos usando 0300L!

Anote agora os primeiros conteúdos da página zero num papel e confira os conteúdos de A com suas anotações após cada 0300G.

Exemplo:

Listando os primeiros três conteúdos da página zero:

```
*0000.0002
0000 - 4C 3C D4
*0300G
```

```
0305 - A=4C X=01 Y=00 P=30 S=E8
*0300G
0305 - A=3C X=02 Y=00 P=30 S=E6
*0300G
0305 - A=D4 X=03 Y=00 P=30 S=E4
```

Restou uma dúvida: se o conteúdo de um registrador ou de um endereço for $FF, o que acontece ao incrementar este valor?

Para responder isto, nada melhor do que a prática:

```
LDY #$FF      A0 FF
INY           C8
BRK           00
```

Tente escrever esta rotina e verifique o conteúdo de Y após 0300G! Verá que o conteúdo de Y voltou a ser $00, e a cada incremento teremos novamente a seqüência de $00 até $FF.

EXERCÍCIOS:

1. *Assinale as afirmações corretas:*

   ( ) *Em linguagem de máquina, podemos incrementar apenas o acumulador.*

   ( ) *Incrementando o conteúdo de um endereço da página zero usando um registrador de índex, é obrigatório o uso do X.*

   ( ) *Ao incrementar um valor, e quando este atingir o valor $FF, qualquer incremento posterior não fará mais efeito.*

2. *Escreva uma rotina que coloque em A o conteúdo do endereço $EC00, em X o conteúdo do endereço $0013, e em Y o valor contido em A incrementado.*

3. *Incremente o conteúdo do acumulador sem o uso do X.*

capítulo 5

dec, dex, dey

No capítulo anterior, vimos as instruções que incrementam os conteúdos dos registradores X, Y ou de um endereço qualquer da memória. Agora, temos as instruções que decrementam os conteúdos.

Prevalece o mesmo critério como nas instruções INC, INX e INY, isto é, nós temos à nossa disposição as seguintes operações:

DEX – OP-Code: CA (diminui o valor de X por 1)

DEY – OP-Code: 88 (diminui o valor de Y por 1)

DEC página zero – OP-Code: C6

DEC página zero,X – OP-Code: D6

DEC absoluto – OP-Code: CE

DEC absoluto,X – OP-Code: DE

Exemplo: Decrementar o valor contido no endereço $0057, e o valor contido no endereço $5608.

Solução: O primeiro endereço faz parte da página zero, motivo pelo qual podemos optar pela instrução DEC página zero (ganhando assim memória e velocidade). Já, o segundo endereço exige um DEC absoluto.

```
DEC $57       C6 57
DEC $5608     CE 08 56
```

Será que funciona? Vamos ver primeiro os conteúdos originais dos endereços $0057 e $5608:

```
*0057
0057 - 00
*5608
5608 - 00
*
```

Temos portanto em ambos os endereços o valor $00. Escrevemos então a nossa rotina (não esqueça o byte $00 adicional!):

```
*0300: C6 57 CE 08 67 00
*
```

E rodando com 0300G, verificamos a seguir novamente os conteúdos dos endereços $0057 e $5608:

```
0307 - A=00  X=00  Y=00  P=B0  S=EE
*0057
0057 - FE
*5608
5608 - FE
*
```

Vamos rodar de novo?


```
*0300G
0307 - A=00  X=00  Y=00  P=B0  S=EE
*0057
0057 - FE
*5608
5608 - FE
```

Percebemos que quando um determinado valor já estiver com $00, o próximo decremento forma o valor $FF. Também para a instrução de decremento não temos condições de decrementar diretamente o valor contido no acumulador.

EXERCÍCIOS:

1. *Assinale as afirmações corretas:*

   ( ) *Decrementando um valor qualquer, este diminui seu valor de 1 até atingir o valor 0. Depois, um decremento adicional não fará mais efeito.*

   ( ) *Falando em registradores, podemos afirmar que somente os registradores de index podem ser incrementados e decrementados.*

   ( ) *A instrução INC página zero, Y é o inverso da instrução DEC página zero, Y.*

2. *Escreva uma rotina que leia o conteúdo do endereço apontado na página zero a partir de endereço $0089, e que transfira este valor encontrado para o registrador X. A seguir, este mesmo valor deve ser decrementado e colocado no acumulador.*

3. Carregue em X o conteúdo do endereço $01F3. A seguir, este valor deve ser incrementado três vezes.

4. Transfira o conteúdo do endereço $FE da página zero para A, e decremente o valor encontrado duas vezes.

5. Decremente o conteúdo de A uma vez.

capítulo 6

rts, brk, nop

RTS – OP-Code: 60

Até agora, usamos sempre o comando "G" após o endereço inicial para rodar uma rotina em linguagem de máquina, lembra-se? Vamos a um exemplo:

Digite a seguinte rotina no seu micro:

```
LDA $34      A5 34
TAX          A8
INY          C8
BRK          00
```

Vale relembrar que o BRK permite ler os conteúdos dos registradores, que por sua vez permite-nos verificar se o nosso programa realmente realizou a operação esperada.

Digitamos então:

```
*0300: A5 34 A8 C8 00
*0300G
0306 - A=05  X=05  Y=06  P=30  S=EA
```

Vamos agora retornar ao BASIC digitando "CONTROL" e a letra "C" ao mesmo tempo, seguido de um RETURN.

Sabemos que a nossa rotina inicia no endereço $0300, que corresponde ao endereço 768 em decimal. Através do comando CALL podemos rodar esta rotina partindo do BASIC, semelhante ao comando "G", a partir do monitor do Assembler.

Vamos lá!

```
] CALL 768
```

Ora, o que é isso na nossa tela? Apareceram novamente os conteúdos dos registradores, e estamos novamente no monitor Assembler!

Muitas vezes isto não é desejável, pois ao usar subrotinas em linguagem de máquina em programas em BASIC, deve haver uma continuidade do programa BASIC, que, provocado pela instrução BRK após a reentrada ao monitor Assembler, com certeza não se efetua.

A solução é substituir o comando BRK pela instrução RTS, que significa "RETURN", semelhante ao comando RETURN do BASIC. Agora sim, ao executar uma rotina em linguagem de máquina partindo do BASIC, este programa devolverá os controles ao BASIC após encontrar a instrução BRK.

Exemplo 1 (usando o BRK):

Escrever uma rotina que incremente o valor do endereço $4000 (16384 em decimal) duas vezes.

Solução:

```
INC $4000          EE 00 40
INC $4000          EE 00 40
BRK                00

*0300: EE 00 40 EE 00 40 00
*
```

Confira os mnemônicos usando 0300L!

Retorne ao BASIC (use CONTROL-C) e escreva agora o seguinte programa em BASIC para testar a nossa rotina:


```
10 POKE 16384,100
20 CALL 768
30 PRINT PEEK (16384)
40 END
```

Tudo certo? Ótimo, então vamos dar um RUN ao nosso programa...

Como era esperado, não temos na nossa tela o novo conteúdo do endereço 16384, mas sim algo assim:

```
0308 - A=03 X=9D Y=00 P=34 S=F2
*
```

Altere agora a nossa rotina de tal modo que a instrução BRK seja substituída pela instrução RTS:

Exemplo 2 (usando o RTS) — Solução

```
INC $4000          EE 00 40
INC $4000          EE 00 40
RTS                60

*0300: EE 00 40 EE 00 40 60
*
```

Use 0300 L para conferir os mnemônicos!

Retorne ao BASIC e digite novamente RUN.

Desta vez, aparece o valor 102 na tela (colocamos inicialmente o valor 100 na posição 16384, e após o CALL 768, que incrementa qualquer valor neste endereço duas vezes, o nosso valor inicial foi alterado para 102).

Daqui para frente só usaremos o BRK se realmente houver a necessidade de consultar os conteúdos dos registradores. Para todos os demais casos, a instrução correta para "terminar" uma rotina em linguagem de máquina é o RTS.

BRK — Op-Code 00

Creio que a esta altura já conhecemos bem esta instrução, portanto não há necessidade de entrar novamente em detalhes.

NOP — Op-Code EA

Eis uma instrução muito especial. É a única instrução do 6502

que não faz absolutamente *nada*, a não ser ocupar espaço na memória!

Tente você mesmo, carregando por exemplo todos os registradores com valores quaisquer, e intercalando na sua rotina alguns NOP's. Verá que nada mudou após rodar o nosso programa. Qual então é a razão da existência do NOP no quadro das instruções da nossa CPU?

Muitas vezes temos a necessidade de escrever uma rotina em linguagem de máquina muito extensa, e nem sempre temos todos os passos na palma da mão, da maneira como o programa será executado. Por exemplo, numa determinada altura do nosso programa, estamos em dúvida se o registrador de índex Y deve ser incrementado ou não. Se nós optarmos por um "não", e mais tarde percebermos que cometemos um erro, será muito trabalhoso "encaixar" a instrução INY no meio da nossa rotina.

Neste caso, no lugar do INY previsto, colocamos a instrução NOP, que "reserva" assim uma posição na memória para eventual INY. Confirmando a necessidade do INY (isto no nosso exemplo), então podemos simplesmente substituir o NOP pela instrução INY, sem a necessidade de deslocar todas as demais instruções.

Dentro deste critério, se queremos "reservar" três posições na memória para um eventual comando adicional (Ex.: INC $AA23), colocamos simplesmente três NOP's no programa.

EXERCÍCIOS:

1. *Assinale as afirmações corretas:*
   - *( ) Para listar os conteúdo dos registradores, a rotina em linguagem de máquina deve terminar com a instrução BRK.*
   - *( ) Para o uso de rotinas Assembler no BASIC, podemos usar somente o RTS.*
   - *( ) Chamando rotinas com CALL que terminam com a instrução BRK, existe a desvantagem do programa desviar-se para o monitor Assembler.*
   - *( ) A instrução NOP pode modificar o conteúdo do registrador de índex Y.*
2. *Escreva uma rotina que carregue primeiramente o acumulador com o valor $30, e a seguir o transfira para X. Entre estas duas operações deve existir um NOP. Inicie a rotina no endereço 0300.*
3. *Introduza na rotina do exercício 2 a instrução TAY, sem alterar a posição da instrução TAX.*

capítulo 7

sta, stx, sty

Até agora já sabemos como copiar em A, X ou Y o conteúdo de um endereço qualquer da memória. Lembre-se que utilizamos para isto as instruções LDA, LDX e LDY. Pois bem, chegou a hora de fazer exatamente o inverso, ou seja, copiar o conteúdo de um registrador na memória.

Usando o acumulador A, temos as seguintes variantes para este fim:

STA página zero	—	OP-Code:	85
STA página zero,X	—	OP-Code:	95
STA absoluto	—	OP-Code:	8D
STA absoluto,X	—	OP-Code:	9D
STA absoluto,Y	—	OP-Code:	99
STA (indireto,X)	—	OP-Code:	81
STA (indireto),Y	—	Op-Code:	91

Vamos ver estas instruções em detalhes:

Ex.: Vamos supor que queremos colocar o valor 7E no endereço F0 da página zero, usando para isto o acumulador. A rotina seria da seguinte maneira:

```
LDA #7E        A9 7E
STA F0         85 F0
RTS            60
```

E usando o monitor:

```
*0300: A9 7E 85 F0 60
*
```

Confira os mnemônicos com 0300L!

Tudo pronto? Ótimo, então vamos verificar primeiro o conteúdo do endereço F0 da página zero, *antes* de rodar a nossa rotina:

```
*00FF
00FF - FF
*
```

Temos o valor FF (ou talvez um outro valor) na posição 00F0, certo? Vamos então rodar a nossa pequena rotina:

```
*0300G
*
```

Percebeu que agora não foram listados os registradores? Isto ocorre porque terminamos o nosso programa com a instrução RET, e não com o BRK como nos programas anteriores. Será que alterou o conteúdo do endereço 00F0? É só conferir:

```
*00F0
00F0 - 7E
*
```

De fato! No endereço 00F0 temos agora realmente o valor 7E, conforme previsto. O que aconteceu dentro do computador é indicado na figura 9.1.

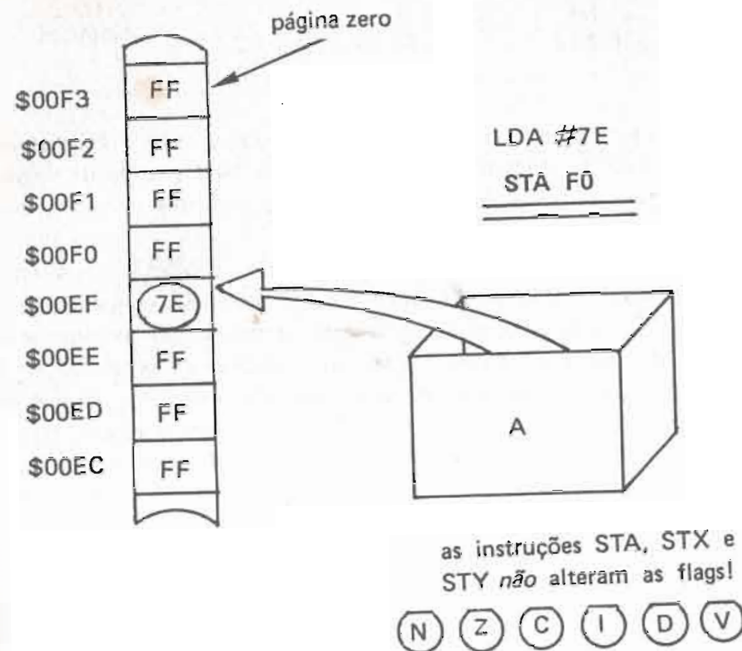


Figura 9.1 — Transferência de um valor contido no acumulador para a página zero da memória.

Creio que a esta altura você já pode imaginar o que irá acontecer na rotina a seguir:

```
LDX #09        A2 09
LDA #CF        A9 CF
STA 80,X       95 80
RTS            60
```

Se você chegar à conclusão que o valor CF será colocado no endereço 0089 da página zero, então está de parabéns. Isto é um sinal que o mundo nebuloso da linguagem de máquina já começa a clarear. Se não chegou a este resultado, não faz mal. Releia o início deste capítulo...

No primeiro exemplo deste capítulo colocamos o valor 7E no endereço 00F0 de página zero, usando a instrução STA página zero. O mesmo efeito pode ser criado com a instrução STA absoluto, conforme segue:

```
LDA #7E          A9 7E
STA 00F0         8D F0 00
RTS              60
```

Tente digitar esta rotina. Comparando-a com o primeiro exemplo, podemos perceber que agora estamos necessitando de um byte a mais para obter o mesmo resultado (no primeiro exemplo, a rotina ocupa 5 bytes, e agora 6).

Já falamos nisto antes. As instruções absolutas permitem acessar qualquer endereço de toda a nossa memória, enquanto as instruções relacionadas com a página zero somente cobrem uma área restrita da memória. Assim sendo, por questões de economia de bytes (e com isto um aumento da velocidade de execução), usamos sempre quando for possível instruções relacionadas com a página zero.

Desta maneira, para colocar por exemplo o valor 1B no endereço 00A1, utilizamos a instrução STA página zero, e para colocar o valor 1B no endereço 17D5, necessariamente o STA absoluto.

Isto também é válido para as instruções indexadas, isto é, usando os registradores X ou Y para completar o endereço final.

Dispenso aqui exemplos para o STA absoluto,X e o STA absoluto,Y, pois prevalece também aqui o mesmo critério como nas instruções LDA, LDX e LDY.

Todavia, nunca é demais rever as instruções indiretas, no nosso caso, o STA (indireto,X) e o STA (indireto),Y. Não seria a primeira vez que uma pequena confusão do funcionamento de cada uma destas duas instruções resulta em rotinas erradas, e muitas vezes desastrosas.

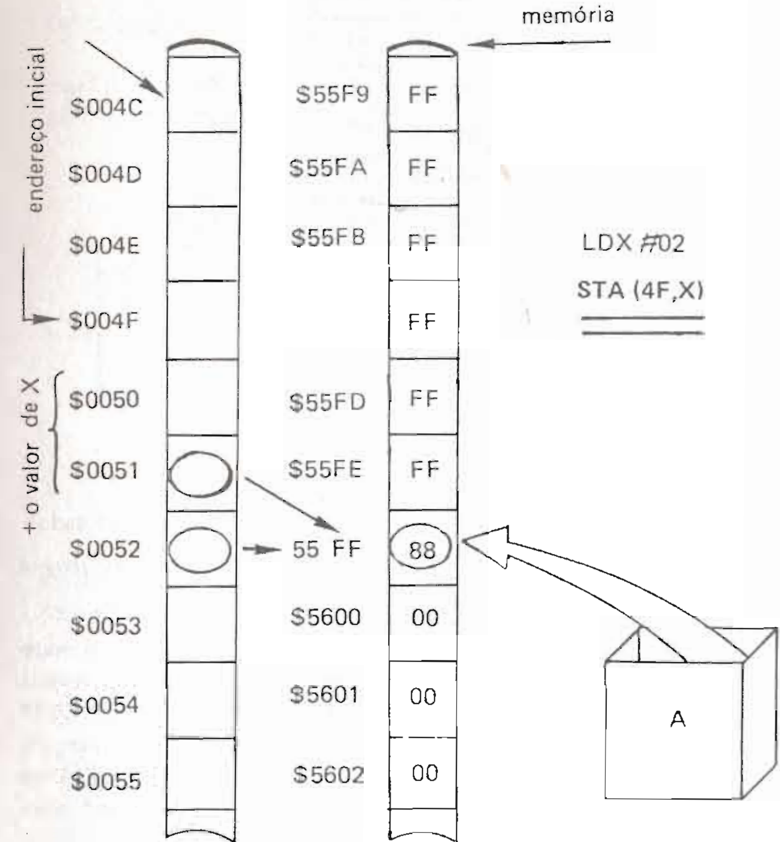
Vamos a um exemplo:

```
LDX 02           A2 02
LDY 02           A0 02
LDA 88           A9 88
```

nesta altura, o acumulador contém o valor 88, e tanto o registrador de index X como o Y contêm o valor 02, confere? Agora tente observar atentamente nas figuras a seguir, qual é a grande diferença das seguintes instruções:

Instrução 1: STA (4F,X) — vide figura 9.2

Instrução 2: STA (4F),Y — vide figura 9.3



nenhum flag é afetado!

N Z C I D V

Figura 9.2 — Transferência de um valor contido no acumulador para a memória, no modo indireto.

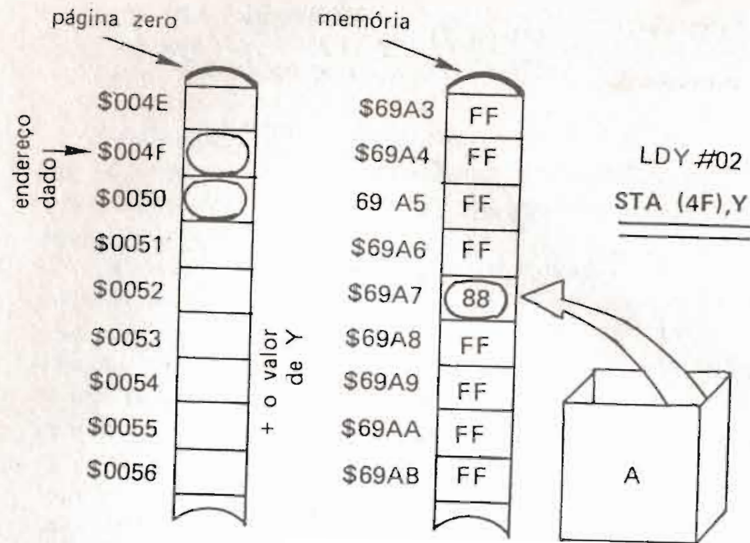


Figura 9.3 — Transferência de um valor contido no acumulador para a memória, no modo indireto.

Percebeu a diferença? Exato, a instrução 1 lê primeiro o endereço *contido* no endereço formado pelo valor 4F e o valor de X (resultando assim no endereço 0051). Lá é encontrado o endereço 55FF (lembre-se que primeiramente temos o byte menos significativo, no caso o valor 55, e a seguir o byte mais significativo na memória). Uma vez encontrado este endereço 55FF, o valor de A (= 88) será colocado exatamente neste endereço!

Confira você mesmo, digitando esta pequena rotina a partir do endereço 0300:

```
*0300: A2 02 A0 02 A9 88 81 4F 60
*0300G
*55FF
55FF - 88
*
```


Temos, então, o valor 88 no endereço 55FF, como podemos ver.

E na instrução 2? Aqui é lido primeiro o endereço contido a partir do endereço 4F da página zero, e somado a este endereço o valor de Y. Este novo endereço resultante será o endereço final para receber o valor de A, no caso, o valor 88!

Tudo claro? Então já podemos ver as instruções para colocar na memória os conteúdos dos registradores X e Y. Para estes, temos apenas algumas das instruções existentes para o acumulador:

STX página zero	—	OP-Code: 86
STX página zero,Y	—	OP-Code: 96
STX absoluto	—	OP-Code: 8E

e, respectivamente para o registrador Y, temos:

STY página zero	—	OP-Code: 84
STY página zero,X	—	OP-Code: 94
STY absoluto	—	OP-Code: 8C

Observe que quando há a necessidade de usar instruções indiretas, necessariamente teremos de usar o acumulador, pois não existem tais instruções para os registradores de índex X e Y.

EXERCÍCIOS:

1. *Assinale as afirmações corretas:*

   ( ) As instruções STA, STX e STY servem unicamente para guardar os conteúdos de A, X ou Y na memória.

   ( ) A instrução STA absoluto pode ser substituída pela instrução STA página zero.

   ( ) A instrução STA página zero pode ser substituída pela instrução STA absoluto

   ( ) Usando apenas o X ou o Y não podemos realizar STX's ou STY's relativos

Trabalhos:

2. *Escreva uma rotina que transfira o conteúdo do endereço $0030 para o endereço $1F00.*

3. *Carregue o registrador de índex Y com o valor BC, coloque o endereço $4500 na página zero a partir do endereço $0055, e realize a seguir um STA (55),Y do valor FF. Em que endereço acharemos o valor 11?*

capítulo 8

adc, sbc

Além das poderosas instruções de transferir valores para endereços, e de endereços para registradores, a CPU 6502 ainda oferece a possibilidade de realizar cálculos com estes valores.

Esta faculdade é o tema deste capítulo, onde veremos as adições e as subtrações semelhantes às operações já conhecidas no BASIC (+ e −).

ADC imediato	–	OP-Code: 69
ADC página zero	–	OP-Code: 65
ADC página zero,X	–	OP-Code: 75
ADC absoluto	–	OP-Code: 6D
ADC absoluto,X	–	OP-Code: 7D
ADC absoluto,Y	–	OP-Code: 79
ADC (indireto,X)		OP-Code: 61
ADC (indireto),Y		OP-Code 71

Temos na relação acima todos os tipos de ADC's possíveis. Ob-

serve que para todos os casos é necessário *sempre* o acumulador. Não existem operações semelhantes com os registradores de índex X ou Y.

A instrução ADC imediato soma ao conteúdo de A o valor definido. Assim, a seqüência :

```
LDA #$00
ADC #$20
```

resulta em 20. Após um novo ADC 20, o conteúdo de A será 40, e assim por diante...

Na prática, acontece algo curioso ao continuar as somas, o que veremos melhor num exemplo:

Ex.: somar ao acumu ador o valor 60 quatro vezes.

Rotina 1:

```
LDA #$00        A9 00
RTS             60
```

A rotina 1 é necessária para definir o valor inicial de A, no nosso caso, com o valor 0. Digitamos portanto:

```
*0300: A9 00 60
```

,seguido de:

```
*0300G
*
```

Agora, a rotina 2, que soma o valor 60 ao acumulador:

Rotina 2:

```
ADC #$60        69 60
BRK             00
```

Colocamos a instrução BRK para permitir a leitura de A após cada soma. Esta rotina também será colocada a partir do endereço $0300, já que não precisamos mais da rotina 1:

```
*0300: 69 60 00
*
```

Vamos fazer as somas?


```
*0300G
0304 - A=60  X=00  Y=00  P=30  S=EC
*
```

Tudo certo? Como o esperado, o acumulador assumiu o valor 60 (valor original + 60). Vamos à segunda soma:

```
*0300G
0304 - A=C0  X=00  Y=00  P=F0  S=EA
*
```

Agora o valor de A é C0, e podemos fazer a prova convertendo os valores hexadecimais 60 e C0 para decimal:

```
                 60 hexadecimal = 96 em decimal
        (somando mais 60) + 96 em decimal
                         --------------
                         192 em decimal (=C0 em hexadecimal))
```

Agora a terceira soma:

```
*0300G
0304 - A=20  X=00  Y=00  P=31  S=E8
*
```

O resultado correto seria 0120 em hexadecimal. Como o acumulador pode manter apenas 8 bits, ou seja, o equivalente a duas casas hexadecimais, o resultado em A só indica 20.

Até aqui nenhuma novidade em especial. Mas como sabemos que ultrapassamos o valor máximo possível do acumulador? A resposta dará a próxima soma, que considerando o valor inicial de 20, resultaria em 80, certo?

```
*0300G
0304 - A=81  X=00  Y=00  P=F0  S=E6
*
```

Para nosso espanto, encontramos o valor 81 em A, e não o valor 80 como esperávamos! O que aconteceu com a nossa soma simples?

Eis a grande curiosidade da qual falei há pouco. Cada vez que numa operação anterior for ultrapassada a capacidade máxima de um registrador, o *Carry-Flag* assume o nível lógico 1. (Falaremos daqui a pouco deste flag!).

E é exatamente o Carry-Flag que a operação ADC inclui na soma:

```
ADC #60
```

= **valor inicial de A + 60 + valor do Carry-Flag**

Para dar mais clareza, vamos analisar o Carry-Flag em si:

Cada registrador tem a capacidade de manter valores binários de até 8 bits (vide apêndice A). Assim, para representar por exemplo o valor hexadecimal C9, temos a seguinte configuração binária:

```
C9 = 1 1 0 0 1 0 0 1
```

O maior valor possível dentro de um registrador tem todos os bits com o nível lógico 1:

```
1 1 1 1 1 1 1 1 = FF
em hexadecimal (=255 em decimal)
```

Somando agora o valor 1 a este valor máximo, teremos como resultado:

```
1 0 0 0 0 0 0 0 0 em binário
          0 1 0 0 em hexadecimal
            2 5 6 em decimal
```

Percebemos que foi necessário um *nono* bit para expressar o novo resultado. Como o registrador tem apenas 8 bits, o nono bit passa a ser chamado de *Carry-Flag,* onde podemos reconhecer através do seu nível lógico se a capacidade de um registrador foi ultrapassada ou não, ou seja:

Carry-Flag	0	estamos dentro do limite máximo
Carry-Flag	1	ultrapassamos o limite máximo

Por isso, após a penúltima soma no exemplo anterior, onde somamos o valor hexadecimal 60 ao conteúdo de A (que era C0 em hexadecimal), criamos um "nono" bit, ou seja, um nível "1" do Carry-Flag. Veja o que aconteceu com os bits nesta soma:

```
Valor anterior em A = C0 hex =   1 1 0 0 0 0 0 0
                    + 60 hex = + 0 1 1 0 0 0 0 0
                                 _______________

resultado intermediário = 1 0 0 1 0 0 0 0 0 (=0120 hex)
somando o Carry anterior =                0
                           _________________
Resultado final            1 0 0 1 0 0 0 0 0 (=0120 hex)
```

Podemos ver na conta acima que o resultado na forma binária necessitou de um nono bit, que no caso tem o valor "1". A soma do Carry anterior não alterou a soma total, pois ainda não havíamos ultrapassado o valor máximo do acumulador.

A esta altura creio que pelo menos já sabemos que o valor em A agora é 20 em hexadecimal, com a diferença que o Carry-Flag tem agora o valor "1". Todavia, somente na próxima operação iremos descobrir este fato:

```
Valor anterior em A = 20 hex =   0 0 1 0 0 0 0 0
                    + 60 hex = + 0 1 1 0 0 0 0 0
                                 _______________

resultado intermediário =  1 0 0 0 0 0 0 0  (=80 hex)
somando o Carry anterior =               1
                           _______________
Resultado final            1 0 0 0 0 0 0 1  (=81 hex)
```

Após esta soma, foi preciso novamente um nono bit para representar o resultado final? Não. Portanto o Carry-Flag vale agora "0", e continuando a soma com o valor 60, teremos como resultado E1. Experimente!

O mesmo critério referente ao Carry-Flag é válido para as demais operações com ADC, isto é, cada vez que a soma ultrapassar o limite máximo de A, será criado um Carry-Flag 1, cujo valor será considerado na operação *a seguir.*

ADC página zero	– Ex.:	`ADC 30`	– soma ao acumulador o valor encontrado no endereço 0030.
ADC página zero,X	– Ex.:	`LDX #$03` `ADC 30,X`	– soma ao acumulador o valor encontrado no endereço 0033 (0030 + valor de X).
ADC absoluto	– Ex.:	`LDA 0445`	– soma ao acumulador o valor encontrado no endereço 0445.
ADC absoluto,X	– Ex.:	`LDX #$05` `ADC 0445,X`	– soma ao acumulador o valor encontrado no endereço 044A (0445 + valor de X).
ADC absoluto,Y	– Ex.:	`LDY #$04` `ADC 0455,Y`	– soma ao acumulador o valor encontrado no endereço 0449 (0445 + valor de Y).
ADC (indireto,X)	– Ex.:	`LDX #$03` `ADC(55,X)`	– primeiro é encontrado o endereço contido a partir do endereço 0058 (0055 + valor de X). A seguir, o valor encontrado neste endereço será somado ao acumulador.
ADC (indireto),Y	– Ex.:	`LDY #$04` `ADC(55),Y`	– primeiro localiza o endereço contido a partir do endereço 0055. A este endereço será somado o valor de Y, e a seguir somado o valor encontrado neste novo endereço ao acumulador.

Agora já temos uma idéia da importância do Carry-Flag em operações de adição. Igualmente para opereções de subtração, o Carry-Flag ocupa também um papel importante. Antes disto, vamos dar uma breve olhada nas diversas operações possíveis envolvendo subtrações:

SBC imediato	–	OP-Code: E9
SBC página zero	–	OP-Code: E5
SBC página zero,X	–	OP-Code: F5
SBC absoluto	–	OP-Code: ED
SBC absoluto,X	–	OP-Code: FD
SBC absoluto,Y	–	OP-Code: F9
SBC (indireto,X)	–	OP-Code: E1
SBC (indireto),Y	–	OP-Code: F1

Comparando o quadro acima com o quadro do ADC, percebemos que temos em princípio as mesmas possibilidades com SBC como tivemos com o ADC.

Há pouco foi dito que o Carry-Flag influi no resultado final. Vamos a um exemplo:

Digite o seguinte programa no seu micro:

```
0300 LDA #$10    A9 10
     SBC #$00    E9 00
     BRK         00

*0300: A9 10 E9 00 00
```

Antes de rodar o nosso programa, vamos analisar o que realmente estamos querendo fazer. Inicialmente, carregamos no acumulador o valor hexadecimal 10, e a seguir subtraímos o valor 0. O resultado deveria ser 10, certo? Vamos ver:

```
*0300G
0306 - A=0F  X=00  Y=00  P=31  S=F0
*
```

Por favor, não desligue seu computador agora e ponha-o no canto! Ao contrário do que parece, ele sabe que 10h, 00h resulta em 10h, e não em 0Fh, como surge no seu monitor. O motivo para esta confusão aparente é nada mais do que o nosso Carry-Flag. Nesta altura quero dar um alerta aos leitores que já conhecem outros microprocessadores:

> Em operações envolvendo subtrações, o Carry-Flag é criado quando o resultado *não* se torna negativo, e permanece com nível 0 quando o resultado é negativo! Isto é uma característica *contrária* à maioria dos outros microprocessadores!

O resultado começa a fazer sentido agora? Ainda não? Então vamos expressar o comportamento do 6502 através de uma fórmula:

$$\text{Resultado} = \text{Acumulador} - \text{valor} - \overline{\text{Carry-Flag}}$$

A linha acima da palavra Carry-Flag indica que será considerado o valor invertido (contrário) do Carry. Assim sendo, quando o Carry-Flag valer 0, este $\overline{\text{Carry-Flag}}$ vale 1, e vice-versa.

Desta maneira, para chegar a um resultado correto, o primeiro passo antes da subtração em si é levar o Carry-Flag para o valor 1 através de uma instrução apropriada SEC (OP-Code: 38). No capítulo a seguir trataremos desta instrução em detalhes.

Usaremos então a instrução SEC no nosso primeiro exemplo:

```
0300 SEC        38
     LDA #$10   A9 10
     SBC #$00   E9 00
     BRK        00

*0300: 38 A9 10 E9 00 00
*0300G
0307 - A=10  X=00  Y=00  P=31  S=EE
```

Agora deu certo! Resumindo, podemos definir as seguintes "regras" ao usar ADC e SBC:

ADC – antes de qualquer operação, o Carry-Flag deverá ser *resetado*, isto é, *deve valer 0*. Isto pode ser obtido usando a instrução CLC – OP-Code: 18. (Vide próximo capítulo).

SBC – antes de qualquer operação, o Carry-Flag deverá ser *setado*, isto é, *deve valer 1*. Isto pode ser obtido usando a instrução SEC – OP-Code: 38. (Vide próximo capítulo).

EXERCÍCIOS:

1. *Assinale as afirmações corretas:*

   ( ) *As operações de adição e de subtração somente podem ser efetuadas com o acumulador.*

   ( ) *Somando o valor 1 ao acumulador, cujo conteúdo já é FF, o resultado em A permanece FF, já que este é o maior valor possível que A pode manter.*

   ( ) *A cada adição, o valor do Carry-Flag será adicionado ao resultado.*

   ( ) *O valor do Carry-Flag depende do resultado da operação anterior.*

2. *Resolva:*

   a) escrever uma rotina que some ao acumulador o conteúdo do endereço $FE44.

   b) Subtraia de A o valor contido no endereço $23 da página zero

   c) o acumulador contém o valor 7E, e o registrador X o valor 1F. Subtraia X de A.

3. *Subtraia o conteúdo do endereço $001F do conteúdo do endereço $FF3A. O resultado deve ficar no registrador X.*

4. *Subtraia X de Y. O resultado deve ser colocado no endereço $00FA.*

5. *Some X e Y. O resultado deve estar em A. Use o endereço $00FD como ajuda*

capítulo 9

clc, cld, clv, sec, sed

Neste capítulo iremos conhecer as instruções que manipulam diretamente os bits do registrador de Status, ou seja, os Flags. Tratávamos estes flags a partir da página *STATUS A*, lembra-se? Até agora, já tivemos contato com o Carry-Flag, um dos mais usados flags do sistema. Todavia, existem outros flags, não menos importantes, como por exemplo o Zero-Flag. E para completar a nossa coleção, temos ainda o Decimal-Flag, o Overflow-Flag, o Negative-Flag, o Break-Flag e o Interrupt-Flag.

Podemos retesar ou setar alguns destes flags através de uma simples instrução:

Para retesar o

Carry-Flag	CLC	(OP-Code: 18)
Decimal-Flag	CLD	(OP-Code: D8)
Overflow-Flag.	CLV	(OP-Code: B8)

Para garantir o flag com nível lógico 1 (verdadeiro), temos que setar:

Carry-Flag . . . . SEC (OP-Code: 38)
Decimal-Flag . . . . SED (OP-Code: F8)
Overflow-Flag. . . . . não existe instrução para tal fim.

Com exceção do Interrupt-Flag, que trataremos mais tarde por ser um flag todo especial dentro do nosso micro, não podemos afetar os demais flags através de instruções específicas como é o caso do Carry-, Decimal- e do Overflow-Flag.

Se alguém perguntar agora para que servem todos estes flags, você saberia responder? Pelo menos no que se refere ao Carry-Flag, provavelmente já deve ter uma idéia. E quanto aos demais? Será que estes flags também são considerados nas operações da CPU?

Posso tranqüilizá-lo; a resposta é não. Basta o cuidado com o Carry-Flag em operações de adição e de subtração. Então, para que servem os demais flags?

Esta pergunta será respondida no próximo capítulo, mas posso adiantar que a falta dos flags em linguagem de máquina é a mesma coisa que a falta do comando IF do BASIC...

Aliás, nas instruções que manipulam os flags, temos as letras CL e a seguir a letra do Flag para as instruções que resetam este flag, e as letras SE juntas com a letra do flag nas instruções que setam este flag. Deste modo, sabendo que CL é a abreviação do CLEAR, e SE a abreviação do SET, será difícil trocar estas instruções acidentalmente.

EXERCÍCIOS:

1. *Assinale as afirmações corretas:*

   ( ) O Decimal-Flag pode ser resetado através de uma instrução específica para este fim.

   ( ) a instrução correta para setar o Zero-Flag é SEZ.

   ( ) desconsiderando o Interrupt-Flag, temos apenas três flags que podem ser manipulados diretamente.

   ( ) com exceção do Carry-Flag, os demais flags não são absolutamente necessários para programas em linguagem de máquina.

capítulo 10

bcc, bcs, beq, bmi, bne, bpl, bvc, bvs

Como já foi dito no capítulo anterior, veremos a partir de agora a utilidade dos flags dentro dos nossos programas em linguagem de máquina. Também foi dito que os flags são tão importantes como a instrução IF do BASIC.

Qualquer programa que não tenha desvios condicionais torna-se bastante inútil. Teríamos apenas uma seqüência de instruções que seriam executadas sempre igualmente quando rodássemos o programa. E é justamente o que podemos fazer em linguagem de máquina: desvios condicionais de acordo com o valor de determinados flags.

Você lembra-se da adição, onde foi criado um Carry-Flag "1" cada vez que ultrapassássemos a capacidade do acumulador? Não seria interessante testar após cada soma o Carry-Flag, e sinalizar de alguma maneira que houve um resultado superior a 255? Para isto, podemos usar uma das seguintes instruções de desvios condicionais (tente descobrir a instrução adequada para o nosso exemplo da adição):

BCC – OP-Code: 90 = desvia quando Carry-Flag for 0
BCS – OP-Code: B0 = desvia quando Carry-Flag for 1
BEQ – OP-Code: F0 = desvia quando Zero-Flag for 1

BMI – OP-Code: 30 = desvia quando Negative-Flag for 1
BNE – OP-Code: D0 = desvia quando Zero-Flag for 0
BPL – OP-Code: 10 = desvia quando Negative-Flag for 0
BVC – OP-Code: 50 = desvia quando Overflow-Flag for 0
BVS – OP-Code: 70 = desvia quando Overflow-Flag for 1

Não se preocupe com os Flags, pois ainda não tratamos deles. Faremos muitos exercícios para demonstrar bem o funcionamento das instruções acima.

Quando falamos em desvios, temos também que indicar de alguma maneira onde deverá ser feito este desvio. Por isto, cada instrução de desvio necessita um *complemento* que indica a *distância* do desvio! Assim, se nós quisermos que seja realizado um desvio de 10 posições na memória para frente, quando o Zero-Flag estiver com nível 0, teríamos que programar BNE 0A (lembre-se que programamos com valores hexadecimais). Só podemos programar este valor quando sabemos exatamente a quantidade de bytes das instruções que devem ser pulados, exigindo, portanto, um pouco mais de cuidado. Existem programas que permitem programar desvios condicionais indicando o novo endereço de destino, mas isto é uma particularidade destes programas Assembler, pois não existem no quadro de instruções da CPU 6502 desvios condicionais absolutos! No caso destes programas Assembler, durante a assemblagem, o endereço dado será substituído pelo valor que representa a distância para este endereço.

Para ilustrar melhor, vamos a um exemplo. Vamos supor que queremos somar duas vezes o conteúdo do endereço $0350 no acumulador. Se o resultado ultrapassar 255, então queremos que o registrador X assuma o valor 1. Caso contrário, o valor de X deverá ser 0.

Em *BASIC*, uma solução seria esta:

```
10 LET A= PEEK(848)
20 LET A= A+PEEK (848)
30 IF A > 255 THEN GOTO 60
40 LET X=0
50 END
60 LET X=1
70 END
```


No BASIC, o nosso desvio condicional vai para a linha 60 assim que o resultado ultrapassa o valor 255. Como em linguagem de máquina não temos números de linha, e tão pouco podemos usar endereços absolutos, teríamos que contar a quantidade de bytes equivalente às linhas 40 e 50 da rotina acima. Vamos então à linguagem de máquina:

Em primeiro lugar, vamos escrever as instruções, por exemplo, a partir do endereço $0300:

```
0300 LDA $0350     AD 50 03
0303 ADC $0350     6D 50 03
0306 BCS n         B0 ...
```

Nesta altura, não sabemos ainda a distância do nosso desvio condicional, motivo pelo qual deixamos este valor por enquanto em aberto.

```
0308 LDX #00       A2 00
030A BRK           00
030B LDX #01       A2 01
030D BRK           00
```

Agora, já sabemos que o endereço para onde o nosso desvio deverá ir é o $030B. Resta contar todos os bytes entre os desvios e este endereço de destino, cuja conta dá 3, certo? Então a distância correta é 03, que deverá ser colocado imediatamente após a instrução BCS. Assim sendo, a rotina completa fica assim:

```
0300 LDA $0350     AD 50 03
     ADC $0350     6D 50 03
     BCS $030B     B0 03 \
     LDX #00       A2 00  )  "pula" 3 endere-
     BRK           00    /   ços se Carry = 1
030B LDX #01       A2 01 <
     BRK           00
```

Observe que indicamos na rotina pronta o endereço $030B, pois assim podemos analisar mais tarde as nossas rotinas mais facilmente. Para fazer este programa, digitamos:

```
*0300: AD 50 03 6D 50 03 B0 03 A2 00 A2 01 00
```

e conferimos os mnemônicos com 0300L. Se tudo estiver certo, temos ainda que definir um valor qualquer no endereço $0350, pois queremos operar com o conteúdo deste endereço. Vamos inicialmente definir o valor 40:

```
*0350: 50
```

e, a seguir, rodar a nossa rotina:

```
*0300G
030C - A=80  X=00  Y=00  P=72  S=E4
*
```

Até aqui, tudo certo. $40 + $40 resulta em $80 (conteúdo de A), e como o resultado não ultrapassou 255, o valor de X é 0. Agora, vamos ver o que acontece com valores maiores, por exemplo A0:

```
*0350: A0
*0300G
030C - A=40  X=01  Y=00  P=72  S=E2
*
```

De fato, X assumiu o valor 1, conforme previsto. O conteúdo de A representa o byte menos significativo do resultado, que seria $0140. Assim temos um Carry-Flag = 1, e o programa foi desviado para o endereço 030B.

Tente jogar outros valores no endereço $0350 e confira os resultados.

Fácil, não? Mas até que distância posso desviar o programa? Se você acha que a distância máxima é 255 (no caso com o valor $FF), então infelizmente está enganado. Podemos utilizar o desvio condicional também para *trás*, e não somente para frente. Por isto, temos dois critérios para definir a direção do desvio:

a) desvios para frente: aqui podemos "pular" até 127 posições, tendo, portanto, os seguintes valores à nossa disposição:
$00 (+0) até $7F (+127)

Assim, a instrução BNE $30 significa que há um desvio de 48 posições para frente quando o Zero-flag estiver em 0.

b) desvios para trás: nesta variante, podemos "pular" entre −1 e −128 posições, representadas pelos seguintes valores:

$80 (−128) até $FF (−1)

Observe que a partir do valor $80 o desvio é efetuado para trás!

Desvio condicional para trás:

Esta opção merece um pouco de atenção. Em primeiro lugar, precisamos saber a partir de que endereço devemos contar os bytes a serem "pulados":

A posição −1 ($FF) é o próprio endereço onde se encontra a distância, e a posição −2 ($FE), o início do desvio condicional:

Ex.: BNE $FE   D0 FE
desvia −2, quando Zero-flag = 0

Também aqui vamos ver isto através de um exemplo:

Agora queremos somar o conteúdo do endereço $0350 tantas vezes, até que o resultado ultrapasse 255. Quando isto ocorrer, o programa deve parar.

Em BASIC, poderíamos escrever:

```
10 LET A = PEEK (848)
20 LET A = A + PEEK (848)
30 IF A < 255 THEN GOTO 20
40 END
```

Percebemos que o desvio condicional somente deverá ser efetuado quando o resultado for menor que 255, ou seja, enquanto o Carry-Flag for 0. Por isso, usaremos agora a instrução BCC:

```
0300 LDA $0350      AD 50 03
0303 ADC $0350      6D 50 03
0306 BCC $0303      90 FB
```

Quando o Carry-Flag estiver com nível 0, queremos que o programa volte para o endereço $0303, onde haverá uma nova soma. Considerando o início da própria instrução do desvio com –2, temos como distância final –5 para atingir o *início* da instrução desejada. Na tabela do final deste capítulo, encontramos para a distância –5 o valor $FB. Resta ainda a instrução final:

```
0308 BRK            00
```

Programamos então:

```
*0300: AD 50 03 6D 50 03 90 FB 00
*300L                          (para conferir os mnemônicos)
```

Certifique-se que antes de rodar este programa, o Carry-Flag esteja com nível 0, pois caso contrário, o valor do registrador A pode estar errado, pois somará à primeira soma o valor do Carry da operação anterior. Em caso de dúvida, desligue o computador e ligue-o novamente.

Vamos colocar no endereço $0350 agora o valor $38:

```
*0350: 38
```

e rodar a nossa rotina:

```
*0300G
030A - A=18  X=00  Y=00  P=31  S=EF
*
```

Através do conteúdo de A podemos ver que realmente o valor $38 foi somado vários vezes, até que o resultado ultrapassou o valor $FF:

1ª soma	$38 + $38 = $70	(Carry = 0, portanto nova soma)
2ª soma	$70 + $38 = $A8	(Carry = 0, portanto nova soma)
3ª soma	$A8 + $38 = $E0	(Carry = 0, portanto nova soma)
4ª soma	$E0 + $38 = $0118	(Carry = 1, portanto fim da rotina)

Tente com outros valores. Observe que daqui para frente, o resultado final sempre será 1 a mais do que deveria ser, pois cada vez será considerado na primeira soma o Carry criado na rotina anterior. Ao colocar no endereço $0350 o valor 0, o nosso programa nunca chegará ao fim. Neste caso, terá de resetar o microcomputador para parar o programa.

Da mesma forma, podemos desviar o programa quando o resultado de uma operação é zero ou não, usando as instruções BEQ e BNE.

Ex.: Se o conteúdo do endereço $0350 mais o conteúdo do endereço $0351 resultar em zero, então o registrador X deverá ser carregado com o valor FF. Caso contrário, o valor de X deverá ser 88:

```
0300 LDA $0350      AD 50 03
0303 ADC $0351      6D 51 03
0306 BCS $030B      B0 03
0308 LDX #$88       A2 88
030A BRK            00
030B LDX #$FF       A2 FF
030D BRK            00
```

Tente rodar jogando diversos valores em $0350 e em $0351. Observe que o Carry da rotina anterior influencia no resultado!

Dentro deste critério, tente desenvolver alguma rotina que coloque em Y o valor 7F quando o resultado de uma subtração for negativo (use BMI ou BPL). Atenção agora, pois o critério para o Negativo-Flag é o seguinte:

1. quando o resultado for entre $00 e $7F, o resultado será considerado como *positivo*. Portanto, o Negativo-Flag será 0!

2. quando o resultado for entre $80 e $FF, o resultado será considerado como *negativo*. Portanto, o Negativo-Flag será 1!

Não esqueça também de iniciar toda rotina que envolve subtrações com a instrução SEC (lembra?).

O Overflow-Flag será tratado mais adiante, quando falarmos da instrução BIT.

Agora, a tabela para obter os valores hexadecimais corretos para desvios condicionais:

Desvio para frente

	0	1	2	3	4	5	6	7	8	9
0	00	01	02	03	04	05	06	07	08	09
10	0A	0B	0C	0D	0E	0F	10	11	12	13
20	14	15	16	17	18	19	1A	1B	1C	1D
30	1E	1F	20	21	22	23	24	25	26	27
40	28	29	2A	2B	2C	2D	2E	2F	30	31
50	32	33	34	35	36	37	38	39	3A	3B
60	3C	3D	3E	3F	40	41	42	43	44	45
70	46	47	48	49	4A	4B	4C	4D	4E	4F
80	50	51	52	53	54	55	56	57	58	59
90	5A	5B	5C	5D	5E	5F	60	61	62	63
100	64	65	66	67	68	69	6A	6B	6C	6D
110	6E	6F	70	71	72	73	74	75	76	77
120	78	79	7A	7B	7C	7D	7E	7F	–	–

Ex.: "pular" 79 endereços para frente:

localize a linha 70 e a coluna 9, e encontrará o valor hexadecimal 4F.

Desvio para trás

	–0	–1	–2	–3	–4	–5	–6	–7	–8	–9
–0	–	FF	FE	FD	FC	FB	FA	F9	F8	F7
–10	F6	F5	F4	F3	F2	F1	F0	EF	EE	ED
–20	EC	EB	EA	E9	E8	E7	E6	E5	E4	E3
–30	E2	E1	E0	DF	DE	DD	DC	DB	DA	D9
–40	D8	D7	D6	D5	D4	D3	D2	D1	D0	CF
–50	CE	CD	CC	CB	CA	C9	C8	C7	C6	C5
–60	C4	C3	C2	C1	C0	BF	BE	BD	BC	BB
–70	BA	B9	B8	B7	B6	B5	B4	B3	B2	B1
–80	B0	AF	AE	AD	AC	AB	AA	A9	A8	A7
–90	A6	A5	A4	A3	A2	A1	A0	9F	9E	9D
–110	92	91	90	8F	8E	8D	8C	8B	8A	89
120	88	87	86	85	84	83	82	81	80	–

Ex.: "pular" 53 endereços para trás:

localize a linha –50 e a coluna –3, e encontrará o valor hexadecimal CB.

EXERCÍCIOS.

1. *Assinale as afirmações corretas:*

   ( ) os desvios condicionais estão intimamente ligados aos flags;

   ( ) no que se refere ao Zero-Flag, podemos criar desvios somente quando este flag estiver com nível 0;

   ( ) a maior distância para frente é 127 posições de memória, e para trás até 128 posições.

2. *Escreva uma rotina que some os conteúdos dos endereços $0350, $0351 e de $0352. Quando o resultado for maior que 255, então o valor de A deverá ser zero. Caso contrário, o conteúdo de A deverá manter o resultado.*

3. *Qual é a instrução correta para desviar um programa 69 endereços para trás, quando o Carry-Flag for 1?*

4. *Subtraia X de A. Se o resultado for zero, então Y deve ser zerado. Caso contrário, o valor de Y deve ser $AA. Use o endereço $03F0 para auxiliar a subtração.*

5. *Some os conteúdos dos endereços $FFFA e $FDA8. Se o resultado for menor que 256, então este resultado deve ser transferido para X Caso contrário, o resultado deve ser transferido para Y.*

capítulo 11

jmp, jsr

No capítulo anterior, vimos desvios condicionais baseados nos flags criados. Agora trataremos de desvios que sempre serão realizados, independente dos estados dos flags (parecido com o GOTO do BASIC).

O primeiro desvio é o GOTO em si, que pode aparecer em duas versões:

JMP absoluto – OP-Code: 4C — aqui, o programa desvia diretamente para o endereço indicado. Esta instrução portanto ocupa 3 bytes (o código 4C, mais dois bytes que definem o endereço de destino).

JMP (indireto) – OP-Code: 6C — neste caso, o novo endereço de destino será formado pelo conteúdo do endereço indicado após o OP-Code 6C.

Ex.: JMP ($4318) – se no endereço $4318 temos o valor 8F, e no endereço a seguir ($4319) o valor 6A, então o novo endereço de destino será $6A8F.

O segundo desvio em linguagem de máquina lembra o GOSUB do BASIC, pois será chamada uma subrotina, após a qual (ao encontrar a instrução RTS) retorna ao programa principal:

JSR absoluto — OP-Code: 20 — também aqui temos que indicar um endereço, motivo pelo qual esta instrução necessita um total de 3 bytes.

Podemos combinar estes desvios com os desvios condicionais, principalmente quando a distância for grande demais para ser "pulada".

Ex.: Digamos que queremos realizar um desvio condicional para o endereço $3210 quando o resultado for zero. Suponha que a distância é grande demais para ser programado (mais que 127 posições).

Nesta caso, podemos fazer uma combinação conforme segue:

```
   .
   .
   .
BNE $03       D0 03       quando o resultado for dife-
JMP $3210     4D 10 32    rente de zero, então o des-
   .                      vio para $3210 não será
   .                      efetuado.
   .
```

Da mesma forma, podemos chamar subrotinas de acordo com a situação dos flags.

Ex.: Escrever uma rotina que transfira o conteúdo de A para o registrador X quando o resultado ultrapassar 255. Caso contrário, o resultado deverá ser transferido para o registrador Y (somar os conteúdos dos endereços $0380 e $0381). A transferência de A para Y deverá ser feita mediante uma subrotina.

Solução: Como estamos nos preocupando com o valor máximo 255, teremos que usar um desvio condicional baseado no Carry-Flag

(BCC ou BCS). Colocaremos nosso programa principal a partir do endereço $0300, e a subrotina a partir do endereço $0320. Assim, a nossa rotina se apresenta da seguinte forma:

```
0300 LDA $0380    AD 80 03 }  soma os conteúdo dos endereços
0303 ADC $0381    6D 81 03 }  $0380 e $0381
0306 BCS $030C    B0 04 ──►  desvia, se a soma for > 255
0308 JSR $0320    20 20 03─► chama a subrotina
030B BRK          00          e termina
030C TAX          AA ─────►  carrega X com o valor de A
030D BRK          00          e termina

0320 TAY          A8       }  carrega Y com o valor de A
0321 RTS          60       }  e volta (ao endereço $030B)
```

Programamos então primeiramente a nossa rotina principal:

```
*0300: AD 80 03 6D 81 03 B0 04 20 20 03 00 AA 00
```

e conferimos a listagem:

```
*0300L
```

Agora, programando a subrotina a partir do endereço $0320:

```
*0320: A8 60
```

Antes de rodar, teremos que definir os valores a serem somados, por exemplo $70 e $80, cuja soma *não* ultrapassa 255:

```
*0380: 70 80
```

Pronto, temos no endereço $0380 agora o valor $70, e em $0381 o valor $80. Vamos rodar a nossa rotina:

```
*0300G
030D - A=F0 X=00 Y=F0 P=80 S=F0
*
```

e, como esperávamos, o registrador Y assumiu o mesmo valor que A. Agora, provocando um resultado maior que 255, o registrador X deverá receber o mesmo valor de A. Para isto, colocaremos no endereço $0380 o valor B0, que somado ao conteúdo do endereço $0381 (= $80) ultrapassará 255:

```
*0308: B0
```

seguido de:

```
*0300G
030F - A=30 X=30 Y=F0 P=31 S=EE
```

P = 31 S = EE

Certo?

Mais uma observação referente a instrução JSR. Dentro de uma subrotina podemos chamar outra subrotina, na qual pode ser chamada outra subrotina, e assim por diante. Em linguagem de máquina não existe limite de subrotinas.

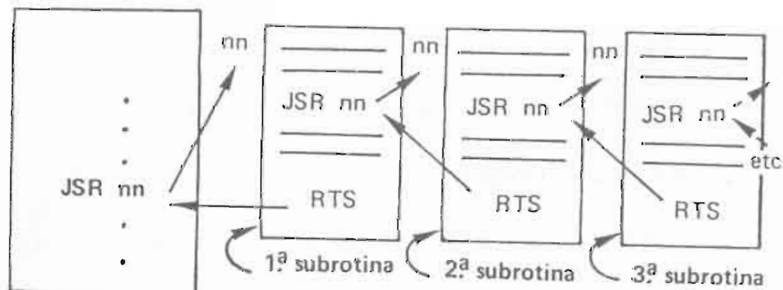


Figura 13.1 — Esquema da utilização de subrotinas sucessivas, em linguagem de máquina.

EXERCÍCIOS:

1. *Assinale as afirmações corretas:*

   ( ) *a instrução JMP pode ser usada para chamar subrotinas.*

   ( ) *o JMP só é usado junto com desvios condicionais.*

   ( ) *é possível chamar dentro de uma subrotina uma outra subrotina.*

2. *Escreva uma rotina que some os valores contidos nos endereços $0388 e $038A. Se o resultado for menor que 255, então seu micro deverá emitir um Bip (para acionar o Bip, chame a subrotina já existente no endereço $FBDD)*

capítulo 12

and, eor, ora

As instruções que veremos a seguir são chamadas de *operações lógicas*. São em princípio instruções que envolvem diretamente os *bits* dos bytes atribuídos. Sempre é feita uma comparação entre dois bits, cujo resultado, de acordo com a instrução usada, revela a relação dos bits comparados.

Parece confuso, não? Mas na verdade não é tão complicado como parece. A primeira instrução que trataremos é o AND:

AND – Operação lógica que resulta num bit 1, quando tanto o bit_n do primeiro byte como o bit_n do segundo byte estiverem com nível 1, ou seja:
se, por exemplo, o bit 6 do primeiro byte estiver com nível 1, e o bit 6 do segundo byte também com nível 1, então o bit 6 do resultado também será 1. Em qualquer outro caso, o bit correspondente ao resultado será 0. Isto é válido para todos os 8 bits.

Esta relação pode ser visualizada através de um circuito elétrico (vide Fig. 14.1), onde a chave A representa um determinado bit do primeiro byte, e a chave B o mesmo bit, porém do segundo byte. A lâm-

pada, por sua vez, corresponde ao respectivo bit de saída (resultado), isto é, quando a lâmpada acende, temos um bit 1, e quando apaga, um bit Ø:

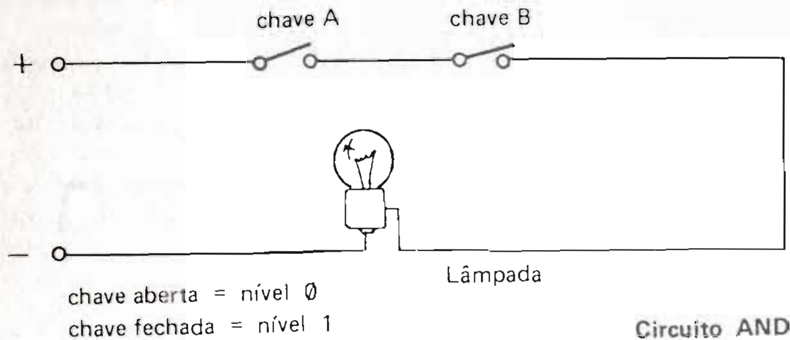


Figura 14.1 – Esquema da operação AND utilizando um circuito elétrico.

Na Fig. 14.1 podemos reparar claramente que a lâmpada somente acende quando tanto a chave A como a chave B estiverem fechadas (= bit 1). Podemos indicar esta dependência através de uma tabela especial, chamada *Tabela Verdade*:

	bit_n do 1. valor	bit_n do 2. valor	bit_n do resultado
	Ø	Ø	Ø
	Ø	1	Ø
AND	1	Ø	Ø
	1	1	1

Para reduzir o tamanho desta tabela, vamos substituir:

bit_n do 1º valor	por	A
bit_n do 2º valor	por	B
bit_n do resultado	por	C

e teremos esta mesma tabela da seguinte forma:

A	B	C
Ø	Ø	Ø
Ø	1	Ø
1	Ø	Ø
1	1	1

Novamente, analisando esta tabela verdade, podemos reparar que C somente será 1 quando tanto A como B estiverem em nível 1.

Usaremos esta tabela verdade para todas as demais operações lógicas. Aliás, não apenas em linguagem de máquina temos tabelas verdade. Elas são amplamente utilizadas em projetos de circuitos digitais, tal como no projeto do seu computador.

Vamos ver agora uma utilização desta operação lógica:

Vamos supor que queremos limitar um dado qualquer entre Ø e 15. Se nós olharmos para a configuração binária de todos os valores entre Ø e 15, podemos reparar que tanto o bit 4, como também os bits 5, 6 e 7 estão sempre com nível Ø:

```
Ex.:  +11   em binário = Ø Ø Ø Ø 1 Ø 1 1
      +15   em binário = Ø Ø Ø Ø 1 1 1 1
      +18   em binário = Ø Ø Ø 1 Ø Ø 1 Ø
```

Assim, resetando os bits 4 a 7, o resultado sempre será um valor entre Ø e 15, certo?

Vamos tomar como exemplo o valor hexadecimal 3A, que queremos reduzir resetando os bits 4 a 7. Usaremos para isto a instrução AND, onde o segundo valor define quais devem ser os bits a serem resetados (lembre-se da tabela verdade, onde o resultado somente será 1 quando tanto o bit_n do primeiro valor como o bit_n do segundo valor estiverem com nível 1).

```
3A = Ø Ø 1 1 1 Ø 1 Ø
     Ø Ø Ø Ø 1 Ø 1 Ø      operação AND, onde os bits
   ----------------       de 4 a 7 ficarão com nível Ø.
     Ø Ø Ø Ø 1 Ø 1 Ø
```

De fato, conseguimos resetar os quatro bits à esquerda. Mas, se nós usarmos um outro valor inicial, por exemplo 23, então o resultado muda, isto é, deixa de ser o mesmo valor original com os bits 4 a 7 resetados:

```
23 = Ø Ø 1 Ø Ø Ø 1 1
     Ø Ø Ø Ø 1 Ø 1 Ø      estamos usando o mesmo va-
   ----------------       lor como no exemplo acima
     Ø Ø Ø Ø Ø Ø 1 Ø
```

Agora não deu certo, pois para obter o resultado correto, o nosso segundo valor deveria ter a configuração binária 0 0 0 0 0 0 0 0 1 1, correto? Como então fazer para obter sempre o resultado correto, independente do valor inicial?

Ora, sabemos que teremos no resultado um nível 1, com bit_n1 e bit_n2 com nível um. Assim, aplicando a operação AND com 0 0 0 0 1 1 1 1, em todos os casos resetamos os bits 4 a 7, e os bits 0 a 3 permanecem inalterados:

Ex.: reduzir 3A	=	0 0 1 1 [1 0 1 0]	bit 0 a bit 3 não
	AND OF	0 0 0 0 1 1 1 1	alteram
		0 0 0 0 [1 0 1 0]	
reduzir 23	=	0 0 1 0 [0 0 1 1]	bit 0 a bit 3 não
	AND OF	0 0 0 0 1 1 1 1	alteram
		0 0 0 0 [0 0 1 1]	

Ficou bem mais fácil, não é? Da mesma forma para resetar, por exemplo, os bits 2 e 6 de um valor qualquer, a configuração binária seria 1 0 1 1 1 0 1 1 (= BB em hexadecimal), e para resetar apenas o bit 0, o valor a ser usado na instrução AND será 1 1 1 1 1 1 1 0.

A instrução lógica AND, como todas as demais operações lógicas, *sempre* é realizada entre o conteúdo do acumulador e um segundo valor. O resultado será colocado no acumulador, isto é, perde-se o valor original. E onde é definido este segundo valor? Bem, podemos tanto definir este valor diretamente, como usar um conteúdo de um endereço, por exemplo:

AND imediato — OP-Code: 29
AND página zero — OP-Code: 25
AND página zero,X — OP-Code: 35
AND absoluto — OP-Code: 2D
AND absoluto, X — OP-Code: 3D
AND absoluto,Y — OP-Code: 39
AND (indireto,X) — OP-Code: 21
AND (indireto),Y — OP-Code: 31

Aliás, se você olhar na tabela do apêndice C, verá que esta instrução afeta o zero-flag. Este flag será 1 quando o resultado desta operação for zero.

A próxima operação lógica que veremos é o ORA:

ORA — operação lógica que resulta num bit 1 quando pelo menos o bit_n1 ou o bit_n2 estiverem com nível 1. Caso contrário, isto é, bit_n1 e bit_n2 com nível 0, o resultado também será 0.

Também para esta operação temos um circuito elétrico equivalente, onde as chaves representam bit_n1 e bit_n2, e a lâmpada, o resultado:

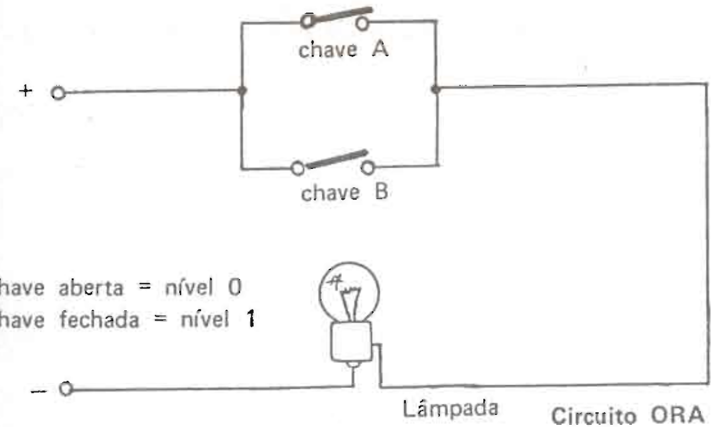


Figura 14.2 — Esquema da operação ORA, utilizando um circuito elétrico.

A tabela verdade para a operação ORA é a seguinte:

bit_n1	bit_n2	resultado
0	0	0
1	0	1
0	1	1
1	1	1

ou

A	B	C
0	0	0
1	0	1
0	1	1
1	1	1

Nós estávamos usando a operação AND para resetar determinados bits, lembra-se? Agora, usando o ORA, podemos fazer exatamente o contrário, ou seja, levar determinados bits para nível 1:

Ex.: Levar o bit 7 de um valor qualquer para nível 1, e deixar os demais bits inalterados:

```
valor inicial = 59 –  0 [1 0 1 1 0 0 1]
          ORA         1  0 0 0 0 0 0 0    bit 0 a 6 inalterados
      resultado 1     1 [1 0 1 1 0 0 1]
valor inicial = 07 –  0 [0 0 0 0 1 1 1]
          ORA         1  0 0 0 0 0 0 0    bit 0 a 6 inalterados
      resultado       1 [0 0 0 0 1 1 1]
```

Resumindo:

resetar bits: usar a instrução AND, onde os bits a serem resetados devem estar com nível 0 no segundo valor.

setar bits: usar a instrução ORA, onde os bits a serem setados devem estar com nível 1 no segundo valor.

E, semelhante a instrução AND, podemos definir este segundo valor de diversas maneiras:

ORA imediato	– OP-Code:	09
ORA página zero	– OP-Code:	05
ORA página zero,X	– OP-Code:	15
ORA absoluto	– OP-Code:	0D
ORA absoluto,X	– OP-Code:	1D
ORA absoluto,Y	– OP-Code:	19
ORA (indireto,X)	– OP-Code:	01
ORA (indireto),Y	– OP-Code:	11

Resta ainda a terceira e última operação lógica que o 6502 é capaz de fazer: o EOR.

EOR – operação lógica que resulta num bit 1 quando *ou* $bit_n 1$ *ou* $bit_n 2$ estiverem com nível 1. É uma condição ORA exclusiva, isto é, a resposta somente será verdadeira se os bits em comparação forem *diferentes*.

O circuito equivalente para visualizar melhor este critério é apresentado na figura 14.3, onde a chave A e a chave B são duplas. Quando

acionadas, um contato fecha, e o outro abre. Assim sendo, a lâmpada acende somente quando apenas *uma* chave for acionada.

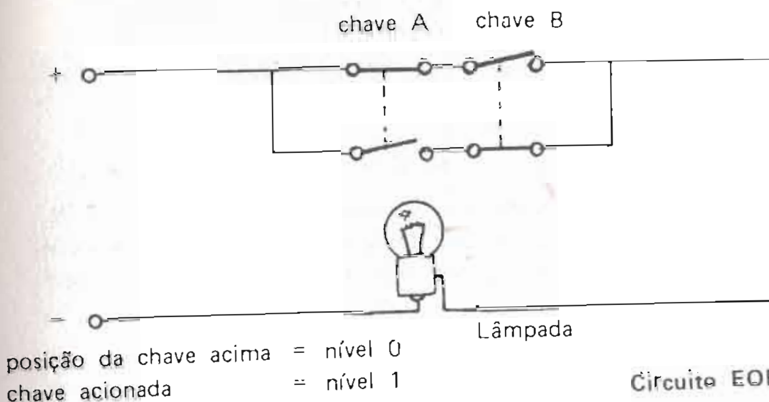


Figura 14.3 – Esquema da operação EOR, utilizando um circuito elétrico.

A tabela verdade para a instrução EOR é a seguinte:

A	B	C
0	0	0
1	0	1
0	1	1
1	1	0

compare esta tabela com a tabela verdade da instrução ORA e tente ver a diferença.

Também aqui está sendo feita uma comparação entre o valor do acumulador e um segundo valor definido de acordo com o quadro abaixo, e o resultado desta operação lógica será colocado em A:

EOR imediato	– OP-Code:	49
EOR página zero	– OP-Code:	45
EOR página zero,X	– OP-Code:	55
EOR absoluto	– OP-Code:	4D
EOR absoluto,X	– OP-Code:	5D
EOR absoluto,Y	– OP-Code:	59
EOR (indireto,X)	– OP-Code:	41
EOR (indireto),Y	– OP-Code:	51

Depois de tanta teoria, vamos digitar um pouco no nosso computador?

Vamos realizar alguns exemplos que envolvem as operações lógicas AND, ORA e EOR:

1. Carregar no acumulador o conteúdo do endereço $D66E e resetar os bits 0 a 3.

Solução:
```
LDA $D66E          AD 6E D6
```

Agora já temos em A o conteúdo do endereço $D66E. Para resetar os bits 0 a 3, teremos que usar a instrução AND cujo segundo valor apresente os bits 0 a 3 com nível 0:

A configuração binária deste valor é portanto **1 1** 1 1 0 0 0 0, que corresponde ao valor hexadecimal F0. Mas, para efeito de controle, vamos guardar o valor original de A no registrador X, pois a operação AND altera este valor original.

```
TAX                AA
AND #$F0           29 F0
BRK                00
```

Temos então a seguinte rotina que iremos colocar no computador:

```
LDA $D66E          AD 6E D6
TAX                AA
AND #$F0           29 F0
BRK                00
```

```
*0300: AD 6E D6 AA 29 F0 00
*
```

Confira os mnemônicos com 0300L. Se tudo estiver certo, rode o programa:

```
*0300G
0308 - A=A0  X=A4  Y=00  P=B0  S=F0
*
```


Se olharmos no registrado X, vemos o valor original, ou seja, $A4 Com este programa resetamos os bits 0 a 3 de tal modo que o resultado agora em A é A0. Tente fazer uma rotina que reseta agora os bits 4 a 7 (use o mesmo valor original). O resultado deverá ser 04, certo?

2. Carregar em Y o conteúdo do endereço $FF7B, transferir este valor para A e levar os bits 0, 2 e 3 para nível 1:

Solução:
```
LDY $FF7B          AC 7B FF
TYA                98
```

Com o valor em A, podemos iniciar a operação para setar os bits 0, 2 e 3. Usaremos para isto a operação lógica ORA com um valor que tenha os bits 0, 2 e 3 com nível 1, ou seja:

```
0 0 0 0 1 1 0 1 = 0D em hexadecimal
```

```
ORA #$0D           09 0D
BRK                00
```

Portanto, a rotina completa será esta:

```
LDY $FF7B          AC 7B FF
TYA                98
ORA #$0D           09 0D
BRK                00
*0300: AC 7B FF 98 09 0D 00
*
```

Também aqui, antes de rodar, confira os mnemônicos com 0300L

```
*0300G
0308 - A=3D  X=A4  Y=30  P=30  S=EE
```

Em Y temos o valor original do endereço $FF7B, ou seja, $30 Como resultado obtivemos 3D (valor de A). Vamos ver o que aconteceu:

```
valor original 30 = 0 0 1 1 0 0 0 0
          ORA 0D = 0 0 0 0 1 1 0 1
                   ---------------
       resultado = 0 0 1 1 1 1 0 1    (que é realmente $3D)
```

EXERCICIOS:

1. *Assinale as afirmações corretas:*
   ( ) *todas as operações lógicas envolvem o conteúdo do acumulador*
   ( ) *para resetar bits podemos usar a operação lógica AND*
   ( ) *o inverso da operação ORA é a operação lógica EOR*
   ( ) *quando o resultado de uma operação AND for zero, o zero-flag será 1*

2. *Escreva um programa que transfira o conteúdo do endereço $D405 para o endereço $0350. A seguir, este mesmo valor, porém com bits 7 e 3 resetados, deve ser colocado no endereço $0351.*

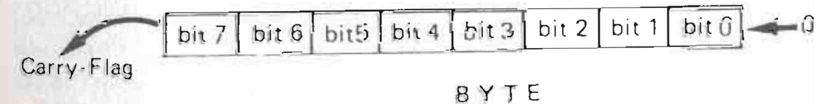
3. *Realize uma operação EOR entre o conteúdo do endereço $D42D e o do endereço $D422.*

capítulo 13

asl, rol, ror, lsr

As instruções a seguir permitem movimentar os bits dentro de um determinado byte, tanto para a esquerda como para a direita. Temos movimentos de deslocamento e movimentos de rotação dos bits, como veremos a seguir:

ASL — desloca todos os bits de um byte para a esquerda, isto é, o bit 0 original passa a ser o novo bit 1, o bit 1 original passa a ser o novo bit 2, e assim por diante. O bit 7 original será colocado no Carry-Flag, e no lugar do bit 0 original entrará um nível 0 (veja figura 15.1).



ASL — deslocamento para esquerda

Figura 15.1 — Deslocamento dos bits para a esquerda.

O que podemos fazer com este recurso do 6502? Qual é a vantagem de deslocar os bits uma casa para a esquerda?

Vamos ver isto através de um exemplo:

```
Valor original = 06 - 0 0 0 0 0 1 1 0
deslocando 1 casa p/
a esquerda, temos:     0 0 0 0 1 1 0 0, certo?
```

Calculando o valor decimal correspondente, teremos agora o valor 12. Já percebeu alguma coisa? Vamos deslocar novamente este byte para a esquerda:

```
0 0 0 1 1 0 0 0
```

Agora, o valor decimal passou a ser 24. Se fizermos novamente este deslocamento, teremos como resultado 48, na próxima vez 96, depois 192.

De fato, multiplicamos por 2 ao deslocar todos os bits para a esquerda! Este procedimento fornece sempre o dobro do valor original, enquanto o bit 7 estiver com nível 0. Pois existe o limite 255, que não pode ser ultrapassado. Se isto acontecer, o resultado final não corresponde mais ao valor real!

Vamos tomar, por exemplo, o valor $A5, que corresponde ao valor decimal 165. Duplicando este valor, o resultado deveria ser 330, certo? Mas como o valor máximo de um registrador ou de uma posição na memória só pode ser 255, qual será o resultado após a instrução ASL?

```
Valor original = A5 - 1 0 1 0 0 1 0 1
      após ASL - 0 1 0 0 1 0 1 0
```

O resultado correto seria 1 0 1 0 0 1 0 1 0 (faça as contas), mas onde foi parar o nosso nono bit? Temos apenas 8 bits para representar o resultado, e consultando a Fig. 15.1, podemos ver que o nono bit realmente existe! O bit 7 original (que será o nono bit após ASL) entra no Carry-Flag. Assim, cada vez que o resultado final ultrapassar o valor 255, o nosso Carry-Flag terá um nível lógico 1, certo? Basta então o uso de instruções condicionais (por exemplo BCC ou BCS — veja o capítulo 12), para indicar de alguma maneira a ultrapassagem ocorrida.

Exemplo: Fazer uma multiplicação sucessiva por 2, a partir do valor 1, até que o resultado seja maior que 255. O registrador X deve manter o resultado máximo real.

Solução: Antes de tudo, vamos definir o valor original e resetar o Carry com a instrução CLC:

```
0300  LDA #$01   A9 01
0302  CLC        18
```

A partir daqui, podemos iniciar a nossa rotina principal que começa no endereço 0303:

```
0303  TAX        AA    transfere o conteúdo de A para X
0304  ASL A      0A    multiplica A por 2
```

Nesta altura, devemos testar o Carry-Flag. Se este flag estiver com nível 0, então está tudo bem e podemos multiplicar A novamente por 2. Teremos, portanto, um desvio condicional para o endereço 0303, enquanto o Carry-Flag estiver com nível 0:

```
0305  BCC $0303  90 FC
```

Se o Carry-Flag estiver com nível 1, então o desvio não será feito, e o programa deve parar:

```
0307  BRK  00
```

A rotina completa é esta:

```
0300  LDA #$01   A9 01
      CLC        18
0303  TAX        AA
      ASL A      0A
      BCC $0303  90 FC
      BRK        00

*0300: A9 01 18 AA 0A 90 FC 00
*
```

Antes de rodar, nunca esqueça de conferir os mnemônicos!

```
*0300G
0309 - A=00 X=80 Y=00 P=33 S=EC
*
```

Temos em X o valor $80, que corresponde ao valor decimal 128. Se nós fizermos a conta, multiplicando 1 por 2, e assim sucessivamente, encontraremos também o valor 128 como valor máximo, pois após a próxima multiplicação o resultado seria 256, portanto maior que 255.

Aqui temos a tabela da instrução ASL, que permite o uso de valores do acumulador de conteúdos de determinados endereços na memória:

ASL acumulador	— OP-Code: 0A
ASL página zero	— OP-Code: 06
ASL página zero,X	— OP-Code: 16
ASL absoluto	— OP-Code: 0E
ASL absoluto,X	— OP-Code: 1E

A próxima instrução é exatamente o inverso da instrução ASL.

LSR — desloca todos os bits de um determinado byte para a direita. O bit 7 original passa a ser o novo bit 6, o bit 6 original passa a ser o novo bit 5, e assim por diante. O novo bit 7 será 0, e o valor do bit 0 original define o Carry-Flag (veja a fig. 15.2).

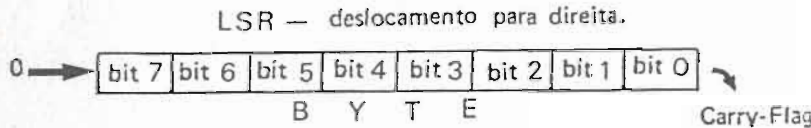


Figura 15.2 — Deslocamento dos bits para a direita.

Se na instrução ASL multiplicamos o valor original por 2, pela lógica podemos agora dividir valores por 2 usando a instrução LSR:

```
valor original = 2A  -  0 0 1 0 1 0 1 0  (42 em decimal)
         após LSR  -  0 0 0 1 0 1 0 1  (21 em decimal)
```

Também aqui pode acontecer do resultado não corresponder mais ao valor real. Isto acontece quando o número é ímpar e queremos dividir por 2. Então semelhante ao que fizemos nas rotinas com ASL, teremos que usar instruções condicionais para evitar resultados irreais.

Exemplo: Testar o conteúdo do endereço $FF39. Se o valor encontrado for par, então X e Y devem manter o valor 01. Caso contrário, o valor de X e Y devem estar com o valor 0.

Solução: Para testar se um número é par ou ímpar, basta dividí-lo por 2 e testar o Carry-Flag. Se este Flag estiver com nível 0, então saberemos que o número era par. Caso contrário, o número era ímpar.

```
0300   LSR $FF39   4E 39 FF   "divide" o conteúdo por 2
0303   BCC $030A   90 05      desvia se for par (Carry = 0)
0305   LDX #$01    A2 01      é ímpar, portanto X e Y = 0
0307   LDY #$01    A0 01
0309   BRK 00
030A   LDX #$00    A2 00      é ímpar, portanto X e Y = 0
030C   LDY #$00    A0 00
030E   BRK         00

*0300: 4E 39 FF 90 05 A2 01 A0 01 00 A2 00 A0 00 00
*
```

E, após conferir tudo, vamos rodar:

```
*0300G
0308 - A=00 X=01 Y=01 P=31 S=E6
*
```

Como X e Y estão com o valor 1, devemos presumir que o valor contido no endereço $FF39 é ímpar. Confira e tente outros endereços:

Aqui a tabela da instrução LSR:

LSR acumulador	— OP-Code: 4A
LSR página zero	— OP-Code: 46
LSR página zero,X	— OP-Code: 56
LSR absoluto	— OP-Code: 4E
LSR absoluto,X	— OP-Code: 5E

As próximas duas instruções são semelhantes ao ASL e LRS, com a diferença que há uma rotação dos bits, e não mais um simples deslocamento:

ROL — roda todos os bits de um determinado byte para esquerda. O bit 7 original define o novo Carry-Flag, e no lugar do novo bit 0 entrará o valor do Carry-Flag anterior (vide fig. 15.3).

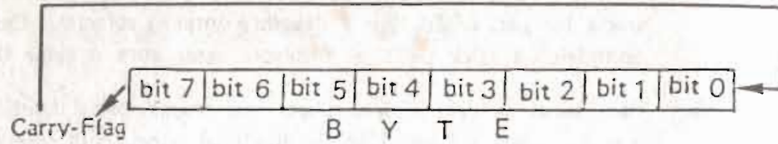


Figura 15.3 — Rotação dos bits para a esquerda.

ROR — roda todos os bits de um determinado byte para direita. O bit 0 original define o novo Carry-Flag, e no lugar do novo bit 7 entrará o valor do Carry-Flag anterior (vide fig. 15.4).

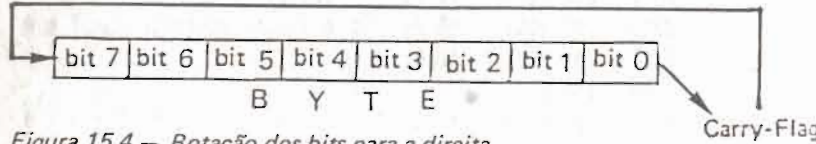


Figura 15.4 — Rotação dos bits para a direita.

E, finalmente, as tabelas referentes às instruções ROL e ROR:

ROL acumulador	— OP-Code: 2A
ROL página zero	— OP-Code: 26
ROL página zero,X	— OP-Code: 36
ROL absoluto	— OP-Code: 2E
ROL absoluto,X	— OP-Code: 3E
ROR acumulador	— OP-Code: 6A
ROR página zero	— OP-Code: 66
ROR página zero,X	— OP-Code: 76
ROR absoluto	— OP-Code: 6E
ROR absoluto,X	— OP-Code: 7E

EXERCÍCIOS:

1. *Assinale as afirmações corretas:*

   ( ) *podemos dividir valores deslocando os bits para esquerda*

   ( ) *a instrução LSR desloca os bits para direita*

   ( ) *podemos usar a instrução ROL para dividir por 2*

2. *Escreva uma rotina que multiplique o valor 34 por 4, usando uma instrução de rotação.*

capítulo 14

CMP, CPX, CPY, BIT

As próximas instruções permitem a você testar se um valor é igual, menor ou maior que um segundo valor. A partir daí, usando desvios condicionais, podemos tomar decisões dentro do nosso programa.

Este tipo de comparação, em princípio, é uma subtração, com a diferença que o resultado desta subtração não aparece, só influencia os flags do registrador de Status. Olhando na tabela das instruções, temos para este tipo de comparação, por exemplo, a instrução CMP, que afeta os seguintes flags:

Zero-Flag
Carry-Flag
Sign-Flag (N-Flag)

Assim, usando a instrução *CMP*, é feita uma comparação entre o conteúdo do acumulador e um outro valor definido por nós. Como resultado desta comparação podemos ter um dos três resultados a seguir:

1. o segundo valor é menor que o valor de A
2. o segundo valor é igual ao valor de A
3. o segundo valor é maior que o valor de A

Após esta comparação, tanto o conteúdo de A como o segundo valor definido por nós permanecem inalterados. Resta, portanto, consultar os flags para saber qual dos três resultados acima foi obtido. Para isto, vale o seguinte critério:

Se o segundo valor é menor

— neste caso, a "subtração" resultou num valor positivo, estando portanto com o Carry-Flag em nível 1 (lembre que em subtrações o Carry-Flag é inverso ao da adição – consulte cap. 10). E, como o resultado não é zero, o zero-flag terá um nível 0.

Se o segundo valor é igual

— agora, como o resultado da "subtração" é zero, o nosso Zero-Flag assume o valor 1. O Carry-Flag continua com nível 1, pois o valor 0 é considerado como um valor positivo pela CPU.

Se o segundo valor é maior

— o resultado da "subtração" é negativo, portanto não é zero. Assim, o Zero-Flag será 0, e o Carry-Flag também.

Para visualizar melhor, vamos dar um exemplo:

Suponhamos que o conteúdo do acumulador seja $30. Comparando este valor com os valores $10, $30 e $50, temos a seguinte situação dos flags:

valor de A	segundo valor	Carry-Flag	Zero-Flag
$30	$10	nível 1	nível 0
$30	$30	nível 1	nível 1
$30	$50	nível 0	nível 0

E o valor do N-Flag depende do bit 7 do resultado da comparação. Se o bit 7 estiver com nível 1, então o N-Flag também será 1. E vice-versa.

Este segundo valor pode ser definido através de uma instrução imediata, ou ser definido através de um conteúdo de um endereço de acordo com a seguinte tabela:

CMP imediato	– OP-Code: C9
CMP página zero	– OP-Code: C5
CMP página zero,X	– OP-Code: D5
CMP absoluto	– OP-Code: CD
CMP absoluto,X	– OP-Code: DD
CMP absoluto,Y	– OP-Code: D9
CMP (indireto,X)	– OP-Code: C1
CMP (indireto),Y	– OP-Code: D1

Vamos a um exemplo prático:

Exemplo: Somar sucessivamente o valor 1 ao acumulador (valor inicial = $05), até que o resultado atinja o valor $EE.

Solução: Vamos definir incialmente o nosso acumulador com o valor $05:

```
0300  LDA #$05  A9 05
```

Agora, somaremos o valor 1 ao acumulador:

```
0302  ADC #$01  69 01
```

A seguir, verificamos se já atingimos o valor $EE:

```
0304  CMP #$EE  C9 EE
```

Se não for, isto é, se o resultado for diferente de zero, então devemos repetir a nossa soma. Usaremos para este desvio condicional o BNE (desvie se Z = 0):

```
0306  BNE $0302  D0 FA
```

Se o conteúdo de A for $EE, então o programa continua com:

```
0308  BRK        00
```

A rotina completa:

```
0300  LDA #$05   A9 05
0302  ADC #$01   69 01
      CMP #$EE   C9 EE
      BNE $0302  D0 FA
      BRK        00
```

Digite, rode o programa e confira se o acumulador realmente atingiu o valor $EE.

O que foi feito com o acumulador até agora pode ser feito também com os registradores X e Y. Prevalece o mesmo critério dos flags, com a diferença que agora o segundo valor definido será "subtraído" de X ou de Y. A definição do segundo valor em conjunto com X ou Y mostra a tabela a seguir:

Comparações com o valor de X:

CPX imediato — OP-Code: E0
CPX página zero — OP-Code: E4
CPX absoluto — OP Code: EC

Comparações com o valor de Y:

CPY imediato — OP-Code: C0
CPY página zero — OP-Code: C4
CPY absoluto — OP-Code: CC

Lembra-se do FOR-NEXT do nosso BASIC? Usando as instruções acima, podemos fazer algo similar:

Exemplo: Multiplicar o valor $0C (contido em A) por 9.

Solução: Como não temos uma instrução específica para realizar multiplicações em linguagem de máquina (exceto na base 2), faremos somas repetidas, no caso, somando nove vezes o valor $0C.

Para controlar a quantidade das somas, usaremos, por exemplo, o registrador X, que será incrementado após cada soma realizada. Assim que X atinge o valor $09, sabemos que terminamos o nosso cálculo. Ainda em função da instrução ADC usada, que soma junto ao resultado o valor do Carry-Flag (vide capítulo 10), temos de resetar inicialmente este flag:

```
0300  CLC         18      reseta o Carry-Flag
0301  LDA #$00    A9 00   define A = 0
0303  LDX #$00    A2 00   define o contador X = 0
0305  ADC #$0C    69 0C   soma $0C ao acumulador
0307  INX         E8      incrementa o contador
0308  CPX #$09    E0 09   testa se já foram feitas 9 somas
030A  BNE $0305   D0 F9   se não for, faça a próxima soma
030C  BRK         00      fim da rotina
```


Programamos então o nosso micro a partir do endereço 0300:

```
*0300: 18 A9 00 A2 00 69 0C E8 E0 09 D0 F9 00
*
```

e conferimos os mnemônicos com 0300L.

Rodando o nosso programa, teremos:

```
*0300G
030E - A=6C  X=09  Y=00  P=33  S=B2
*
```

Vamos fazer a prova? Multiplicamos o valor $0C por 9. $0C vale 12 em decimal. O resultado desta multiplicação é 108 em decimal. O que achamos no acumulador? Converta o valor $6C para decimal!

Temos ainda uma instrução que pode ser enquadrada nas instruções de comparações, que é a instrução *BIT*. Esta instrução, em princípio, é semelhante a instrução AND (cap 14), só com uma diferença: o resultado *não* aparece no acumulador, isto é, o valor original será mantido

Como nas instruções CMP, CPX e CPY, a instrução BIT altera somente os flags a seguir:

Zero-Flag (será 1, quando o "resultado" da operação AND é zero)
N-Flag (assume o valor do bit 7 do segundo valor)
V-Flag (assume o valor do bit 6 do segundo valor)

A instrução BIT compara sempre o conteúdo do acumulador com um conteúdo de um endereço da memória:

BIT página zero — OP-Code: 24
BIT absoluto — OP-Code: 2C

Exemplo: Se o acumulador tiver o valor $37, e no endereço $6E33 o valor $2F, então a instrução BIT $6E33 altera os flags da seguinte maneira:

```
    valor de A = $37  -  0 0 1 1 0 1 1 1
valor em $6E33 = $2F  -  0 0 1 0 1 1 1 1  (operação "AND")
                         0/0 1 0 0 1 1 1  ("resultado teórico")

      N-Flag  V-Flag          como o resultado teórico não
                              é zero, então o Zero-Flag
                              será 0
```

Obs.: Lembre-se que no acumulador permanece o valor $37, e no endereço $6E33 o valor $2F.

EXERCÍCIOS:

1. *Assinale as afirmações corretas:*

   ( ) as instruções CMP, CPX e CPY alteram os valores originais;

   ( ) as instruções CMP, CPX e CPY alteram somente os flags;

   ( ) a instrução BIT é semelhante à instrução AND, só que o conteúdo de A permanece inalterado.

2. *Escreva uma rotina que multiplique $09 (no acumulador) por $1E. A seguir, o resultado deve ser comparado com o conteúdo do endereço $FF37. Se o valor neste endereço for igual ao resultado, então os registradores X e Y devem ser carregados com o valor $00. Caso contrário, o valor em X e Y deve ser $FF.*

capítulo 15

pha, php, pla, plp

Vamos supor que você queira chamar uma subrotina dentro de um programa escrito em linguagem de máquina. Se você leu atentamente o capítulo 13, já deve estar pensando na instrução JSR, não é mesmo? Mas, se os valores atuais de A, X ou Y – antes mesmo de chamar a subrotina – devem ser mantidos, qual é a solução para não perder estes valores durante a execução da subrotina?

Uma possibillidade seria guardar estes valores temporariamente em algum endereço da memória, e após rodar a subrotina, recolocá-los nos respectivos registradores.

Ex.: Chamar uma subrotina sem perder os conteúdos de A, X e Y:

1ª Solução – primeiramente, teremos de encontrar três endereços livres para guardar os nossos valores. Escolheremos, por exemplo, os endereços $0350, $0351 e $0352. Assim, a nossa rotina teria o seguinte aspecto:

```
STA $0350        8D 50 03
STX $0351        8E 51 03
STY $0352        8C 52 03
JSR $nnnn        20 nn nn   (chama a subrotina)
LDY $0352        AC 52 03
LDX $0351        AE 51 03
LDA $0350        AD 50 03
```

Gastamos para este artifício nada menos que 18 bytes. Se nós escolhemos três endereços livres na página zero, ainda gastaríamos 12 bytes!

Para este fim, isto é, para guardar valores temporariamente, temos à nossa disposição uma memória especial, chamada *Stack*.

Mas o que é um Stack?

Trata-se de uma parte da memória capaz de guardar até 256 bytes. Para colocar ou retirar valores desta memória existem instruções específicas, que, por incrível que pareça, não definem nenhum endereço! Como então a CPU sabe qual o valor a ser retirado desta memória?

A resposta é a própria estrutura do Stack. Ele é acessado de tal maneira que forneça sempre o último valor colocado no stack. Confuso? Em outras palavras: imagine uma pilha de discos que você forma colocando um disco em cima do outro. Depois de formar a pilha, qual é o disco que você pode retirar primeiro? – O último disco colocado! E quando você pode retirar o primeiro disco da sua pilha? – Quando já retirou todos os demais!

Exatamente desta forma trabalha o stack. Se nós colocarmos pela seqüência os valores de A, X e Y no stack, então teremos de retirar estes valores começando pelo valor de Y, depois o valor de X, e no final o valor de A

Infelizmente não existem instruções para levar os conteúdos de X e Y diretamente para o Stack. Somente para o acumulador temos uma instrução específica para este fim:

PHA – OP-Code 48 (transfere o conteúdo de A para o stack)
PLA – OP-Code 68 (transfere o conteúdo do stack para A)

Para guardar também os conteúdos de X e Y, teremos que transferi-los primeiro para A, e depois aplicar as instruções que operam com o stack.

Assim, para o exemplo do início desta parte, teremos agora uma segunda solução:

2ª Solução

```
PHA     48        (guarda o valor de A no stack)
TXA     8A
PHA     48        (guarda o valor de X no stack)
TYA     98
PHA     48        (guarda o valor de Y no stack)
JSR $nnnn  20 nn nn (chama a subrotina)
PLA     68
TAY     A8        (devolve ao Y o valor original)
PLA     68
TAX     AA        (devolve ao X o valor original)
PLA     68        (devolve ao A o valor original)
```

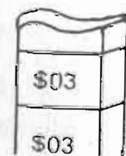
Gastamos nesta segunda solução apenas 10 bytes, com o mesmo efeito. Ainda temos a vantagem de não termos usado posições da memória como na solução 1.

Aliás, neste mesmo stack são guardados também os endereços de retorno ao chamar uma subrotina. Este endereço também obedece a ordem dos valores colocados no stack. Desta forma, é muito importante deixar o stack *sempre* vazio antes de usar a instrução RTS. Caso contrário, a CPU interpreta o próximo valor disponível no stack como o endereço de retornar, que com certeza não corresponde ao endereço real.

Exemplo:

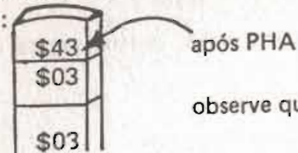
```
0300   JSR $0360 ← antes de desviar para o endereço $0360,
0303   BRK         será colocado no stack pela CPU o ende-
                   reço de retorno, ou seja, o endereço
                   $0303.
```

A situação do stack agora é a seguinte:



← topo do stack. Atualmente temos aqui o endereço de retorno após completar a subrotina (RTS).

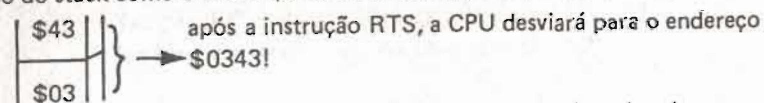
Vamos supor que na subrotina do endereço $0360 transferimos o conteúdo de A (valor $43) para o stack. O stack terá então a seguinte configuração:



após PHA

observe que o endereço de retorno se deslocou

Se no decorrer do programa nós não retirarmos o valor colocado do stack, ao atingir a instrução RTS, a CPU interpretará o endereço presente no topo do stack como o endereço de retorno (que deveria ser $0303):



Além do acumulador, podemos também guardar o registrador de status no stack. Isto é particularmente útil, quando se quer manter as informações de uma operação anterior. Para colocar e retirar o registrador de status do stack, temos as seguintes instruções:

PHP — OP-Code: 08 (transfere o conteúdo do status para o stack)
PLP — OP-Code: 28 (transfere o conteúdo do stack para o status)

A CPU não memoriza se foi o acumulador ou o status que forneceu um valor ao stack. Assim é possível colocar em A o valor do status através da seqüência:

```
PHP
PLA
```

Descubra por quê.

Exercícios:

1. *Assinale as afirmações corretas:*

   ( ) *o stack é uma memória para guardar valores temporariamente e endereços de retorno de subrotinas.*

   ( ) *podemos colocar até 128 bytes no stack.*

   ( ) *não existem instruções que levem os conteúdos de X e Y diretamente para o stack.*

2. *Escreva uma rotina que transfira o conteúdo do status para o registrador Y.*

capítulo 16

tsx, txs

No capítulo anterior falamos de uma área do stack que é capaz de manter até 256 bytes. Mas em que parte se encontra este stack na memória?

Lembra-se da página zero? É uma área da memória cujo endereço mais significativo é $00 (daí o nome página *zero*), que vai de $0000 até $00FF. A próxima página de 256 bytes é chamada de página um, que vai de $0100 até $01FF. E exatamente aqui se encontra o nosso stack. Deste modo, devemos evitar colocar valores nesta área para não correr o risco de destruir dados ou endereços importantes existentes nesta memória.

Temos que ter, portanto, algum indicador que possa localizar o próximo dado existente do stack. Este indicador é chamado Stack-Pointer (apontador do stack), cujo valor muda a cada vez que alteramos a quantidade dos bytes existentes no stack.

Às vezes é desejável que se altere o valor contido neste Stack-Pointer, ou verificar em que endereço ele aponta. Aliás, quando falamos em apontar um endereço no stack, estamos nos limitando apenas aos bits menos significativos do endereço (de $00 até $FF), e a CPU já fornece automaticamente o byte mais significativo $01 da página um.

Como não existem instruções que permitam ler diretamente o valor do stack-pointer (algo como LD), e também não existem instruções para alterar este registrador (algo como STA); foram criadas duas instruções que possibilitam transferir o conteúdo do stack-pointer para o registrador X e vice-versa.

São estas as instruções:

TSX – OP-Code: BA — transfere o conteúdo do stack-pointer para o registrador de índex X.

TXS – OP-Code: 9A — transfere o conteúdo do registrador de índex X para o stack-pointer.

Assim, se quiséssemos por exemplo guardar o conteúdo do stack-pointer no endereço 0350, teríamos que escrever:

```
0300  TSX        BA
0301  STX $0350  SE 50 03
```

e para recolocar este valor de volta:

```
0304  LDX $0350  AE 50 03
0307  TSX        9A
```

Tudo certo? Creio que dificilmente serão usadas estas instruções, mas como elas fazem parte do quadro das instruções da CPU 6502, estão à sua disposição para eventuais explorações.

Exercícios:

1. *Assinale as afirmações corretas:*

   ( ) *o conteúdo do stack-pointer pode ser alterado diretamente.*

   ( ) *a instrução TXS coloca o valor do stack-pointer em X.*

   ( ) *a única maneira de alterar o stack-pointer é através da instrução TXS.*

2. *Escreva um programa que carregue o acumulador com o valor do stack-pointer e transfere o conteúdo de Y para ele.*

capítulo 17

sei, cli, rti

Estas últimas instruções tratam o comportamento da CPU em rotinas de interrupção. Existe no seu computador uma linha de interrupção que geralmente é chaveada por um periférico (o drive, por exemplo). Neste caso, podemos programar a CPU para aceitar ou não a interrupção solicitada.

Um outro exemplo é o gerador de pulso externo, que fornece um pulso a cada segundo. Este pulso aciona a linha de interrupção, que por sua vez desvia a CPU para uma rotina especial para manter, por exemplo, sempre o horário certo na tela. Neste caso, a linha de interrupção sempre deve estar programada para reconhecer os sinais de interrupção.

E quem mantém o modo programado, é nada mais do que o bit 2 do registrador de status, ou seja, o Interruptor-Flag. Se este flag estiver com nível 1, então os sinais de interrupção serão reconhecidos. Caso contrário, isto é, com o Interrupt-Flag em nível 0, os sinais de interrupção serão ignorados pela CPU.

Para programar este flag, temos as seguintes instruções:

SEI – OP-Code: 78 leva o Interrupt-Flag para nível 1. Os sinais de interrupção serão reconhecidos.

CLI – OP-Code: 58 leva o Interruptor-Flag para nível 0. Os sinais de interrupção serão ignorados.

Foi dito que a CPU desvia para atender uma rotina de interrupção quando o Interrupt-Flag estiver em 1 e quando ocorreu um sinal de interrupção. Mas, no momento em que o sinal de interrupção foi dado por um periférico, a CPU provavelmente estava executando alguma coisa. Esta tarefa anterior terá continuidade após rodar a rotina de interrupção, não acha? Por isso, antes do desvio em si, a CPU guarda no stack todas as referências do programa original (por exemplo, o valor do contador de programa), que serão recuperadas após terminar a rotina de interrupção.

Para indicar à CPU que a rotina de interrupção chegou ao fim, existe uma instrução semelhante ao RTS, com a diferença que o valor contido no stack será colocado no program-counter (contador de programa). Esta instrução é:

RTI – OP-Code: 40 indica à CPU o final de uma rotina de interrupção. Após encontrar esta instrução, a CPU continua com a execução normal do programa.

Exercícios:

1. Assinale as afirmações corretas:

( ) *as interrupções são provocadas por periféricos.*

( ) *com o interrupt-Flag em nível 0, os sinais de interrupção serão ignorados.*

( ) *uma rotina de interrupção pode ser terminada com a instrução RTS.*

capítulo 18

etiquetas

A partir das próximas páginas veremos todos os mnemônicos da CPU 65Ø2, um por um. Procuraremos manter uma seqüência lógica, iniciando com as instruções mais usuais, até atingir as instruções menos usadas.

Como trata-se de um curso prático, à medida que você estuda as operações da CPU, sempre aparecerão exemplos que possam ser digitados no seu computador. Se você tiver um microcomputador TK 2ØØØ, então poderá usar a opção LM ou o próprio mini-assembler existente, acessado com ASS. Para eventuais dúvidas, consulte o seu manual técnico.

Já, os proprietários de micros APPLE II (PLUS, SENIOR etc.) necessitam um BASIC residente ou um disco que contém um programa capaz de digitar os mnemônicos de cada operação. Se porventura estiver na posse do programa LISA, poderá usá-lo normalmente para escrever as rotinas apresentadas.

Mesmo para os usuários que não têm nenhum programa disponível para escrever rotinas em linguagem de máquina, este impasse pode ser resolvido da seguinte maneira:

Cada operação corresponde a um OP-CODE, que é um valor hexadecimal específico para esta operação. Digite antes o seguinte programa em BASIC e – após digitar RUN – entre simplesmente com os OP-CODES correspondentes.

```
10  REM  PROGRAMAR COM OP-CODES
     HEXADECIMAIS
20 E = 768:E$ = "0300": HOME
30  PRINT "ENDERECO INICIAL = $";
     E$;" (";E;")"
40  PRINT : PRINT : PRINT "PARA M
     ODIFICAR O ENDERECO DIGITE '
     M'"
50  GET R$: IF R$ = "" THEN 50
60  IF R$ = "M" THEN 180
70  PRINT "ENTRE COM OS CODIGOS E
     M HEXADECIMAL. DIGITE 'FIM'
     PARA ENCERRAR.": INPUT A$
80  IF A$ = "FIM" THEN  END
90 A =  LEN (A$): IF A < 2 THEN 7
     0
100 W$ =  LEFT$ (A$,2): PRINT E; TAB(
     10);W$,
110 H =  ASC ( LEFT$ (W$,1)):L =
      ASC ( RIGHT$ (W$,1)):A$ =  MID$
     (A$,3,A)
120  IF H > 64 THEN H = (H - 55) *
     16: GOTO 140
130 H = (H - 48) * 16
140  IF L > 64 THEN T = H + L - 5
     5: GOTO 160
150 T = H + L - 48
160  POKE E,T:E = E + 1: PRINT T:
      IF A$ = "" THEN 70
170  GOTO 100
180  HOME : PRINT "ENTRE COM O NO
     VO ENDERECO EM HEXADECIMAL"
190  INPUT E$: IF  LEN (E$) <  >
     4 THEN 190
200 E = 0
210  FOR I = 1 TO 4:X =  ASC ( MID$
     (E$,I,1)): GOSUB 230: NEXT
220  GOTO 30
230  IF X > 64 THEN X = X - 55: GOTO
     250
```



```
240 X = X - 48
250 E = E + X * 16 ^ (4 - I): RETURN
```

O endereço inicial desta rotina para o nosso programa escrito em linguagem de máquina está fixado em $Ø3ØØ (= 768 em decimal). Se quiser, pode modificar este endereço digitando a letra "M". A seguir serão solicitados os códigos em hexadecimal, que podem ser digitados isoladamente ou em grupos.

Exemplo: digitar a seguinte rotina a partir do endereço $Ø32F:

```
Ø32F LDX #$3E
     BRK
```

Ao digitar "RUN", o nosso programa pergunta se o endereço inicial $Ø3ØØ deverá ser alterado ou não. Como no nosso caso o novo endereço inicial é diferente, apertamos a tecla "M" e fornecemos ao computador o novo endereço hexadecimal Ø32F, seguido de RETURN. Veremos que o endereço inicial foi alterado de Ø3ØØ para Ø32F.

Agora, ao digitar novamente a tecla RETURN, o programa solicita os códigos hexadecimais da nossa rotina em linguagem de máquina. Verificando no texto, ou no seu manual, encontramos para a instrução LDX #$ n o OPCODE A2. Este código exige ainda um complemento, no nosso caso o 3E (LDX #$3E). Já temos dois valores hexadecimais (A2 e 3E), definindo a instrução LDX #$3E. Resta ainda o OP-CODE para a instrução BRK, que vale ØØ. Digitemos portanto a seguinte seqüência no computador:

A23EØØ seguido de RETURN

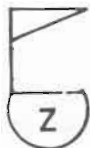
Como já terminamos, podemos digitar agora "FIM" para abordar o programa.

Para rodar agora a nossa rotina é só usar o comando direto CALL 815. Este valor decimal, equivalente ao valor hexadecimal Ø32F, pode ser encontrado na tela, na primeira coluna à esquerda. A coluna do meio corresponde aos nossos códigos hexadecimais digitados, e a coluna da direita a estes mesmos códigos em decimal.

Visualizando as flags (bandeiras)

A maioria das operações da CPU 65Ø2 afeta as bandeiras. No capítulo a seguir, usamos os seguintes símbolos para representar estas bandeiras:

Se um determinado flag for afetado, então este fato será representado graficamente da seguinte maneira:



aqui, o Z-flag será alterado

Às vezes, já sabemos de antemão o nível lógico de um determinado flag após a operação efetuada. Neste caso, o nível lógico (1 ou Ø) aparece graficamente desta maneira:

aqui, o N-flag assume o valor 1

Ao escrever-se rotinas em linguagem de máquina, principalmente quando estas se tornam extensas, perde-se com facilidade a visão geral da rotina. Isto ocorre com freqüência quando uma rotina escrita for analisada semanas ou até meses depois. Vamos a um exemplo típico desta "visão geral":

Suponha que achamos no meio das nossas anotações uma rotina como esta:


```
0300 JSR $FC58
     JSR $FBDD
     JSR $FBDD
     RTS
```

Pelo que podemos ver, são chamadas subrotinas, após as quais o programa retorna. Mas quais eram mesmo estas subrotinas? Qual era a finalidade desta rotina?

Uma das maneiras menos elegantes é a de se digitar toda a rotina para ver o que acontece. Com certeza este não é o método correto para descobrir o seu funcionamento. Como poderemos evitar, no futuro, estes dilemas?

Em primeiro lugar, vamos dar *sempre* um título a nossas rotinas. Mas, com apenas o título, ainda não sabemos o que acontece em $FC58 e $FBDD da rotina dada. Este impasse poderia ser resolvido da seguinte maneira:

```
0300 JSR $FC58  ; leva o cursor ao inicio da tela e limpa a tela
     JSR $FBDD  ; aciona o Bip
     JSR $FBDD  ; aciona o Bip
     RTS
```

Agora a nossa rotina tornou-se bastante clara. Sabemos exatamente o que acontece à medida que o programa está sendo executado.

Por outro lado, somos obrigados a mencionar, após cada JSR, o que acontece nesta subrotina. Será que existe uma maneira mais fácil?

A resposta é sim. Se nós sabemos que no endereço $FC58 é executada uma rotina equivalente ao comando HOME de BASIC, porque não substituir simplesmente o endereço $FC58 pela palavra "HOME"? Assim , como no endereço $FBDD é acionado o Bip, não parece lógico substituir $FBDD pela palavra "Bip".

Vamos ver como a nossa rotina fica agora:

```
0300 JSR HOME
     JSR Bip
     JSR Bip
     RTS
```

Ficou mais clara ainda, não ficou? Nesta altura, ainda é aconselhável anotar, no início do programa, os endereços equivalentes a todas etiquetas usadas. Como a rotina acima é curta, isto pode parecer até mais complicado, mas posso garantir que em rotinas compridas este procedimento é válido.

Assim sendo, a rotina com o uso de etiquetas apresenta-se assim:

```
        HOME = $FC58
        Bip  = $FBDD

   0300 JSR HOME
        JSR Bip
        JSR Bip
        RTS
```

Daqui em diante, a interpretação de todas as suas rotinas documentadas torna-se mais clara.

Vamos ver um outro exemplo:

```
0300 NOP
0301 LDA $0300
     JSR $FDDA ; imprime o conteúdo de A em hexadecimal
     JSR $FBDD ; aciona o Bip
     JSR $FF2D ; imprime "ERR" na tela
     RTS
```

Analisando esta rotina, podemos ver que qualquer valor decimal colocado no endereço $0300 será impresso na tela na forma hexadecimal. Depois disto soará um Bip e a seguir o computador imprime as letras ERR na tela. Como o uso das etiquetas não se restringe apenas aos endereços de subrotinas, vamos ver como esta rotina pode ser "melhorada":

```
VALOR  = $0300
HEXA   = $FDDA
Bip    = $FBDD
"ERR"  = $FF2D
INICIO= $0301
```



```
VALOR NOP
INICIO LDA VALOR
       JSR HEXA
       JSR Bip
       JSR "ERR"
       RTS
```

Aliás, também podemos usar esta técnica em BASIC. Suponhamos a seguinte subrotina a partir da linha 2000:

```
2000 CALL 768
2010 PRINT "NOVA PAGINA"
2020 CALL 802
2030 CALL 768
2040 GR : COLOR = 1
2050 HLIN 0,20 AT 15
2060 CALL 802
2070 RETURN
```

Sabendo que a partir do endereço 768 escrevemos uma rotina para a impressora, você seria capaz de saber, após algum tempo, o que exatamente acontece ao chamar as rotinas em 768, 802 e 782? Provavelmente não.

Poderíamos incluir no nosso programa umas linhas adicionais (de preferência no início do programa) conforme segue:

```
10 TEXT=768
20 GRAFICO=782
30 IMPRIMIR=802
 .
 .
 .
2000 CALL TEXT
2010 PRINT "NOVA PAGINA"
2020 CALL IMPRIMIR
2030 CALL GRAFICO
2040 GR: COLOR=1
2050 HLIN 0,20 AT 15
2060 CALL IMPRIMIR
2070 RETURN
```

Podemos ver que não há limites em usar etiquetas, desde que elas sejam claras e auto-explicativas.

EXERCÍCIOS

Assinale as afirmações corretas e complete onde for necessário:

( ) *Etiquetas facilitam a interpretação de uma rotina Assembly.*
( ) Somente endereços *podem ser substituídos por etiquetas.*
( ) *Em BASIC o uso de etiquetas é permitido.*

É aconselhável usar . . . . *um título para nossas rotinas.*
Ao usar etiquetas, é preciso . . . . *estas etiquetas antes.*

capítulo 19

fluxogramas

Você já tentou alguma vez elaborar e escrever uma rotina extensa em linguagem de máquina? Se a resposta for sim, então provavelmente já sentiu o drama em manter uma visão geral do seu programa. Quantas vezes você já pegou uma folha com um trecho do seu programa sem saber onde "encaixá-lo"? Ou ainda, escrevendo as rotinas, nunca sentiu falta de uma instrução ou rotina esquecida? Normalmente a sua resposta é afirmativa. Mesmo utilizado etiquetas, contribuindo para uma clareza maior, ainda você sente a falta de um "resumo geral" do seu programa.

Este "resumo geral" é o objetivo deste capítulo. Veremos como reunir qualquer programa num *Fluxograma* antes da elaboração das rotinas em si.

O que é um fluxograma

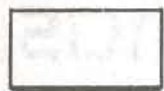
Podemos dizer que um fluxograma é uma representação semi-gráfica do funcionamento do nosso programa. Cada conjunto de operações isoladas, bem como todas as rotinas envolvendo decisões, são indicados separadamente por símbolos gráficos.

Estes símbolos gráficos usados em fluxogramas têm formatos geométricos diferentes, onde cada formato indica se temos na nossa frente um

processamento isolado ou uma decisão a tomar. Para o nosso uso, temos à nossa disposição os seguintes formatos para montar fluxogramas:

Em primeiro lugar vamos analisar os formatos mais usados, que na maioria dos fluxogramas para o nosso uso são suficientes para representar qualquer programa:

início ou fim de um programa



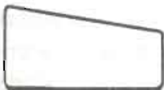
bloco de processamento isolado

bloco de decisão

Temos ainda setas/flechas interligando estes blocos, indicando o caminho a seguir pelo programa.

Fica a critério de cada programador usar também formatos adicionais como:

continuação
(por exemplo na página seguinte)



bloco de leitura (ex.: leitura do teclado)

bloco de escrita
(ex.: saída para impressora)

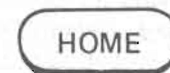
Não existem "leis" quanto ao uso dos formatos acima. Basta adotar alguns deles e manter o critério aplicado para todos os demais fluxogramas desenvolvidos por você.

Vamos a um exemplo prático:

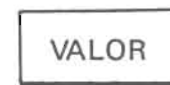
Queremos desenvolver uma rotina que imprima todos os valores hexadecimais entre $ØØ e $FF na tela.

Solução: antes de escrever a rotina em si, vamos elaborar o respectivo fluxograma:

Passo 1 – antes de iniciar a impressão dos valores hexadecimais, convém limpar a tela executando um "HOME". Como esta rotina inicia o nosso programa, o bloco será o seguinte:



Passo 2 – usaremos um contador para o nosso programa, que será incrementado de Ø até 255 que por sua vez fornece a base dos valores hexadecimais a serem impressos. Para fixar este valor inicial, usamos o seguinte símbolo gráfico:



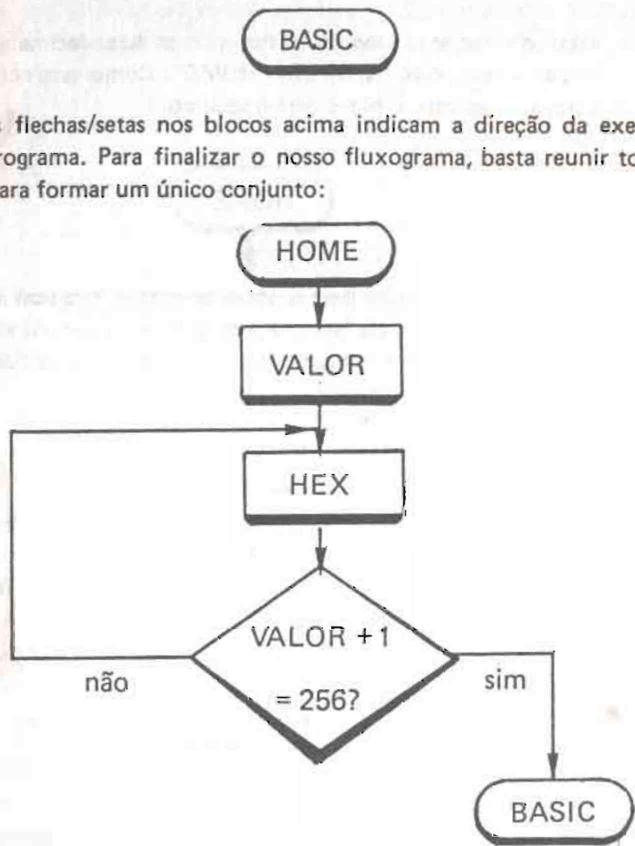
Passo 3 – com a tela limpa e o valor inicial pronto, podemos imprimir o mesmo valor na forma hexadecimal:

Passo 4 – tendo o valor impresso na tela, basta incrementá-lo e verificar se este valor já ultrapassou o valor máximo 255. Neste caso, temos uma decisão a tomar, pois se o valor 255 for ultrapassado, o programa deve retornar ao BASIC. Caso contrário, deverá ser impresso o próximo número:

Passo 5 — e, no final, temos o nosso retorno ao BASIC:

BASIC

As flechas/setas nos blocos acima indicam a direção da execução do nosso programa. Para finalizar o nosso fluxograma, basta reunir todos estes blocos para formar um único conjunto:



Analisando agora este fluxograma, já podemos ter uma idéia bastante clara do que o nosso programa faz. Observe que até agora nós não nos preocupamos em como cada bloco realiza a operação indicada, pois fixamos por ora, somente a seqüência lógica das operações.

Nesta altura, já temos condições em descobrir eventuais falhas de lógica ou ainda a possível falta de rotinas adicionais.

Uma vez pronto o fluxograma, podemos começar a nos preocupar com as rotinas individuais:

HOME

Já temos à nossa disposição uma subrotina no sistema operacional, realizando a operação equivalente ao comando HOME do BASIC. Sendo assim, colocamos aqui simplesmente:

```
JSR $FC58 ; chama a subrotina "HOME"
```

VALOR

Para manter um contador, necessitamos uma variável ou um endereço na memória para manipular este contador. Escolhemos o registrador X:

```
LDX #$ØØ ; contador (X) = Ø
```

HEX

Temos no sistema operacional uma rotina no endereço $FDDA que imprime um número em hexadecimal. Este valor a ser impresso deve estar no acumulador. Transferimos portanto o valor de X para o A:

```
TXA         ; transfere o contador para o A
JSR $FDDA   ; imprime este valor em hexa-
              decimal
```

VALOR + 1
= 256?

Vamos incrementar o contador e testar se já foi atingida a contagem 256. Neste caso, o contador terá o valor Ø:

```
INX         ; soma 1 ao contador
BNE $HEX    ; volta a rotina HEX, se for
              256
```

BASIC

E, finalizando o nosso programa, devemos retornar ao BASIC mediante a seguinte instrução:

```
RTS    ;    retorna ao BASIC
```

A nossa rotina, em conjunto, apresenta-se portanto da seguinte maneira:

```
HOME    JSR $FC58
VALOR   LDX #$00
HEX     TXA
        JSR $FDDA
TEST    INX
        BNE HENX
BASIC   RTS
```

Resta ainda definir o endereço inicial da nossa rotina, e com isto o endereço "HEX" usado na instrução envolvendo o desvio adicional (BNE).

EXERCÍCIOS

Assinale a alternativa correta e complete onde for necessário:

( ) *O conjunto das instruções de uma rotina é chamado de fluxograma.*
( ) *Um fluxograma é o resumo lógico de um programa.*
( ) *Diversos símbolos gráficos formam um fluxograma.*
( ) *Somente decisões têm um formato próprio dentro dos fluxogramas.*
( ) *Para elaborar um fluxograma, é preciso utilizar todos os símbolos existentes.*

Para indicar a seqüência lógica de um programa, usamos . . . .
Através do formato . . . . podemos reconhecer as rotinas no diagrama.
Um símbolo oval indica sempre o . . . . e o . . . . de uma rotina.
O fluxograma deve ser elaborado . . . . das rotinas individuais.
O formato quadrado . . . . indica uma rotina que toma decisão.

apêndice A

Sistemas de Numeração

Estamos acostumados, desde a infância, a operar com valores decimais, portanto, bastante familiarizados em somar, subtrair, dividir e multiplicar valores decimais como 23, 1123, 19 etc. Definindo o sistema decimal matematicamente, temos o seguinte critério:

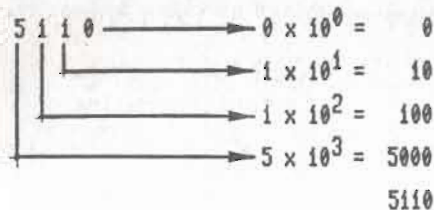
- a primeira casa à direita vale o próprio valor (Ex. 3 vale realmente o valor 3)
- a segunda casa à direita vale dez vezes o seu número (Ex. o número 7 na segunda casa vale 7 x 10 = 70)

ou ainda, estabelecendo uma fórmula genérica:

valor da casa n+1 = valor do dígito nela contido x 10^n

Atenção: daqui em diante, o valor n sempre inicia com 0. Assim, para definir, por exemplo, a terceira casa de um valor qualquer, n será 2.

Exemplo para o sistema decimal:



Sistema binário

O computador não tem condições de realizar cálculos e manipular valores decimais. Na verdade, no "interior" do computador são reconhecidos apenas dois estados:

Estado 0 = falso
Estado **1** = verdadeiro

Representando estas duas possibilidades com um interruptor (Fig. A-1), podemos dizer que o interruptor aberto representa um estado 0 (falso) e o fechado o estado 1 (verdadeiro).

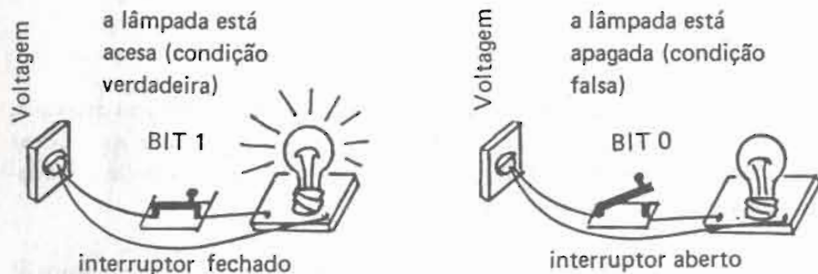


Figura A.1 — Representação dos estados dos bits.

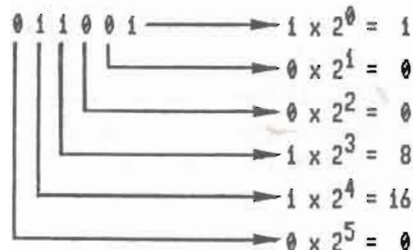
Cada estado é denominado BIT, onde cada BIT pode ser apenas 0 ou 1. Utilizando BITS para representar valores decimais, podemos indicar com um único BIT somente os valores 0 e 1 decimais. A partir do número decimal **2** já é preciso o uso de um BIT adicional. Com dois BITS então somos capazes de representar os valores decimais de 0 a 3, e a partir do valor 4 necessitamos novamente um BIT adicional:

Valor decimal	Valor binário	Valor decimal	Valor binário
0	0	5	101
1	1	6	110
2	10	7	111
3	11	8	1000
4	100	9	1001 ...etc

Matematicamente, cada casa binária vale portanto:

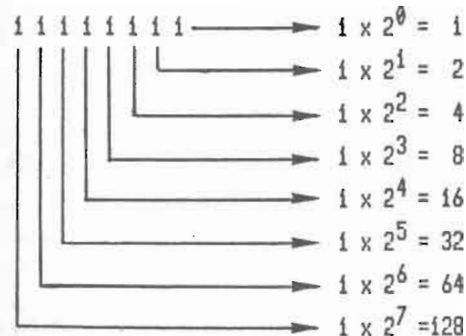
valor da casa n + 1 = dígito x 2^n

Vamos transformar o valor binário 011001 para o respectivo valor decimal:



Geralmente, como é o caso da CPU 6502, são manipulados dados formados por 8 BITS. Um dado ou uma palavra de 8 BITS é chamado de um BYTE. Assim, a cada vez, quando falamos de um BYTE, subentende-se que um BYTE é formado por 8 BITS.

Qual é o maior valor decimal possível, que um BYTE pode representar? É só colocar todos os bits para o nível 1, e teremos



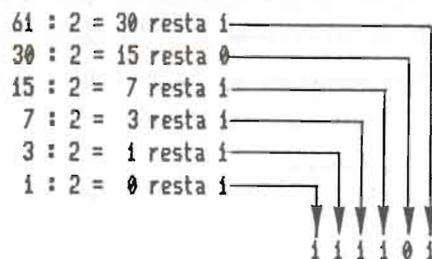
Isto nos dá 256 combinações, pois temos que levar em conta também o valor 0 como uma destas combinações possíveis. Não ficou claro de repente, porque o POKE do BASIC não permite números maiores que 255 como dados?

Para endereçar as memórias, a CPU 6502 utiliza dois BYTES, que correspondem a um total de 16 BITS. Tente descobrir o maior endereço possível que a CPU pode endereçar. Se a sua conta for correta, achará o valor 65535. Confere?

Já sabemos como transformar um valor binário em seu respectivo valor decimal. Faremos agora o contrário:

Ex.: Transformar 61 para a configuração binária.

Solução:



O que realmente fizemos foi dividir constantemente o valor decimal por 2 até que o resultado se torne 0; o BIT correspondente também será 0, e se o resto é 1, o respectivo BIT também será 1.

Tente agora transformar o valor 254 para o sistema binário.

Como já foi dito anteriormente, a CPU manipula dados de 1 BYTE e endereços de 2 BYTES, onde cada BYTE é formado por 8 BITS. Convém escrever *sempre* todos os 8 BITS de cada BYTE, mesmo quando não forem necessários.

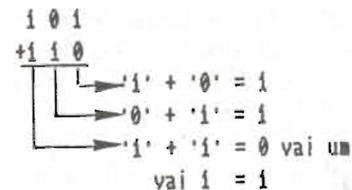
Ex.: o valor 7 decimal, na configuração binária, vale 111. Todavia, a regra é mencionar também os 5 BITS restantes: 00000111

Operações com valores binários

Pode parecer no início até um pouco estranho, mas tenho certeza que com um pouco de treino você facilmente se adaptará ao sistema binário. Para realizar por exemplo uma soma binária, somamos simplesmente todos os BITS da mesma coluna. Se o resultado for par, o resulado nesta coluna será 0. Se for ímpar, o resultado será 1. E o resto da soma deverá ser somado com os BITS da coluna seguinte.

Ex.: Somar os valores binários 101 e 110.

Solução:

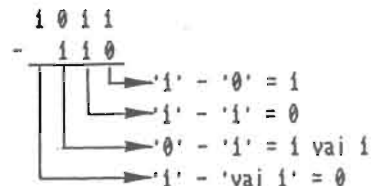


Portanto, o resultado será 1011. Passe os valores para o sistema decimal e faça a prova.

Semelhante à adição, a subtração não oferece problemas.

Ex.: Subtrair o valor binário 110 do valor binário 1011.

Solução:



Resultado: 101. Confira aqui também o resultado em decimal.

Um aviso aos preocupados:

A programação em linguagem de máquina não exige um conhecimento profundo do sistema binário. Basta saber como transformar esta configuração para o sistema decimal ou hexadecimal (vide o assunto a seguir) e vice-versa. Em apenas alguns casos esporádicos, utilizamos a configuração binária de um dado, e não faz mal recapitular este assunto nestas páginas.

Sistema hexadecimal

Falando em programa de microcomputadores em linguagem de máquina, o sistema numérico mais usado, sem dúvida, é o hexadecimal. Como vere-

mos mais adiante, este tipo de notação oferece uma visualização muito mais eficaz do que a forma binária, ou até que a forma decimal.

No sistema decimal, a base de cálculo é o valor 10, e no sistema binário, o valor 2. Neste caso, como já o próprio nome do sistema revela, trabalharemos na base 16. Agora, cada casa pode representar até 16 combinações diferentes, antes de empregar uma casa adicional. A questão é como é que podemos indicar 16 combinações para uma casa, se nós temos apenas 10 algarismos à nossa disposição (de 0 a 9)?

Este impasse é resolvido com o emprego de letras do nosso alfabeto, onde a partir do valor 10 utilizaremos o A, para o valor 11 o B, e assim por diante até o valor 15, representado por um F:

Valor decimal	Valor hexadecimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

etc ...

Continuando esta seqüência, o valor decimal 16 corresponde ao valor 10 em hexadecimal. Não parece engraçado este sistema? Se eu disser que tenho 1F anos de idade, você saberia dizer a minha idade em decimal?

Percebemos que facilmente podemos nos confundir, principalmente quando o valor hexadecimal não contém nenhuma letra. Por exemplo, se um determinado valor é 23, como posso saber se é 23 em decimal ou 23 em hexadecimal? Pior ainda, com um valor como 10, que poderia valer:

10 em decimal, ou
10 em hexadecimal (= 16 em decimal), ou ainda
10 em binário (= 2 em decimal)

É óbvi a necessidade de criar características para cada sistema empregado. Geralmente, todos os valores binários devem ser escritos com pelo menos 8 BITS, o que elimina desde o início o perigo de confundí-los com valores decimais ou hexadecimais (não operamos com valores tão altos). Resta ainda diferenciar os valores hexadecimais dos valores decimais.

Como ao escrever um programa em linguagem de máquina normalmente empregamos valores hexadecimais, deixaremos estes valores da maneira como são escritas. Se porventura houver necessidade de indicar um valor decimal, podemos optar por uma das seguintes possibilidades:

Ex.: o valor 23 decimal pode ser escrito como:
+23 ou 23d, onde o 'd' indica que se trata de um valor decimal.

Já descobriu como calcular o valor de cada casa hexadecimal? É só manter o mesmo princípio como usamos nos sistemas decimais e binários:

valor da casa n+1 = dígito × 16^n

Utilizando esta fórmula, vamos calcular o valor decimal para o valor 0B1F hexadecimal:

Solução:

```
0 B 1 F ----------> 15 x 16^0 =   15
| | |
| | +------------>  1 x 16^1 =   16
| |
| +-------------->  11 x 16^2 = 2816
|
+---------------->  0 x 16^3 =    0
                              ------
                               2847
```

Foi dito que o sistema hexadecimal oferece uma visualização muito maior do que os outros sistemas já conhecidos e veremos agora o porquê:

Cada BYTE tem 8 BITS. Se nós dividirmos um BYTE por 2, teremos dois grupos de 4 BITS cada, certo? Até agora, nenhuma novidade. Mas com 4 BITS eu não consigo representar valores decimais entre 0 e 15? (Ex-

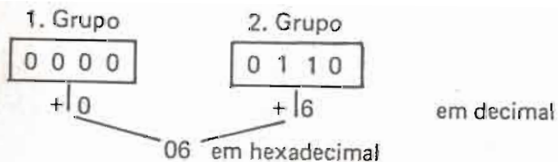
perimente, para ver se isto é verdade). Ora, a faixa de 0 até 15 não lembra o nosso sistema hexadecimal?

Pois é, com apenas uma casa hexadecimal somos capazes de indicar qualquer valor formado por 4 BITS, e com 2 casas hexadecimais o valor de qualquer BYTE! Desta maneira, podemos reunir qualquer configuração de 8 BITS, ou seja, qualquer valor decimal entre 0 e +255 com apenas duas casas hexadecimais. E é justamente isto o que faremos daqui por diante. Nas rotinas apresentadas, cada dado será formado por 2 casas hexadecimais, e cada endereço (2 BYTES) por 4 casas hexadecimais. Como no sistema binário, vale também aqui o critério de manter *sempre* a quantidade certa das casas hexadecimais, mesmo quando não houver necessidade.

Ex.: o dado E (= +14) deve ser escrito como 0E

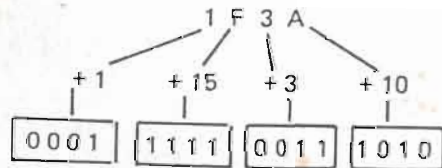
Ex.: Transformar o valor binário (= dado) 00000110 para hexadecimal.

Solução:



Ex.: transformar o endereço 1F3A para a configuração binária:

Solução:



A seguir, uma tabela com os valores decimais, hexadecimais e binários de 0 a +255.

Tabela decimal — hexadecimal — binário

dec.	hex.	binário	dec.	hex.	binário
0	*00*	00000000	35	*23*	00100011
1	*01*	00000001	36	*24*	00100100
2	*02*	00000010	37	*25*	00100101
3	*03*	00000011	38	*26*	00100110
4	*04*	00000100	39	*27*	00100111
5	*05*	00000101	40	*28*	00101000
6	*06*	00000110	41	*29*	00101001
7	*07*	00000111	42	*2A*	00101010
8	*08*	00001000	43	*2B*	00101011
9	*09*	00001001	44	*2C*	00101100
10	*0A*	00001010	45	*2D*	00101101
11	*0B*	00001011	46	*2E*	00101110
12	*0C*	00001100	47	*2F*	00101111
13	*0D*	00001101	48	*30*	00110000
14	*0E*	00001110	49	*31*	00110001
15	*0F*	00001111	50	*32*	00110010
16	*10*	00010000	51	*33*	00110011
17	*11*	00010001	52	*34*	00110100
18	*12*	00010010	53	*35*	00110101
19	*13*	00010011	54	*36*	00110110
20	*14*	00010100	55	*37*	00110111
21	*15*	00010101	56	*38*	00111000
22	*16*	00010110	57	*39*	00111001
23	*17*	00010111	58	*3A*	00111010
24	*18*	00011000	59	*3B*	00111011
25	*19*	00011001	60	*3C*	00111100
26	*1A*	00011010	61	*3D*	00111101
27	*1B*	00011011	62	*3E*	00111110
28	*1C*	00011100	63	*3F*	00111111
29	*1D*	00011101	64	*40*	01000000
30	*1E*	00011110	65	*41*	01000001
31	*1F*	00011111	66	*42*	01000010
32	*20*	00100000	67	*43*	01000011
33	*21*	00100001	68	*44*	01000100
34	*22*	00100010	69	*45*	01000101

dec.	hex.	binário	dec.	hex.	binário
70	*46*	01000110	107	*6B*	01101011
71	*47*	01000111	108	*6C*	01101100
72	*48*	01001000	109	*6D*	01101101
73	*49*	01001001	110	*6E*	01101110
74	*4A*	01001010	111	*6F*	01101111
75	*4B*	01001011	112	*70*	01110000
76	*4C*	01001100	113	*71*	01110001
77	*4D*	01001101	114	*72*	01110010
78	*4E*	01001110	115	*73*	01110011
79	*4F*	01001111	116	*74*	01110100
80	*50*	01010000	117	*75*	01110101
81	*51*	01010001	118	*76*	01110110
82	*52*	01010010	119	*77*	01110111
83	*53*	01010011	120	*78*	01111000
84	*54*	01010100	121	*79*	01111001
85	*55*	01010101	122	*7A*	01111010
86	*56*	01010110	123	*7B*	01111011
87	*57*	01010111	124	*7C*	01111100
88	*58*	01011000	125	*7D*	01111101
89	*59*	01011001	126	*7E*	01111110
90	*5A*	01011010	127	*7F*	01111111
91	*5B*	01011011	128	*80*	10000000
92	*5C*	01011100	129	*81*	10000001
93	*5D*	01011101	130	*82*	10000010
94	*5E*	01011110	131	*83*	10000011
95	*5F*	01011111	132	*84*	10000100
96	*60*	01100000	133	*85*	10000101
97	*61*	01100001	134	*86*	10000110
98	*62*	01100010	135	*87*	10000111
99	*63*	01100011	136	*88*	10001000
100	*64*	01100100	137	*89*	10001001
101	*65*	01100101	138	*8A*	10001010
102	*66*	01100110	139	*8B*	10001011
103	*67*	01100111	140	*8C*	10001100
104	*68*	01101000	141	*8D*	10001101
105	*69*	01101001	142	*8E*	10001110
106	*6A*	01101010	143	*8F*	10001111

dec.	hex.	binário	dec.	hex.	binário
144	*90*	10010000	181	*B5*	10110101
145	*91*	10010001	182	*B6*	10110110
146	*92*	10010010	183	*B7*	10110111
147	*93*	10010011	184	*B8*	10111000
148	*94*	10010100	185	*B9*	10111001
149	*95*	10010101	186	*BA*	10111010
150	*96*	10010110	187	*BB*	10111011
151	*97*	10010111	188	*BC*	10111100
152	*98*	10011000	189	*BD*	10111101
153	*99*	10011001	190	*BE*	10111110
154	*9A*	10011010	191	*BF*	10111111
155	*9B*	10011011	192	*C0*	11000000
156	*9C*	10011100	193	*C1*	11000001
157	*9D*	10011101	194	*C2*	11000010
158	*9E*	10011110	195	*C3*	11000011
159	*9F*	10011111	196	*C4*	11000100
160	*A0*	10100000	197	*C5*	11000101
161	*A1*	10100001	198	*C6*	11000110
162	*A2*	10100010	199	*C7*	11000111
163	*A3*	10100011	200	*C8*	11001000
164	*A4*	10100100	201	*C9*	11001001
165	*A5*	10100101	202	*CA*	11001010
166	*A6*	10100110	203	*CB*	11001011
167	*A7*	10100111	204	*CC*	11001100
168	*A8*	10101000	205	*CD*	11001101
169	*A9*	10101001	206	*CE*	11001110
170	*AA*	10101010	207	*CF*	11001111
171	*AB*	10101011	208	*D0*	11010000
172	*AC*	10101100	209	*D1*	11010001
173	*AD*	10101101	210	*D2*	11010010
174	*AE*	10101110	211	*D3*	11010011
175	*AF*	10101111	212	*D4*	11010100
176	*B0*	10110000	213	*D5*	11010101
177	*B1*	10110001	214	*D6*	11010110
178	*B2*	10110010	215	*D7*	11010111
179	*B3*	10110011	216	*D8*	11011000
180	*B4*	10110100	217	*D9*	11011001

218	*DA*	11011010
219	*DB*	11011011
220	*DC*	11011100
221	*DD*	11011101
222	*DE*	11011110
223	*DF*	11011111
224	*E0*	11100000
225	*E1*	11100001
226	*E2*	11100010
227	*E3*	11100011
228	*E4*	11100100
229	*E5*	11100101
230	*E6*	11100110
231	*E7*	11100111
232	*E8*	11101000
233	*E9*	11101001
234	*EA*	11101010
235	*EB*	11101011
236	*EC*	11101100
237	*ED*	11101101
238	*EE*	11101110
239	*EF*	11101111
240	*F0*	11110000
241	*F1*	11110001
242	*F2*	11110010
243	*F3*	11110011
244	*F4*	11110100
245	*F5*	11110101
246	*F6*	11110110
247	*F7*	11110111
248	*F8*	11111000
249	*F9*	11111001
250	*FA*	11111010
251	*FB*	11111011
252	*FC*	11111100
253	*FD*	11111101
254	*FE*	11111110
255	*FF*	11111111

Assinale a afirmação correta e complete onde for necessário:

( ) *O sistema binário tem apenas 1 combinação por casa*
( ) *Cada BIT no sistema binário é chamado BYTE*
( ) *O maior valor decimal obtido com 8 BITS é 255*
( ) *A CPU 6502 manipula endereços formados por 2 BYTES*
( ) No sistema hexadecimal *usamos apenas números*
( ) *Um dado é formado por duas casas hexadecimais*
( ) *Quatro casas hexadecimais sempre indicam um endereço*

O computador opera no sistema . . . .
Para representar o valor decimal 8 necessitamos . . . . BITS.
O maior endereço possível, formado por 16 BITS, é. . . . .
8 BIST formam um . . . .
No sistema hexadecimal, a letra . . . . . . corresponde ao valor 11 em decimal.
Para distinguir um valor decimal, usamos o . . . .
. . . . BITS formam uma cas hexadecimal.
Uma condição verdadeira corresponde a um bit com nível lógico . . . .

apêndice B

O seu conhecimento da estrutura e das instruções da CPU Z80 certamente irá ajudá-lo a entender as funções da CPU 6502, apresentada neste livro. Mas — para iniciar — onde está a principal diferença entre a Z80 e a 6502?

Esta pergunta não pode ser respondida de maneira simples. Começaremos pelos registradores. Se eu disser que a 6502 tem apenas um acumulador e mais dois registradores de *index*, provavelmente você vai pensar numa inferioridade óbvia do 6502 em relação ao Z80. Afinal, este último apresenta nada mais e nada menos que 7 registradores, sem falarmos dos pares IX e IY!

Tem mais: a CPU 6502 não é capaz de formar pares, isto é, não consegue manipular endereços . . . Quer outra? A 6502 não transfere blocos de dados, não tem modos de interrupção diversificados, não comanda unidades de I/O com as instruções IN e OUT ou semelhantes . . .

Chegando até aqui, resta perguntar: o que faz a 6502 realmente??? Parece um milagre que computadores considerados "bons", como a linha APPLE, COMMODORE e outros utilizam a CPU 6502. Onde afinal se encontra esta vantagem da 6502, não encontrada na Z80?

Nada melhor do que um exemplo simples. Digamos que queremos incrementar o valor contido no endereço 003F. Utilizando a CPU Z80, você resolveria esta questão assim:

```
LD A,(003F)
INC A
LD (003F), A
```

ou assim:

```
LD HL,003F
INC (HL)
```

Utilizando a primeira solução necessita-se um total de 7 bytes para obter o incremento desejado, considerando que o valor inicial em A não importa. Na solução seguinte, utilizando o par HL, a quantidade necessária é de 4 Bytes. Em ambos os casos, foi preciso o emprego de um ou mais registradores, pois a CPU Z80 não oferece instruções tais como INC (003F). Mesmo existindo, esta última instrução teórica ainda ocuparia 3 bytes.

Veremos agora como a CPU 6502 resolve a mesma questão:

```
INC $3F
```

É isso mesmo, com apenas uma unica instrução de 2 bytes, sem o envolvimento de qualquer registrador, temos a solução na palma da mão! Não só o fato de não usar registradores, mas sim a manipulação *direta* com a memória ocasiona, além de uma simplificação, uma velocidade de execução muito maior do que com uma rotina semelhante para a CPU Z80.

O que vimos neste pequeno exemplo é a manipulação da página zero, a principal característica da CPU 6502. E justamente esta característica faz a 6502 tão poderosa, como você verá ao estudar as instruções desta CPU.

Por falar em instruções da CPU, você está lembrado da quantidade de todas as instruções da CPU Z80? Contando todas as alternativas, atingimos o valor 693. E a CPU 6502? – Apenas 151 combinações, baseadas em 56 instruções básicas.

É claro que existem instruções na Z80 muito poderosas, que realmente fazem falta na CPU 6502. Mas temos diante de nós uma CPU com uma filosofia totalmente diferente a da família do Z80, motivo pelo qual é muito difícil comparar as vantagens e desvantagens de cada CPU.

O que não muda é a estrutura básica do funcionamento. Isto é, também a CPU 6502 utiliza um programa counter (contador de programa), um stack pointer e as inevitáveis flags para tomar decisões. Ah, a CPU 6502 tem não só 6 flags aproveitáveis como a CPU Z80, mas sim 7 flags! Mas não se preocupe, tenho certeza que você não encontrará dificuldades em dominar também a nova filosofia da CPU 6502.

apêndice C

Tabela 1 – Códigos das operações

00	BRK	12	–	24	BIT pág. 0
01	ORA (Ind.,X)	13	–	25	AND pág. 0
02	–	14	–	26	ROL pág. 0
03	–	15	ORA pág. 0,X	27	–
04	–	16	ASL pág. 0,X	28	PLP
05	ORA pág. 0	17	–	29	AND imed.
06	ASL pág. 0	18	CLC	2A	ROL acum.
07	–	19	ORA abs.,Y	2B	–
08	PHP	1A	–	2C	BIT abs.
09	ORA imed.	1B	–	2D	AND abs.
0A	ASL acum.	1C	–	2E	ROL abs.
0B	–	1D	ORA abs.X	2F	–
0C	–	1E	ASL abs.X	30	BMI
0D	ORA abs.	1F	–	31	AND (ind.),Y
0E	ASL abs.	20	JRS	32	–
0F	–	21	AND (ind.,X)	33	–
10	BPL	22	–	34	–
11	ORA (Ind.),Y	23	–	35	AND pág. 0,X

36	ROL pág. 0,X
37	–
38	SEC
39	AND abs.,Y
3A	–
3B	–
3C	–
3D	AND abs.,X
3E	ROL abs.,X
3F	–
40	RTI
41	EOR (ind.,X)
42	–
43	–
44	–
45	EOR pág. 0
46	LSR pág. 0
47	–
48	PHA
49	EOR imed.
4A	LSR acum.
4B	–
4C	JMP abs.
4D	EOR abs.
4E	LSR abs.
4F	–
50	BVC
51	EOR (ind.),Y
52	–
53	–
54	–
55	EOR pág. 0,X
56	LSR pág. 0,X
57	–
58	CLI
59	EOR abs.,Y
5A	–
5B	–
5C	–

5D	EOR abs.,X
5E	LSR abs.,X
5F	–
60	RTS
61	ADC (ind.,X)
62	–
63	–
64	–
65	ADC pág. 0
66	ROR pág. 0
67	–
68	PLA
69	ADC imed.
6A	ROR acum.
6B	–
6C	JMP ind.
6D	ADC abs.
6E	ROR abs.
6F	–
70	BVS
71	ADC (ind.),Y
72	–
73	–
74	–
75	ADC pág. 0,X
76	ROR pág. 0,X
77	–
78	SEI
79	ADC abs.,Y
7A	–
7B	–
7C	–
7D	ADC abs.,X
7E	ROR abs.,X
7F	–
80	–
81	STA (ind.,X)
82	–
83	–

84	STY pág. 0
85	STA pág. 0
86	STX pág. 0
87	–
88	DEY
89	–
8A	TXA
8B	–
8C	STY abs.
8D	STA abs.
8E	STX abs.
8F	–
90	BCC
91	STA (ind.),Y
92	–
93	–
94	STY pág. 0,X
95	STA pág. 0,X
96	STX pág. 0,Y
97	–
98	TYA
99	STA abs.,Y
9A	TXS
9B	–
9C	–
9D	STA abs.,X
9E	–
9F	–
A0	LDY imed.
A1	LDA (ind.,X)
A2	LDX imed.
A3	–
A4	LDY pág. 0
A5	LDA pág. 0
A6	LDX pág. 0
A7	–
A8	TAY
A9	LDA imed.
AA	TAX

AB	–
AC	LDY abs.
AD	LDA abs.
AE	LDX abs.
AF	–
B0	BCS
B1	LDA (ind.),Y
B2	–
B3	–
B4	LDY pág. 0,X
B5	LDA pág. 0,X
B6	LDX pág. 0,Y
B7	–
B8	CLV
B9	LDA abs.,Y
BA	TSX
BB	–
BC	LDY abs.,X
BD	LDA abs.,X
BE	LDX abs.,Y
BF	–
C0	CPY imed.
C1	CMP (ind.,X)
C2	–
C3	–
C4	CPY pág. 0
C5	CMP pág. 0
C6	DEC pág. 0
C7	–

C8	INY
C9	CMP imed.
CA	DEX
CB	–
CC	CPY abs.
CD	CMP abs.
CE	DEC abs.
CF	–
D0	BNE
D1	CMP (ind.),Y
D2	–
D3	–
D4	–
D5	CMP pág. 0,X
D6	DEC pág. 0,X
D7	–
D8	CLD
D9	CMP abs.,Y
DA	–
DB	–
DC	–
DD	CMP abs.,X
DE	DEC abs.,X
DF	–
E0	CPX imed.
E1	SBC (ind.,X)
E2	–
E3	–
E4	CPX pág. 0

E5	SBC pág. 0
E6	INC pág. 0
E7	–
E8	INX
E9	SBC imed.
EA	NOP
EB	–
EC	CPX abs.
ED	SBC abs.
EE	INC abs.
EF	–
F0	BEQ
F1	SBC (ind.),Y
F2	–
F3	–
F4	–
F5	SBC pág. 0,X
F6	INC pág. 0,X
F7	–
F8	SED
F9	SBC abs.,Y
FA	–
FB	–
FC	–
FD	SBC abs.,X
FE	INC abs.,X
FF	–

Os códigos indicados com um traço (Ex. 9E –) são previstos para futuras expansões da CPU 6502, e por ora sem função. Para transformar os códigos acima (na forma hexadecimal), utilize a tabela de conversão apresentada no apêndice A.

Abreviações usadas na tabela acima:

ind. – indireto
pág. 0 – página zero
imed. – imediato
acum. – acumulador
abs. – absoluto

As instruções da CPU 6502 em ordem alfabética:

ADC – some a memória ao acumulador com Carry
AND – operação AND entre a memória e o acumulador
ASL – desloca os bits uma casa para a esquerda (memória ou acumulador)
BCC – desvia se Carry = 0
BCS – desvia se Carry = 1
BEQ – desvia se o resultado é zero
BIT – testa os bits do acumulador ou da memória
BMI – desvia se o resultado é negativo
BNE – desvia se o resultado não é zero
BPL – desvia se o resultado é positivo
BRK – parada forçada
BVC – desvia se Overflow = 0
BVS – desvia se Overflow = 1
CLC – reseta o Carry (Carry = 0)
CLD – reseta o modo decimal
CLI – reseta o bit de interrupção
CLV – reseta o Overflow (Overflow = 0)
CMP – compara o acumulador com a memória/dado
CPX – compara a memória com o índex X
CPY – compara a memória com o índex Y
DEC – decrementa a memória pelo valor 1
DEX – decrementa o índex X pelo valor 1
DEY – decrementa o índex Y pelo valor 1
EOR – operação OR EXCLUSIVO entre a memória e o acumulador
INC – incrementa a memória pelo valor 1
INX – incrementa o índex X pelo valor 1
INY – incrementa o índex Y pelo valor 1
JMP – desvia para uma nova posição
JSR – desvia para uma nova posição e armazena o endereço de retorno
LDA – carrega o acumulador com a memória/dado
LDX – carrega o índex X com a memória/dado
LDY – carrega o índex Y com a memória/dado
LSR – desloca os bits uma casa para a direita (memória ou acumulador)
NOP – nenhuma operação
ORA – operação OR entre o acumulador e a memória/dado
PHA – coloca o valor do acumulador no stack (pilha)
PHP – coloca o valor do status register no stack (pilha)

PLA – retira o valor do stack (pilha) e coloca-o no acumulador
PLP – retira o valor do stack (pilha) e coloca-o no status register
ROL – roda os bits uma casa para esquerda (memória ou acumulador)
ROR – roda os bits uma casa para direita (memória ou acumulador)
RTI – retorna de uma rotina de interrupção
RTS – retorna de uma sub-rotina
SBC – subtrai do acumulador uma memória/dado com Borrow
SEC – seta o Carry (Carry = 1)
SED – seta o modo decimal
SEI – seta o status de interrupção
STA – coloca o valor do acumulador na memória
STX – coloca o valor do índex X na memória
STY – coloca o valor do índex Y na memória
TAX – transfere o valor do acumulador para o índex X
TAY – transfere o valor do acumulador para o índex Y
TSX – transfere o valor do Stack-pointer para o índex X
TXA – transfere o valor do índex X para o acumulador
TXS – transfere o valor do índex X para o Stack-pointer
TYA – transfere o valor do índex Y para o acumulador

Resumo das instruções da CPU 6502

No resumo a seguir, os seguintes símbolos são usados para explicar o funcionamento de cada instrução:

A	acumulador
X,Y	índex registradores
M	memória ou dado
P	status-register do processador
S	stack-pointer
X	muda de estado
–	não muda de estado
+	adição
$\wedge$	operação lógica AND
-	subtração
$\forall$	operação lógica OR EXCLUSIVO
$\downarrow$	retira do stack
$\uparrow$	coloca no stack

→	coloca em
←	retira de
V	operação lógica OR
PC	program counter (contador do programa)
PCH	byte mais significativo do program counter
PCL	byte menos significativo do program counter
OPER	operante
#	modo de endereçamento imediato

ADC operação: A + M + C → A,C

N Z C I D V
X X X – – X

imediato	ADC	# OPER	= 69
página zero	ADC	OPER	= 65
página zero,X	ADC	OPER,X	= 75
absoluto	ADC	OPER	= 6D
absoluto,X	ADC	OPER,X	= 7D
absoluto,Y	ADC	OPER,Y	= 79
(indireto,X)	ADC	(OPER,X)	= 61
(indireto),Y	ADC	(OPER),Y	= 71

AND operação: A Λ M → A

N Z C I D V
X X – – – –

imediato	AND	# OPER	= 29
página zero	AND	OPER	= 25
página zero,X	AND	OPER,X	= 35
absoluto	AND	OPER	= 2D
absoluto,X	AND	OPER,X	= 3D
absoluto,Y	AND	OPER,Y	= 39
(indireto,X)	AND	(OPER,X)	= 21
(indireto),Y	AND	(OPER),Y	= 31

ASL operação: C ← [7|6|5|4|3|2|1|0] ← 0

N Z C I D V
X X X – – –

acumulador	ASL A		= 0A
página zero	ASL	OPER	= 06
página zero,X	ASL	OPER,X	= 16
absoluto	ASL	OPER	= 0E
absoluto,X	ASL	OPER,X	= 1E

BCC operação: desvie se C = 0

N Z C I D V
– – – – – –

relativo	BCC	OPER	= 90

BCS operação: desvie se C = 1

N Z C I D V
– – – – – –

relativo	BCS	OPER	= B0

BEQ operação: desvie se Z = 1

N Z C I D V
– – – – – –

relativo	BEQ	OPER	= F0

BMI operação: desvie se N = 1

N Z C I D V
– – – – – –

relativo	BMI	OPER	= 30

BIT operação: A Λ M, M_7 → N, M_6 → V

N Z C I D V
M_7 X – – – M_6

página zero	BIT	OPER	= 24
absoluto	BIT	OPER	= 2C

BNE operação: desvie se Z = 0

N Z C I D V
– – – – – –

relativo	BNE	OPER	= D0

BPL operação: desvie se N = 0

N Z C I D V
– – – – – –

relativo	BPL	OPER	= 10

BRK operação: interrupção forçada PC + 2 ↓ P ↓

N Z C I D V
– – – 1 – –

implícito	BRK		= 00

BVC operação: desvie se V = 0

N Z C I D V
– – – – – –

relativo	BVC	OPER	= 50

BVS operação: desvie se V = 1

N Z C I D V
– – – – – –

relativo	BVS	OPER	= 70

CLC operação: 0 → C

N Z C I D V
— — 0 — — —

implícito CLC			= 18

CLD operação: 0 → D

N Z C I D V
— — — — 0 —

implícito . . . . .CLD			= D8

CLI operação: 0 → I

N Z C I D V
— — — 0 — —

implícito CLI			= 58

CLV operação: 0 → V

N Z C I D V
— — — — — 0

implícito CLV			= B8

CMP operação: A — M

N Z C I D V
X X X — — —

imediato . . . . .CMP	#	OPER	= C9
página zero. . . . . .CMP		OPER	= C5
página zero,X . . . . CMP		OPER,X	= D5
absoluto . . . . .CMP		OPER	= CD
absoluto,X. . . . . .CMP		OPER,X	= DD
absoluto,Y. . . . . .CMP		OPER,Y	= D9
(indireto,X) . . . . .CMP		(OPER,X)	= C1
(indireto),Y . . . . .CMP		(OPER),Y	= D1

CPX operação: X — M

N Z C I D V
X X X — — —

imediato . . . . .CPX	#	OPER	= E0
página zero. . . . . .CPX		OPER	= E4
absoluto . . . . .CPX		OPER	= EC

CPY operação: Y — M

N Z C I D V
X X X — — —

imediato . . . . .CPY	#	OPER	= C0
página zero. . . . . .CPY		OPER	= C4
absoluto . . . . .CPY		OPER	= CC

DEC operação: M — 1 → M

N Z C I D V
X X — — — —

página zero. . . . . .DEC		OPER	= C6
página zero,X . . . . .DEC		OPER,X	= D6
absoluto . . . . .DEC		OPER	= CE
absoluto,X. . . . . .DEC		OPER,X	= DE

DEX operação: X — 1 → X

N Z C I D V
X X — — — —

implícito . . . . .DEX			= CA

DEY operação: Y — 1 → Y

N Z C I D V
X X — — — —

implícito . . . . .DEY			= 88

EOR operação: A ⊻ M → A

N Z C I D V
X X — — — —

imediato . . . . .EOR	#	OPER	= 49
página zero. . . . . .EOR		OPER	= 45
página zero,X . . . . .EOR		OPER,X	= 55
absoluto . . . . .EOR		OPER	= 4D
absoluto,X. . . . . .EOR		OPER,X	= 5D
absoluto,Y. . . . . .EOR		OPER,Y	= 59
(indireto,X) . . . . .EOR		(OPER,X)	= 41
(indireto),Y . . . . .EOR		(OPER),Y	= 51

INC operação: M + 1 → M

N Z C I D V
X X — — — —

página zero. . . . . .INC		OPER	= E6
página zero,X . . . . .INC		OPER,X	= F6
absoluto . . . . .INC		OPER	= EE
absoluto,X. . . . . .INC		OPER,X	= FE

INX operação: X + 1 → X

N Z C I D V
X X — — — —

implícito . . . . .INX			= E8

INY operação: Y + 1 → Y

N Z C I D V
X X — — — —

modo	instrução	operando	código
implícito	INY		= C8

JMP operação: (PC + 1) → PCL
(PC + 2) → PCH

N Z C I D V
— — — — — —

modo	instrução	operando	código
absoluto	JMP	OPER	= 4C
indireto	JMP	(OPER)	= 6C

JSR operação: PC + 2 ↓, (PC + 1) → PCL
(PC + 2) → PCH

N Z C I D V
— — — — — —

modo	instrução	operando	código
absoluto	JSR	OPER	= 20

LDA operação: M → A

N Z C I D V
X X — — — —

modo	instrução	operando	código
imediato	LDA	# OPER	= A9
página zero	LDA	OPER	= A5
página zero,X	LDA	OPER,X	= B5
absoluto	LDA	OPER	= AD
absoluto,X	LDA	OPER,X	= BD
absoluto,Y	LDA	OPER,Y	= B9
(indireto,X)	LDA	(OPER,X)	= A1
(indireto),Y	LDA	(OPER),Y	= B1

LDX operação: M → X

N Z C I D V
X X — — — —

modo	instrução	operando	código
imediato	LDX	# OPER	= A2
página zero	LDX	OPER	= A6
absoluto	LDX	OPER	= AE
página zero,Y	LDX	OPER,Y	= B6
absoluto,Y	LDX	OPER,Y	= BE

LDY operação: M → Y

N Z C I D V
X X — — — —

modo	instrução	operando	código
imediato	LDY	# OPER	A0
página zero	LDY	OPER	A4
página zero,X	LDY	OPER	B4
absoluto	LDY	OPER	AC
absoluto,X	LDY	OPER,X	BC

LSR operação: 0 → [7|6|5|4|3|2|1|0] → C

N Z C I D V
X 0 X — — —

modo	instrução	operando	código
acumulador	LSR	A	= 4A
página zero	LSR	OPER	= 46
página zero,X	LSR	OPER,X	= 56
absoluto	LSR	OPER	= 4E
absoluto,X	LSR	OPER	= 5E

NOP operação: nenhuma operação

N Z C I D V
— — — — — —

modo	instrução	operando	código
implícito	NOP		= EA

ORA operação: A V M → A

N Z C I D V
X X — — — —

modo	instrução	operando	código
imediato	ORA	# OPER	= 09
página zero	ORA	OPER	= 05
página zero,X	ORA	OPER,X	= 15
absoluto	ORA	OPER	= 0D
absoluto,X	ORA	OPER,X	= 1D
absoluto,Y	ORA	OPER,Y	= 19
(indireto,X)	ORA	(OPER,X)	= 01
(indireto),Y	ORA	(OPER),Y	= 11

PHA operação: A ↓

N Z C I D V
— — — — — —

modo	instrução	operando	código
implícito	PHA		= 48

PHP operação: P ↓

N Z C I D V
— — — — — —

modo	instrução	operando	código
implícito	PHP		08

PLA operação: A ↑

N Z C I D V
X X — — — —

modo	instrução	operando	código
implícito	PLA		68

PLP operação: P ↑

N Z C I D V
(conf. o stack)

implícito . . . . .PLP		= 28

ROL operação:

M ou A
[7|6|5|4|3|2|1|0] ← C ←

N Z C I D V
X X X – – –

acumulador . . . . .ROL	A	= 2A
página zero. . . . . .ROL	OPER	= 26
página zero,X . . . . .ROL	OPER,X	= 36
absoluto . . . . .ROL	OPER	= 2E
absoluto,E . . . . .ROL	OPER,X	= 3E

ROR operação:

C → [7|6|5|4|3|2|1|0]

N Z C I D V
X X X – – –

acumulador . . . . .ROR	A	= 6A
página zero. . . . . .ROR	OPER	= 66
página zero,X . . . . .ROR	OPER,X	= 76
absoluto . . . . .ROR	OPER	= 6E
absoluto,X . . . . .ROR	OPER,X	= 7E

RTI operação: P ↑ PC ↑

N Z C I D V
(conf. o stack)

implícito . . . . .RTI		= 40

RTS operação: PC ↑ , PC + 1 → PC

N Z C I D V
– – – – – –

implícito . . . . .RTS		60

SBC operação: A – M – $\bar{C}$ → A

N Z C I D V
X X X – – X

imediato . . . . .SBC	# OPER	= E9
página zero. . . . . .SBC	OPER	= E5
página zero,X . . . . .SBC	OPER,X	F5
absoluto . . . . .SBC	OPER	= ED
absoluto,X . . . . .SBC	OPER,X	= FD
absoluto,Y . . . . .SBC	OPER,Y	F9
(indireto,X) . . . . .SBC	(OPER,X)	E1
(indireto),Y . . . . .SBC	(OPER),Y	F1

SEC operação: 1 → C

N Z C I D V
– – 1 – – –

implícito . . . . .SEC		= 38

SED operação: 1 → D

N Z C I D V
– – – – 1 –

implícito . . . . .SED		= F8

SEI operação: 1 → I

N Z C I D V
– – – 1 – –

implícito . . . . .SEI		= 78

STA operação: A → M

N Z C I D V
– – – – – –

página zero. . . . . .STA	OPER	= 85
página zero,X . . . . .STA	OPER,X	= 95
absoluto . . . . .STA	OPER	= 8D
absoluto,X . . . . .STA	OPER,X	= 9D
absoluto,Y . . . . .STA	OPER,Y	= 99
(indireto,X) . . . . .STA	(OPER,X)	= 81
(indireto),Y . . . . .STA	(OPER),Y	= 91

STX operação: X → M

N Z C I D V
– – – – – –

página zero. . . . . .STX	OPER	= 86
página zero,X . . . . .STX	OPER,X	= 96
absoluto . . . . .STX	OPER	= 8E

STY operação: Y → M

N Z C I D V
– – – – – –

página zero. . . . . .STY	OPER	= 84
página zero,X . . . . .STY	OPER,X	94
absoluto . . . . .STY	OPER	8C

TAX operação: A → X

N Z C I D V
X X – – – –

implícito . . . . .TAX		AA

TAY operação: A → Y		N Z C I D V
		X X – – – –
implícito . . . . .TAY	= A8	
TSX operação: S → X		N Z C I D V
		X X – – – –
implícito . . . . .TSX	= BA	
TXA operação: X → A		N Z C I D V
		X X – – – –
implícito . . . . .TXA	= 8A	
TXS operação: X → S		N Z C I D V
		– – – – – –
implícito . . . . .TXS	= 9A	
TYA operação: Y → A		N Z C I D V
		X X – – – –
implícito . . . . .TYA	= 98	

A tabela a seguir permite calcular o tempo de cada rotina escrita em linguagem de máquina. Os números indicados nas colunas 1 a 13 representam a quantidade dos ciclos necessários para completar a operação em questão. Assim sendo, tendo um Clock de 1 Megahertz no sistema, onde cada "ciclo" corresponde a 1×10^{-6} segundos, uma rotina de um total de 3100 ciclos demora $3100 \times 1 \times 10^{-6}$ = 3,1 milissegundos para ser completada.

A seguir, o significado das colunas 1 a 13:

coluna 1 = acumulador
coluna 2 = imediato
coluna 3 = página zero
coluna 4 = página zero,X
coluna 5 = página zero,Y
coluna 6 = absoluto
coluna 7 = absoluto,X
coluna 8 = absoluto,Y
coluna 9 = implícito
coluna 10 = relativo
coluna 11 = (indireto,X)
coluna 12 = (indireto),Y
coluna 13 = indireto absoluto

	01	02	03	04	05	06	07	08	09	10	11	12	13
ADC	–	2	3	4	–	4	4	4	–	–	6	5	–
AND	–	2	3	4	–	4	4	4	–	–	6	5	–
ASL	2	–	5	6	–	6	7	–	–	–	–	–	–
BCC	–	–	–	–	–	–	–	–	–	2	–	–	–
BCS	–	–	–	–	–	–	–	–	–	2	–	–	–
BEQ	–	–	–	–	–	–	–	–	–	2	–	–	–
BIT	–	–	3	–	–	4	–	–	–	–	–	–	–
BMI	–	–	–	–	–	–	–	–	–	2	–	–	–
BNE	–	–	–	–	–	–	–	–	–	2	–	–	–
BPL	–	–	–	–	–	–	–	–	–	2	–	–	–
BRK	–	–	–	–	–	–	–	–	–	–	–	–	–
BVC	–	–	–	–	–	–	–	–	–	2	–	–	–
BVS	–	–	–	–	–	–	–	–	–	2	–	–	–
CLC	–	–	–	–	–	–	–	–	2	–	–	–	–
CLD	–	–	–	–	–	–	–	–	2	–	–	–	–
CLI	–	–	–	–	–	–	–	–	2	–	–	–	–
CLV	–	–	–	–	–	–	–	–	2	–	–	–	–
CMP	–	2	3	4	–	4	4	4	–	–	6	5	–
CPX	–	2	3	–	–	4	–	–	–	–	–	–	–
CPY	–	2	3	–	–	4	–	–	–	–	–	–	–
DEC	–	–	5	6	–	6	7		–	–	–	–	–
DEX	–	–	–	–	–	–	–	–	2	–	–	–	–
DEY	–	–	–	–	–	–	–	–	2	–	–	–	–
EOR	–	2	3	4	–	4	4	4	–	–	6	5	–
INC	–	–	5	6	–	6	7	–	–	–	–	–	–
INX	–	–	–	–	–	–	–	–	2	–	–	–	–
INY	–	–	–	–	–	–	–	–	2	–	–	–	–
JMP						3							5

	01	02	03	04	05	06	07	08	09	10	11	12	13
JSR	–	–	–	–	–	6	–	–	–	–	–	–	–
LDA	–	2	3	4	–	4	4	4	–	–	6	5	–
LDX	–	2	3	–	4	4	–	4	–	–	–	–	–
LDY	–	2	3	4	–	4	4	–	–	–	–	–	–
LSR	2	–	5	6	–	6	7	–	–	–	–	–	–
NOP	–	–	–	–	–	–	–	–	2	–	–	–	–
ORA	–	2	3	4	–	4	4	4	–	–	6	5	–
PHA	–	–	–	–	–	–	–	–	3	–	–	–	–
PHP	–	–	–	–	–	–	–	–	3	–	–	–	–
PLA	–	–	–	–	–	–	–	–	4	–	–	–	–
PLP	–	–	–	–	–	–	–	–	4	–	–	–	–
ROL	2	–	5	6	–	6	7	–	–	–	–	–	–
ROR	2	–	5	6	–	6	7	–	–	–	–	–	–
RTI	–	–	–	–	–	–	–	–	6	–	–	–	–
RTS	–	–	–	–	–	–	–	–	6	–	–	–	–
SBC	–	2	3	4	–	4	4	4	–	–	6	5	–
SEC	–	–	–	–	–	–	–	–	2	–	–	–	–
SED	–	–	–	–	–	–	–	–	2	–	–	–	–
SEI	–	–	–	–	–	–	–	–	2	–	–	–	–
STA	–	–	3	4	–	4	5	5		–	6	6	–
STX	–	–	3	–	4	4	–	–	–	–	–	–	–
STY	–	–	3	4	–	4	–	–	–	–	–	–	–
TAX	–	–	–	–	–	–	–	–	2	–	–	–	–
TAY	–	–	–	–	–	–	–	–	2	–	–	–	–
TSX	–	–	–	–	–	–	–		2				
TXA	–	–	–	–	–	–			2	–	–	–	
TXS	–	–	–	–	–				2				
TYA	–			–		–		–	2		–	–	

respostas dos exercícios

Página 11

F / V / V / F

CPU 6502 / informações / acumulador

Página 12

F / V / F

bit / verdadeira / Z / sete

Página 21

```
1. LDA #$5A   = A9 5A
   LDY #$13   = A0 13
   BRK        = 00
2. LDX #$F8   = A2 F8
   LDY #$00   = A2 00
   LDA #$24   = A9 24
```

Página 24

1. a) F   b) V   c) F

```
2. LDX #$73   = A2 73
   LDY $4A    = A4 4A
```

3. igual ao conteúdo do endereço $0045

```
4. LDA #$00   = A9 00
   LDX $99    = A2 99
   LDY $99    = A4 99
```

Página 27

```
1. LDX #$32   = A2 32
   LDA $56    = A5 56
2. LDY $13    = A4 13
   LDX $00,Y  = B6 00
```

3. a) V   b) F   c) V

Página 29

```
1. LDA #$23   = A9 23
   LDY $F550  = AC 50 F5
2. solução a) LDX $3F   = A2 3F
   solução b) LDX $003F = AE 3F 00
3. LDX $BC,Y  = B6 BC
```

a) V   b) F   c) V   d) F   e) V

Página 32

1. a resposta é afirmativa
2. LDA #$20 = A9 20
   LDY $A320 = AC 20 A3
   LDA $E555,Y = B9 55 E5
   O valor em A é idêntico ao valor contido no endereço $E555 + (valor de Y).
3. a) V b) F c) V

Página 36

1. a) V b) F c) V d) F
2. LDX #$05 = A2 05
   LDA ($40,X) = A1 40
3. LDX $FD1B = AE 1B FD

Página 39

1. a) V b) V c) F
2. LDA $32 = A5 32
   TAX = AA
3. LDX #$ = A2 00
   LDA ($73,X) = A1 73

Página 46

1. a) F b) V c)F
2. LDA $EC00 = AD 00 EC
   LDX $13 = A6 13
   TAY = A8
   INY = C8
3. TAY = A8
   INY = C8
   TYA = 98

Página 49

1. a) F b) V c) F (só com X)
2. LDX #$00 = A2 00
   LDA ($89,X) = A1 89
   TAX = AA
   TAY = A8
   DEY = 88
   TYA = 98
3. LDX $01F3 = AE F3 01
   INX = E8
   INX = E8
   INX = E8
   BRK = 00
4. LDX #$00 = A2 00
   LDA ($FE,X) = A1 FE
   TAX = AA
   DEX = CA
   DEX = CA
   BRK = 00
5. TAX = AA
   DEX = CA
   TXA = 8A
   BRK = 00

Página 54

1. a) V b) F c) V d) F
2. LDA #$30 = A9 30
   NOP = EA
   TAX = AA
3. LDA #$30 = A9 30
   TAY = A8
   TAX = AA

Página 61

1. V / F (nem sempre) / V / V
2. LDA 30 = A5 30
   STA 1F00 = 8D 1F 00
   RTS = 60
3. LDY #BC = A0 BC
   LDA #00 = A9 00
   STA 55 = 85 55
   LDA #45 = A9 45
   STA 56 = 85 56
   LDA #FF = A9 FF
   STA (55,Y) = 91 55
   RTS = 60
   O valor 11 será achado no endereço $458C (endereço contido em $0055 + valor de Y)

Página 71

1. V / F / V / V
2. a) CLC = 18 18
   ADC FE44 = 6D 44 FE
   (observe a inversão!)
   b) LDX #00 = A2 00
   SEC = 38
   SBC(23,X)= E1 23
   c) Como não existe a instrução SBC X, teremos que colocar o valor de X numa posição livre da memória, por exemplo, $0350:
   LDX #1F = A2 1F
   STX 0350 = 8E 50 03
   (observe a inversão!)
   LDA #7E = A9 7E
   SEC = 38
   SBC 0350 = ED 50 03
   (observe a inversão!)
3. LDA $FF3A = AD 3A FF
   SEC = 38
   SBC $1F = E5 1F
   TAX = AA
   BRK = 00
4. STX $FA = 86 FA
   TYA = 98
   SEC = 38
   SBC $FA = E5 FA
   STA $FA = 85 FA
   RTS = 60
5. TXA = 8A
   STY $FD = 84 FD
   CLC = 18
   ADC $FD = 65 FD
   RTS = 60

Página 74

1. V / F / V / F

Página 83

1. V / F / V
2. CLC = 18
   LDA $0350 = A9 50 03
   ADC $0351 = 6D 51 03
   ADC $0352 = 6D 52 03
   BCS $01 = B0 01
   BRK = 00
   LDA #00 = A9 00
   BRK = 00
   ou
   CLC = 18
   LDA $0350 = A9 50 03
   ADC $0351 = 6D 51 03
   ADC $0352 = 6D 52 03
   BCC $02 = 90 02
   LDA #00 = A9 00
   BRK = 00
3. A instrução correta é BCS $BB.
4. STX $03F0 = 8E F0 03
   SEC = 38
   SBC $03F0 = ED F0 03
   BEQ +3 = F0 03
   LDY #$AA = A0 AA
   RTS = 60
   LDY #$00 = A0 00
   RTS = 60
5. LDA $FFFA = AD FA FF
   CLC = 18
   ADC $FDA8 = 6D A8 FD
   BCS +2 = B0 02
   TAX = AA
   RTS = 60
   TAY = A8
   RTS = 60

Página 88

1. F / F / V

```
2. 0300 LDA $0388     AD 88 03
   0303 ADC $038A     6D 8A 03
   0306 BCS $030B     B0 03
   0308 JSR $FBDD     20 DD FB
   030B BRK           00
```

Página 98

1. V / V / F / V

```
2. LDA $D405   = AD 05
   STA $0350   = 8D 50 03
   AND #$77    = 29 77
   STA $0351   = 8D 51 03
   BRK         = 00
3. LDA $D42D   = AD 2D D4
   EOR $D422   = 4D 22 D4
   BRK         = 00
```

Página 104

1. F / V / F

```
2. CLC         = 18
   LDA #$34    = A9 34
   ROL A       = 2A
   ROL A       = 2A
   BRK         = 00
```

Página 110

1. F / V / V

```
2. 0300 CLC          18
   0301 LDA #$00     A9 00
   0303 LDX #$00     A2 00
   0305 ADC #$09     69 09
   0307 INX          E8
   0308 CPX #$1E     E0 1E
   030A BNE $0305    D0 F9
   030C CMP $FF37    CD 37 FF
   030F BNE $0316    D0 05
   0311 LDX #$00     A2 00
   0313 LDY #$00     A0 00
   0315 BRK          00
   0316 LDX #$FF     A2 FF
   0318 LDY #$FF     A0 FF
   031A BRK          00
```

Página 114

1. V / F / V

```
2. 0300 PHP          08
   0301 PLA          68
   0302 TAY          A8
   0303 BRK          00
```

Página 116

1. F / F / V

```
2. 0300 TSX          BA
   0301 TXA          8A
   0302 STY $0350    8C 50 03
   0305 LDX $0350    AE 50 03
   0308 TXS          9A
```

Página 118

1. V / V / F

Página 126

V / F / V

sempre / definir

Página 132

F / V / V / F / F

setas / geométrico / começo - fim

antes / não

Página 144

F / F / V / V / F / V / V

binário / quatro / 65535

BYTE / B / sinal / 4 / 1

Escreva para ALEPH P.A.P. Ltda., Av. Brig. Faria Lima, 1451/31 - CEP 01451 São Paulo para saber como adquirir a tabela de mnemônicos do 6502, 12 lâminas, plastificada e completa.

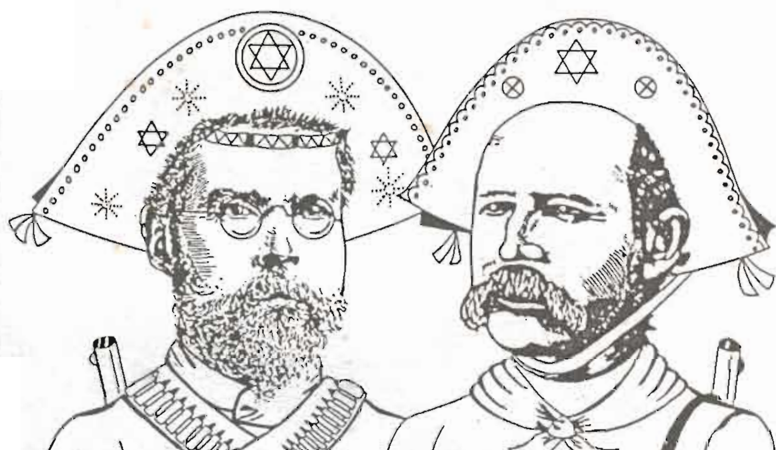
Apple nosso de cada dia

Associe-se ao único clube nacional, dedicado exclusivamente à linha Apple.

Oferecemos:

- disqueteca de 25 MegaBytes.
- dicas de programação.
- aulas de ioga e break,
- horóscopos...

... e principalmente aquele apoio que lhe faltou.



Para maiores informações, escreva-nos.

clube dos applemaníacos

Caixa Postal 54.131 - Cep 01296 - São Paulo - SP

Este livro foi impresso com fotolitos fornecidos pela Editora
Gráfica Palas Athena
Associação "Palas Athena" do Brasil
Rua José Bento, 384
Fone: 279-6288 — CEP 01523
Cambuci — São Paulo

Segredos do apple

A coleção SEGREDOS DO APPLE propõe-se a desvendar os detalhes de um dos primeiros (e mais disseminados) microcomputadores do mundo. A eventual obsolescência pela qual deveria passar o APPLE, numa área da tecnologia de tão bruscas mudanças como é a computação, é constantemente adiada pela riqueza de periféricos e, principalmente, pela super-abundância de software disponível.

Neste volume são abordados, de maneira clara e didática, os segredos da linguagem de máquina do microprocessador 6502, cujo conhecimento é indispensável ao programador que não quer se sujeitar às limitações do BASIC.

Trata-se de uma obra pioneira em português, escrita especificamente para o público brasileiro, e que leva em conta a existência dos computadores nacionais compatíveis com o APPLE II norte-americano.

BERNHARD WOLFGANG SCHÖN formou-se Técnico em Comandos Digitais Industriais na Alemanha Federal em 1971. Após um estágio na Mercedes-Benz de Sindelfingen serviu no exército alemão, tornando-se Tenente da Reserva. Chegou ao Brasil em 1975 onde desenvolveu, paralelamente à sua atividade profissional na Volkswagen, um intenso trabalho de divulgação junto aos usuários de microcomputadores pessoais, principalmente da linha Sinclair.

Após publicar vários artigos em revistas dirigidas a hobbistas da área, sua atividade didática de divulgação culminou com a publicação de "O SEU MICRO E O MUNDO EXTERNO", voltado para a confecção de periféricos para microcomputadores. O grande sucesso desta obra animou-o a continuar nesta linha didática, produzindo agora uma obra indispensável para os "applemaníacos".

Mais que um técnico, Bernhard é um professor, no melhor significado que este termo pode ter, tendo o dom da clareza, conseguindo transformar em coisas simples as mais exotéricas complicações de algo tão árduo quanto a linguagem de máquina.

assembly 6502

Editora Aleph