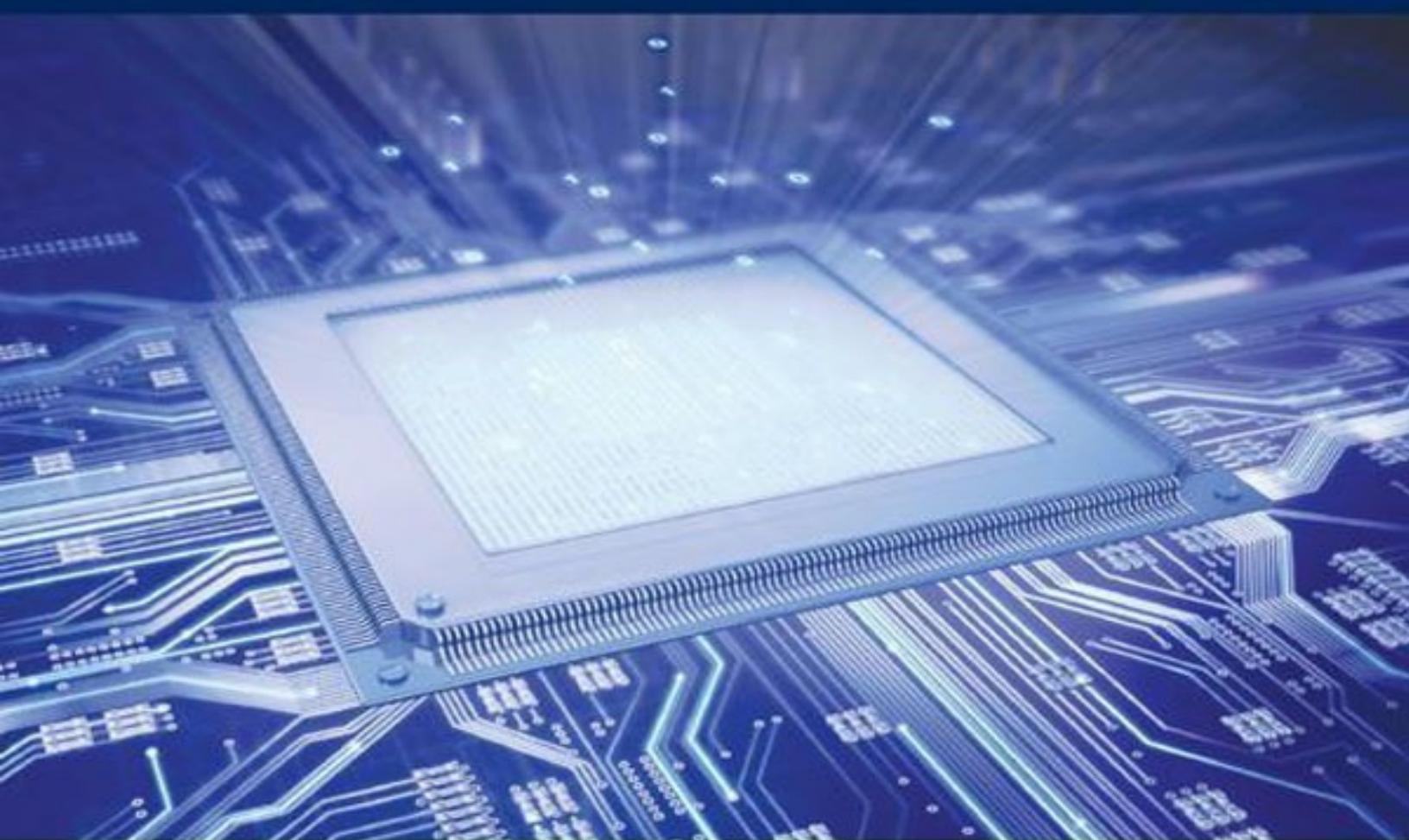




Ricardo Pannain  
Frank Herman Behrens  
Dilermando Piva Jr.



# ORGANIZAÇÃO BÁSICA DE COMPUTADORES E LINGUAGEM DE MONTAGEM



Material  
na WEB

[www.elsevier.com.br](http://www.elsevier.com.br)

# **Organização Básica de computadores e linguagem de montagem**

---

Ricardo Pannain

Frank Herman Behrens

Dilermando Piva Jr.



---

# Obrigado por adquirir este e-book

---

Esta obra é acompanhada de conteúdo complementar. Para acessá-lo, encaminhe a confirmação de compra deste e-book para [pin@elsevier.com.br](mailto:pin@elsevier.com.br), solicitando seu código de acesso.



# Sumário

---

[Capa](#)

[Folha de rosto](#)

[Obrigado por adquirir este e-book](#)

[Cadastro](#)

[Copyright](#)

[Dedicatória](#)

[Agradecimentos](#)

[Prefácio](#)

[Sobre os Autores](#)

[Capítulo 1. Sistemas de computação](#)

[Objetivos Do Capítulo](#)

[Apresentação](#)

[O Que Vem Depois](#)

## **Capítulo 2. Sistemas de numeração**

[Objetivos Do Capítulo](#)

[Apresentação](#)

[O Que Vem Depois](#)

## **Capítulo 3. Aritmética binária**

[Objetivos Do Capítulo](#)

[Apresentação](#)

[O Que Vem Depois](#)

## **Capítulo 4. Variáveis e funções lógicas**

[Objetivos Do Capítulo](#)

[Apresentação](#)

[O Que Vem Depois](#)

## **Capítulo 5. A álgebra booleana**

[Objetivos Do Capítulo](#)

[Apresentação](#)

[O Que Vem Depois](#)

## **Capítulo 6. Funções lógicas e simplificação**

[Objetivos Do Capítulo](#)

[Apresentação](#)

[O Que Vem Depois](#)

## **Capítulo 7. Circuitos lógicos combinacionais (primeira parte)**

[Objetivos Do Capítulo](#)

[Apresentação](#)

[O Que Vem Depois](#)

## [Capítulo 8. Circuitos lógicos combinacionais \(segunda parte\)](#)

[Objetivos Do Capítulo](#)

[Apresentação](#)

[O Que Vem](#)

## [Caítulo 9. Elementos armazenadores](#)

[Objetivos Do Capítulo](#)

[Apresentação](#)

[O Que Vem Deposis](#)

## [Capítulo 10. Circuitos lógicos sequenciais](#)

[Objetivos Do Capítulo](#)

[Apresentação](#)

[O Quevem Depois](#)

## [Capítulo 11. Organização de um computador digital](#)

[Objetivos Do Capítulo](#)

[Apresentação](#)

[O Que Vem Depois](#)

## [capítulo 12. A Execução de uma instrução e a linguagem de montagem](#)

[Objetivo Do Capítulo](#)

[Apresentação](#)

[O Que Vem Depois](#)

## **Caítulo 13. O processador Intel® X86**

[Objetivos Do Capítulo](#)

[Apresentação](#)

[Ovem Depois](#)

## **Capítulo 14. Introdução à programação em linguagem de montagem do Intel 8086**

[Objetivos Do Capítulo](#)

[Apresentação](#)

[O Que Vem Depois](#)

## **Capítalo 15. Instruções de movimentação de dados, lógicas, de deslocamento, de rotação e instruções aritméticas**

[Objetivos Do Capítulo](#)

[Apresentação](#)

[O Que Vem Depois](#)

## **Capítulo 16. Instruções de controle de fluxo**

[Objetivos O Capítulo](#)

[Apresentação](#)

[O Que Vem Depois](#)

## **capítulo 17. Pilha, procedimentos, macros, vetores, matrizes e modos de endereçamento**

[Objetivos Do Capítulo](#)

[Apresentação](#)

[Que Vem Depois](#)

## **capítulo 18. A família PIC®**

[Objetivos Do Capítulo](#)

[Apresentação](#)

[Que Vem Depois](#)

## **capítulo 19. Programação do microcontrolador PIC® 16F84**

[Objetivos Do Capítulo](#)

[Apresentação](#)

[Que Vem Depois](#)

## **Referências Bibliográficas**

# Cadastro

---



Preencha a **ficha de cadastro** no final deste livro e receba gratuitamente informações sobre os lançamentos e as promoções da Elsevier.

Consulte também nosso catálogo completo e últimos lançamentos em [www.elsevier.com.br](http://www.elsevier.com.br)

---

# Copyright

---

© 2012, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei nº 9.610, de 19/02/1998.  
Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá  
ser reproduzida ou transmitida, sejam quais forem os meios empregados:  
eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

*Copidesque:* Jean Xavier

*Revisão:* Lara Alves

*Ilustrador:* Beto Nascimento

*Editoração Eletrônica:* Triall Composição Editorial Ltda.

Elsevier Editora Ltda.

Conhecimento sem Fronteiras

Rua Sete de Setembro, 111 – 16º andar  
20050-006 – Centro – Rio de Janeiro – RJ – Brasil

Rua Quintana, 753 – 8º andar  
04569-011 – Brooklin – São Paulo – SP

Serviço de Atendimento ao Cliente  
0800-026-5340  
[sac@elsevier.com.br](mailto:sac@elsevier.com.br)

ISBN 978-85-352-5021-3  
ISBN (versão eletrônica): 978-85-352-5022-0

**Nota:** Muito zelo e técnica foram empregados na edição desta obra. No entanto,  
podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer  
das hipóteses, solicitamos a comunicação ao nosso Serviço de Atendimento ao

Cliente, para que possamos esclarecer ou encaminhar a questão.  
Nem a editora nem o autor assumem qualquer responsabilidade por eventuais  
danos ou perdas a pessoas ou bens originados do uso desta publicação.

CIP-BRASIL. CATALOGAÇÃO-NA-FONTE SINDICATO NACIONAL DOS  
EDITORES DE LIVROS, RJ

P764o

Piva Junior, Dilermano

Organização básica de computadores / Dilermano Piva Junior, Frank Behrens, Ricardo Pannain / Rio de Janeiro. – Elsevier: 2012.

Inclui bibliografia

ISBN 978-85-352-5021-3

1. Organização de computador. 2. Arquitetura de computadores. I. Behrens, Frank. II. Pannain, Ricardo. III. Título.

12-5197.

CDD: 004.22

CDU: 004.2

---

# Dedicatória

---

*Esta obra é dedicada aos nossos mestres, que semearam em nossas mentes o desejo incansável pela busca contínua do conhecimento.  
Aos nossos alunos, com a esperança de que herdem com alegria e perseverança essa responsabilidade e a propaguem.*

---

# Agradecimentos

---

Agradecemos as nossas famílias, que sempre nos apoiaram, incondicionalmente.

---

# Prefácio

---

Quando fui solicitado a escrever este Prefácio, senti uma mistura de alegria e reconhecimento. Honrado por ter sido convidado quando, ao mesmo tempo, vi retomada um pouco da minha história na PUC-Campinas, quando ministrei a disciplina de Introdução ao Software Básico (ISB) em 1991. Hoje, mais de vinte anos depois, sigo outros caminhos e tenho outros interesses. Mas o fato de ter compartilhado uma história, que hoje me faz cúmplice dos autores desse livro, revela uma experiência inesquecível. Autores esses que são dois ex-colegas professores da PUC-Campinas e um ex-aluno. A disciplina ISB pertencia ao curso de Análise de Sistemas do então Instituto de Informática. E minha formação como engenheiro eletrônico certamente muito me ajudou. Era o início da minha vida acadêmica.

O livro que aqui se apresenta traz em seu conteúdo informações preciosas. Uma parte inicial, dedicada aos conceitos básicos da computação, permite aos recém-chegados compreender a representação e a forma como os dados (e informações) são processados. Em sua segunda parte, a lógica binária é explorada. A terceira parte contém material sobre a arquitetura básica de um computador e sua linguagem. As quarta e quinta partes apresentam estudos de caso de um processador e de um controlador.

O leitor dessa obra deparar-se-á com o encantamento inicial sobre sistemas de numeração. Descobrir que as máquinas, em sua arquitetura atual, trabalham com dígitos binários, e não decimais como os seres humanos, revela uma descoberta fascinante. Certamente, isso se deve ao fato de, do ponto de vista da representação e da medição, valores binários serem muito mais fáceis de se trabalhar: trata-se da presença ou ausência de uma grandeza física, normalmente a tensão ou a corrente elétrica. Isso implica, certamente, uma responsabilidade adicional: entender como se faz cálculos nessa base. E conhecer dispositivos eletrônicos que viabilizam toda uma gama de possibilidades de trabalho com essa lógica.

O universo do computador, explorado de forma bastante acessível, permite a

compreensão do funcionamento de suas partes: quem é responsável pelo quê. Conhecer que há um circuito capaz de gerenciar toda a troca de sinais entre as diversas partes, fornecendo um padrão de tempo para que isso ocorra. Saber da existência de um núcleo que coordena todas as ações executadas pela máquina, enviando e recebendo dados e guardando-os em locais de armazenamento adequados. Perceber que há uma coleção de fios que faz a comunicação entre as partes e que há elementos que permitem entrar com dados e sair com resultados. Tudo controlado por uma sequência de instruções, numa linguagem que a máquina comprehende.

Então, partindo do conhecimento teórico, experimentar componentes reais para executar tudo o que foi ensinado pela teoria. A teoria assim se faz prática: o computador real não é uma máquina teórica, o que é percebido quando estudamos circuitos eletrônicos e os programamos para executar tarefas que gostaríamos que o computador solucionasse.

Tenho a certeza de que, neste livro, o leitor encontrará uma fonte de pesquisa e de estudos, indispensável para todos os que, como eu, têm grande interesse em assuntos computacionais. Ainda acrescento que, ao lê-lo inúmeras vezes, o leitor poderá construir alicerces para os fundamentos da ciência da computação e, com os exercícios propostos ao final de cada parte, assentar seu aprendizado. Bons estudos!

**Prof. Dr. João Luís Garcia Rosa**, *Laboratório de Computação Bioinspirada  
Departamento de Ciências de Computação Instituto de Ciências Matemáticas e de  
Computação Universidade de São Paulo - USP*

---

# Sobre os Autores

---

## Ricardo Pannain

É professor titular do Centro de Ciências Exatas, Ambientais e de Tecnologias da Pontifícia Universidade Católica de Campinas – PUC-Campinas desde 1988, e professor doutor do Instituto de Computação da Universidade Estadual de Campinas – UNICAMP desde 1990. Doutor e Mestre em Engenharia Elétrica pela UNI-CAMP na área de Arquitetura de Computadores. Atua nas áreas de Arquitetura e Organização de Computadores e Projeto de Sistemas Digitais. Foi coordenador do Curso de Engenharia de Computação, Diretor Adjunto do Centro de Ciências Exatas, Ambientais e de Tecnologias e, atualmente, é Pró-Reitor de Administração da PUC-Campinas. Atuou na área de Microeletrônica no Brasil no período de 1983 a 1992.

## Frank Herman Behrens

É engenheiro de projeto de circuitos integrados digitais na Freescale Semicondutores desde 2000, tendo participado de diversos projetos de circuitos integrados analógicos e digitais. Doutor em Engenharia Elétrica pela UNICAMP na área de Microeletrônica. Atua profissionalmente nas áreas de Microeletrônica, projeto de circuitos integrados digitais, desenvolvimento de IP (*Intellectual Property*) de circuitos digitais sintetizáveis descritos em HDL (*hardware description language*). Atua também como professor da Faculdade de Engenharia Elétrica da Pontifícia Universidade Católica de Campinas desde 1995. Tem atuado na área de Microeletrônica no Brasil em empresas, institutos de pesquisa e de ensino desde 1983. Autor de dezenas de artigos nacionais e internacionais na área de microeletrônica e projeto de sistemas eletrônicos digitais integrados.

## Dilermando Piva Jr.

É coordenador de Educação a Distância do Centro Paula Souza para o Ensino Superior (Fatecs). Doutor em Engenharia Elétrica e de Computação pela

UNICAMP na área de Automação – Inteligência Artificial e Ensino a Distância. Professor pleno da Faculdade de Tecnologia de Indaiatuba e responsável técnico do Centro Paula Souza no convênio com a UNIVESP (Universidade Virtual do Estado de SP). Atua nas áreas de Educação a Distância, Programação (Algoritmos e Engenharia de Software), Organização de Computadores, Inteligência Artificial, Tecnologia Educacional e Gestão Educacional. Autor de dezenas de artigos nacionais e internacionais e de livros na área de tecnologia educacional, educação a distância e algoritmos. Avaliador do Ministério da Educação e do Conselho Estadual de Educação do Estado de SP.

---

## CAPÍTULO 1

# Sistemas de computação

---

### Objetivos do capítulo

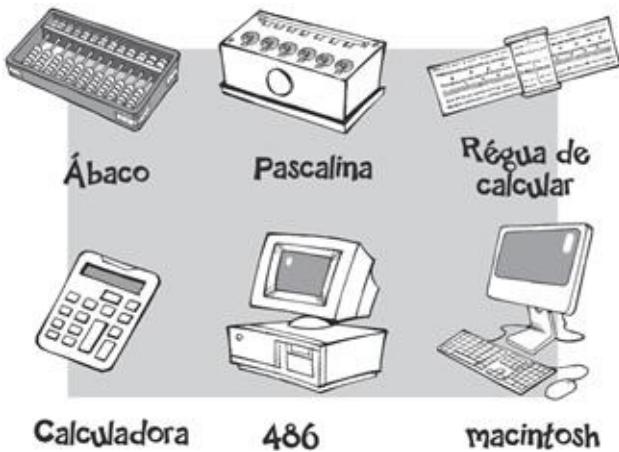
Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Compreender a evolução das máquinas que originaram o computador que utilizamos hoje.
- Identificar as principais máquinas criadas no processo de evolução das máquinas de calcular.
- Compreender a complexidade envolvida no processo de evolução das máquinas de calcular.
- Compreender e utilizar os conceitos de hardware e software.
- Identificar as tecnologias envolvidas nas diversas gerações de computadores.
- Compreender como ocorre o processamento das informações em um sistema digital de computadores.
- Identificar as partes componentes de uma informação digital.
- Compreender como a informação é codificada digitalmente.
- Identificar os vários tipos de software básico e os aplicativos.



### Apresentação

Quando pensamos em todas as coisas que utilizamos em nosso dia a dia, frequentemente não consideramos as ferramentas utilizadas por nossas gerações anteriores. Mas é com base nessa reflexão que podemos vislumbrar a evolução e, o que é mais importante, imaginar como será o futuro.



Para entender melhor o papel da evolução, principalmente na área tecnológica, experimente fazer o seguinte exercício: feche os olhos e pense em como era sua vida há 10 anos e nas tecnologias que estavam presentes no seu dia a dia. Certamente não era comum encontrarmos pessoas com *smartphones*, *iPads* ou *tablets*, e não tínhamos canais HD; a TV digital ainda estava sendo criada nos laboratórios.

Agora tente imaginar como será sua vida daqui a 10 ou até mesmo cinco anos... Difícil, não é? Mas pensando em como se deu a evolução, o ritmo e as tendências de modernização do passado, poderíamos até arriscar alguns palpites!

E é com esse propósito que neste primeiro capítulo vamos mostrar um pouco da evolução dos computadores, suas principais tecnologias e os conceitos básicos da área.

Depois de resgatarmos os conceitos de hardware e software, vamos fazer uma viagem pelas várias gerações de computadores, identificando suas principais características e as tecnologias que marcaram cada uma delas.

Boa viagem!

## Fundamentos

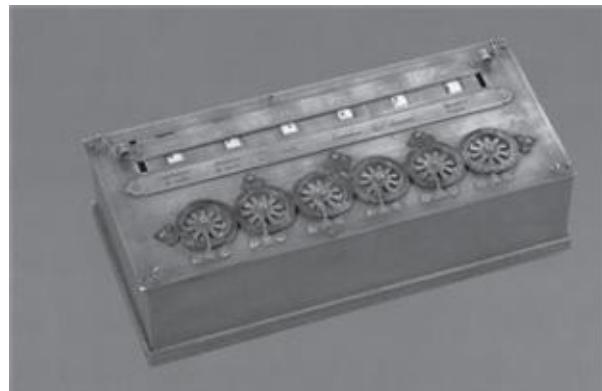
### **As primeiras máquinas**

Alguns autores geralmente associam o início do processamento de dados ao aparecimento do ábaco ([Figura 1.1](#)), por volta de 2500 a.C. Contudo, não podemos esquecer outros inventos tão importantes quanto o ábaco ao considerar a história dos computadores, como as tabelas móveis do escocês John Napier, em 1614; a régua de cálculo do inglês Willian Oughtred, em 1621, e a primeira máquina de calcular, idealizada pelo alemão Wilhelm Schikard em 1623.



**FIGURA 1.1** O ábaco.

Outro fato marcante foi o aparecimento da máquina pascalina, inventada pelo francês Blaise Pascal em 1624. Esta máquina somava e subtraía utilizando um sistema de engrenagens para os cálculos.



**FIGURA 1.2** A máquina de Pascal.

O inglês Charles Babbage projetou a máquina diferencial, que nunca chegou a funcionar, e a máquina analítica, que não foi concluída. Apesar disso, os conceitos de Babbage foram muito importantes no desenvolvimento de máquinas que surgiram depois, como a máquina de Hollerith e o computador MARK I.

Em 1880, Herman Hollerith inventou uma máquina para automatizar as informações do recenseamento da população norte-americana. O sistema foi utilizado no Censo Americano de 1890: a tabuladora fazia a leitura de cartões perfurados e assim efetuava a tabulação das informações. Hollerith fundou em

1896 a Tabulating Machine Company, que em 1924 deu origem à International Business Machine (IBM), dirigida por Tomas Watson.



**FIGURA 1.3** Cartão de papel perfurado pela tabuladora de Hollerith.

## Os Primeiros Computadores

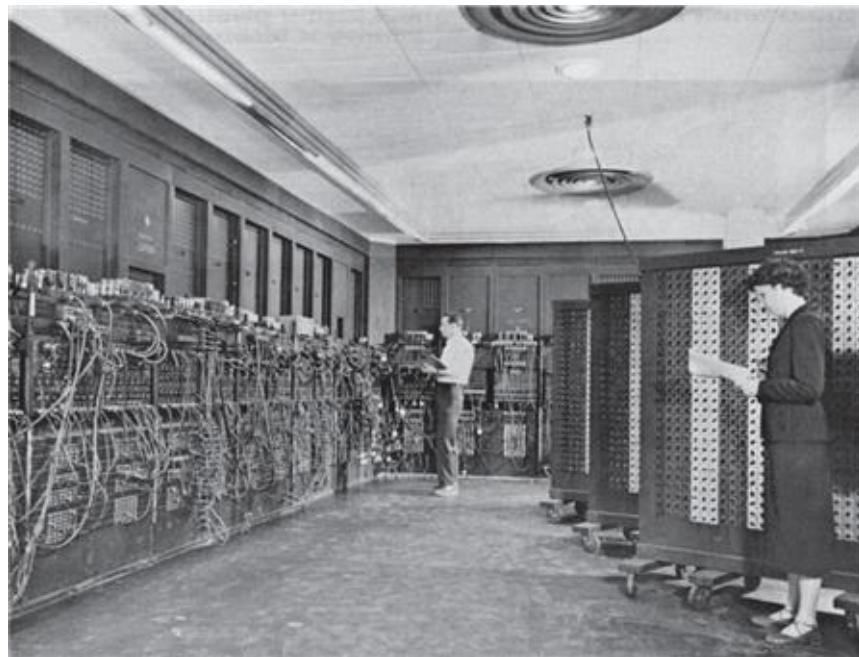
Os primeiros computadores datam de 1937. Naquele ano, com a iniciativa de Howard Aiken, a IBM e a Marinha Americana construíram o MARK I, um computador eletromecânico.

Em 1939, o americano John Atanasof criou o primeiro computador eletrônico digital, construído com válvulas eletrônicas. Na mesma época, em 1940, o alemão Konrad Zuse, incentivado pela Segunda Guerra Mundial, construiu um computador eletrônico digital, destruído pelos americanos ao término da Segunda Guerra.

Alan Turing, famoso pela Máquina de Turing, também contribuiu com um computador eletrônico nos idos de 1941; o objetivo era decifrar códigos de guerra dos alemães.

O computador ENIAC (*Electronic Integer Analyser and Computer*) era uma máquina com aproximadamente 170 m<sup>2</sup> e 18 mil válvulas eletrônicas, que pesava cerca de 30 toneladas. O ENIAC foi construído em 1946 e tornou-se um marco para o aparecimento de novas máquinas. Sua programação era feita por

meio de ligações de fios em seu painel frontal, o que dificultava muito a preparação do computador para a realização das tarefas. Sua confiabilidade era baixa, pois era comum os seus componentes queimarem, principalmente as válvulas eletrônicas. Sua velocidade de cálculo era mil vezes maior que a do MARK I.



**FIGURA 1.4** O ENIAC (*Electronic Integer Analyser and Computer*). Fonte: <http://gadgetfanbr.blogspot.com/2011/04/eniac-first-computer.html>

Um dos mais importantes acontecimentos para o aparecimento dos atuais computadores foi o estabelecimento dos conceitos de programa, processamento e armazenamento de instruções e dados no computador, assim como a utilização de operações com números binários, esta apresentada por John Von Neumann em 1945. Um exemplo clássico de utilização dessa teoria, ainda na década de 1940, foi o EDVAC (*Electronic Discrete Variable Automatic Computer*), um computador 90% menor que o ENIAC.

A partir daquele momento, os computadores e todas as tecnologias envolvidas em cada nova fase de desenvolvimento dos computadores foram classificados por gerações, levando em consideração características específicas e similaridades.

Apresentamos a seguir as gerações dos computadores, de forma bem objetiva; se o leitor quiser se aprofundar mais no assunto, poderá pesquisá-lo na internet.

# **As Gerações Dos Computadores**

Alguns autores classificam os computadores em gerações. A evolução dos computadores, ou seja, a mudança de gerações ao longo do tempo, é marcada principalmente pela evolução do hardware, ligada ao desenvolvimento do software.

## ***Primeira geração***

Os computadores de primeira geração apareceram aproximadamente no período de 1945 a 1959. Eram máquinas que utilizavam válvulas eletrônicas para o processamento dos dados, e sua programação era feita por ligações de fios. Eles eram lentos e enormes, além de consumirem muita energia (principalmente por causa das válvulas eletrônicas), esquentando muito. Um exemplo destes computadores é o ENIAC, mencionado anteriormente.

## ***Segunda geração***

A segunda geração apareceu no período de 1959 a 1964, e foi marcada pela substituição das válvulas eletrônicas pelos transistores (dispositivos semicondutores) montados em circuitos impressos. Circuitos com transistores consomem menos energia, têm maior confiabilidade, menor tamanho e menor custo, além de trabalharem a uma velocidade maior, ou seja, processam as informações mais rapidamente se comparados aos circuitos que utilizam válvulas eletrônicas. O exemplo mais conhecido desta geração foi o IBM 1401.

## ***Terceira geração***

Nos computadores da terceira geração (1964 a 1970), os transistores foram substituídos por circuitos integrados, e assim os circuitos digitais que constituíam o computador ficaram menores e mais compactos, além de apresentarem menor custo, maior velocidade de processamento e menor consumo de energia. Um marco com relação ao software é o início da utilização de programas que permitiam o gerenciamento e a utilização dos recursos de hardware, facilitando o uso do computador pelo usuário. O representante dessa geração é o IBM 360.

## ***Quarta geração***

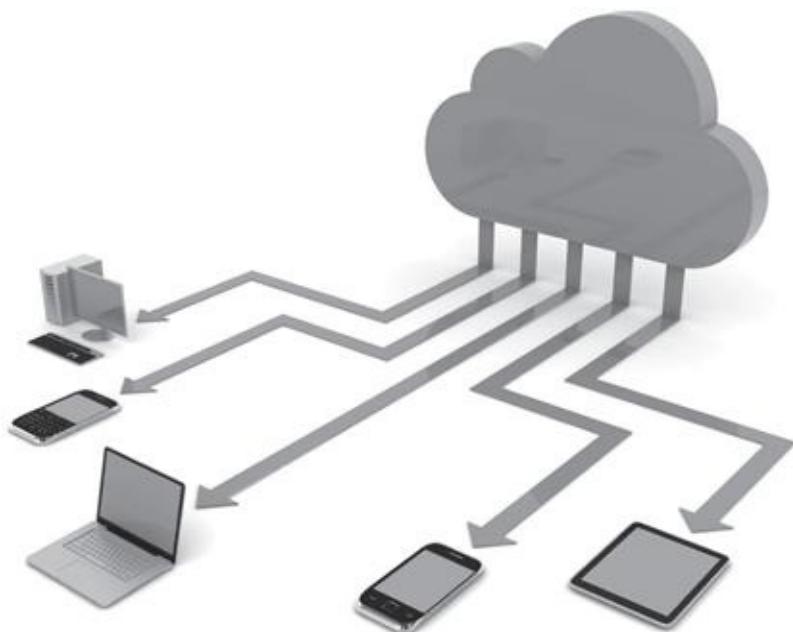
Podemos afirmar que os computadores desta geração, que começou no início da

década de 1970, praticamente ainda estão em uso, sobrepondose à quinta geração (a qual veremos a seguir). Ela tem como característica o aperfeiçoamento das tecnologias existentes, o que deixou as máquinas com um maior grau de compactação pela integração dos circuitos integrados. Como pudemos ver na evolução das gerações anteriores, isto acarreta mais confiabilidade e maior velocidade de processamento.

## **Quinta geração**

Alguns autores classificam sob a quinta geração as máquinas chamadas de computadores inteligentes. Isto se iniciou na década de 1990, com as pesquisas relacionadas à inteligência artificial (feitas principalmente pelos japoneses), aos sistemas especialistas e à linguagem natural.

Nos dias de hoje, podemos dizer que a verdadeira característica da quinta geração é a conectividade, ou seja, a capacidade de fazer com que os usuários conectem seus computadores a outros computadores, ou a outras redes de computadores, chegando até os conceitos modernos de computação em nuvem (*cloud computing*). Computação em nuvem (Figura 1.5), em outras palavras, é a utilização de recursos computacionais (memória e processamento) compartilhados e interligados por meio da internet (rede mundial de computadores).



**FIGURA 1.5** Computação em nuvem (*cloud computing*).

# Hardware E Software

Em sua composição atual, um computador é constituído de hardware e software, dois componentes de natureza complementar e indissociáveis que cooperam para produzir o processamento da informação. Vocês já devem saber diferenciar os dois, mas para dirimir qualquer dúvida iremos formalizar as definições.

## **Hardware**

O hardware é o conjunto de elementos físicos (circuitos eletrônicos, monitor, teclado, elementos de armazenamento de dados, entre outros) que compõem o computador ([Figura 1.6](#)). O hardware pode ser entendido como uma máquina lógica programável, cuja operação é controlada pelo software.



**FIGURA 1.6** Computador digital: gabinete com os circuitos digitais, monitor de vídeo, teclado e mouse.

## **Software**

O software é o conjunto de programas que permitem a transformação de dados, ou seja, o processamento de dados. São exemplos de softwares os sistemas operacionais, os compiladores (linguagens de programação) e os aplicativos (editor de texto, planilha de cálculo, gerenciador de banco de dados, navegadores de internet etc.). É o software que controla o hardware.

## **Processamento de informações**

Um computador digital é uma máquina eletrônica capaz de processar informações, pelo hardware e pelo software. O termo “digital” deve-se ao tipo dos circuitos que formam as principais partes do computador – justamente os circuitos digitais. Estes circuitos funcionam com sinais ou variáveis lógicas, ou seja, sinais que, em qualquer intervalo de tempo, admitem apenas dois valores.

Processar uma informação significa modificar um conjunto de dados produzindo um novo conjunto, que é o resultado dessa transformação. O conjunto de dados a ser modificado é chamado de **dados de entrada**, e o resultado desta modificação são os **dados de saída** (Figura 1.7).



**FIGURA 1.7** O processamento de informações.

O processamento de dados é feito pelo conjunto hardware e software, e o funcionamento de um depende do funcionamento do outro. Mais adiante veremos que um programa é um conjunto de instruções que faz com que o hardware realize algumas tarefas. O conjunto de tarefas realizadas é responsável pelo processamento dos dados.

## **Representação Das Informações Em Um Sistema De Computação**

### **Bits, bytes e palavras**

Como já vimos, o computador digital trabalha com sinais elétricos que em um dado intervalo de tempo admitem apenas dois valores. Para representar estes dois valores, utilizamos o *bit*. O bit é a menor parte de uma informação no mundo digital. A palavra bit é originária do termo ***binary digit*** (dígito binário), cujo significado é um dígito numérico que pode admitir apenas dois valores, 0 ou 1.

Um conjunto de 8 bits é chamado de *byte*, e um conjunto de n bytes é denominado *palavra* (*word*, **em inglês**). O byte e a palavra possibilitam a representação das informações para o computador. A palavra de um computador é o conjunto de bits que contém uma informação a ser processada, como uma

instrução a ser executada, um dado ou um endereço de um dado armazenado no computador. Chamamos de *nibble* o conjunto de quatro *bits*.

---

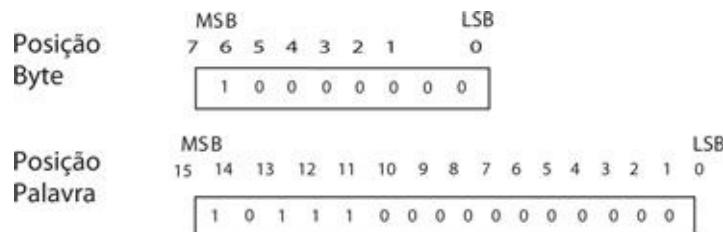
### Tabela 1.1

#### Bits, bytes e palavras

---

Bit	<i>binary digit</i> (0 ou 1)
Byte	Conjunto de oito bits
Palavra	Conjunto de n bytes
Nibble	Conjunto de quatro bits

Em um byte, ou em uma palavra, cada bit ocupa uma posição. Na [Figura 1.8](#), podemos ver como é definida esta posição, tanto em 1 byte como em uma palavra de 16 bits.



**FIGURA 1.8** Posição dos bits dentro de um byte e de uma palavra de 16 bits.

Ainda considerando a figura, vemos que o bit mais à esquerda de uma palavra é chamado de **MSB** (*Most Significative Bit*- bit mais significativo), assim como o bit mais à direita é chamado de **LSB** (*Least Significative Bit* – bit menos significativo).

### Representação de caracteres alfanuméricicos

Já vimos que uma informação pode ser uma instrução, um dado ou um endereço. Os dados podem representar valores numéricos ou caracteres. Os caracteres são representações de símbolos normalmente utilizados em textos.

Para representar os caracteres alfanuméricicos em um computador digital é necessário que os codifiquemos em códigos binários, utilizando palavras ou bytes para representá-los. São duas as codificações mais utilizadas: o código **EBCDIC** (*Extended Binary Coded Decimal Interchange Code*) e o código

**ASCII (American Standard Code Interchange Information).**

O código EBCDIC foi desenvolvido pela IBM no início dos anos 1960, e foi bastante utilizado em computadores de grande porte fabricados pela própria IBM. Nesta codificação, o caractere é representado por 1 byte, possibilitando a representação de 256 caracteres diferentes. Esta codificação, juntamente com a codificação ASCII, foi a primeira tentativa de padronizar representação dos caracteres.

O código ASCII inicialmente utilizava sete bits, possibilitando a representação de 128 caracteres diferentes. Para aumentar o número de caracteres que pudessem ser representados, surgiu o *Extended ASCII*, com oito bits. Este tinha o mesmo número de possíveis caracteres a serem representados que o código EBCDIC, ou seja, 256. Este código é o mais utilizado atualmente, e pode representar caracteres alfanuméricos, caracteres referentes à pontuação e alguns caracteres especiais. A [Tabela 1.2](#) mostra a lista de caracteres ASCII e seus respectivos códigos, na representação decimal e hexadecimal.

---

### **Tabela 1.2**

#### **Tabela ASCII, oito bits (*Extended American Standard Interchange Information*)**

---

<b>Decimal</b>	<b>Binário</b>	<b>Hexa</b>	<b>Referência/Caractere</b>
0	00000000	00	Null - NUL
1	00000001	01	Start of Heading - SOH
2	00000010	02	Start of Text - STX
3	00000011	03	End of Text - ETX
4	00000100	04	End of Transmission - EOT
5	00000101	05	Enquiry - ENQ
6	00000110	06	Acknowledge - ACK
7	00000111	07	Bell, rings terminal bell - BEL
8	00001000	08	BackSpace - BS
9	00001001	09	Horizontal Tab - HT
10	00001010	0A	Line Feed - LF
11	00001011	0B	Vertical Tab - VT
12	00001100	0C	Form Feed - FF
13	00001101	0D	Enter - CR
14	00001110	0E	Shift-Out - SO
15	00001111	0F	Shift-In - SI
16	00010000	10	Data Link Escape - DLE
17	00010001	11	Device Control 1 - D1
18	00010010	12	Device Control 2 - D2
19	00010011	13	Device Control 3 - D3
20	00010100	14	Device Control 4 - D4
21	00010101	15	Negative Acknowledge - NAK
22	00010110	16	Synchronous idle - SYN
23	00010111	17	End Transmission Block - ETB
24	00011000	18	Cancel line - CAN
25	00011001	19	End of Medium - EM
26	00011010	1A	Cancel line - CAN
27	00011011	1B	Escape - ESC
28	00011100	1C	File Separator - FS
29	00011101	1D	Group Separator - GS
30	00011110	1E	Record Separator - RS
31	00011111	1F	Unit Separator - US
32	00100000	20	Space - SPC



33	00100001	21	!
34	00100010	22	"
35	00100011	23	#
36	00100100	24	\$
37	00100101	25	%
38	00100110	26	&
39	00100111	27	'
40	00101000	28	(
41	00101001	29	)
42	00101010	2A	*
43	00101011	2B	+
44	00101100	2C	,
45	00101101	2D	-
46	00101110	2E	.
47	00101111	2F	/
48	00110000	30	0
49	00110001	31	1
50	00110010	32	2
51	00110011	33	3
52	00110100	34	4
53	00110101	35	5
54	00110110	36	6
55	00110111	37	7
56	00111000	38	8
57	00111001	39	9
58	00111010	3A	:
59	00111011	3B	;
60	00111100	3C	<
61	00111101	3D	=
62	00111110	3E	>



63	0011111	3F	?
64	01000000	40	@
65	01000001	41	A
66	01000010	42	B
67	01000011	43	C
68	01000100	44	D
69	01000101	45	E
70	01000110	46	F
71	01000111	47	G
72	01001000	48	H
73	01001001	49	I
74	01001010	4A	J
75	01001011	4B	K
76	01001100	4C	L
77	01001101	4D	M
78	01001110	4E	N
79	01001111	4F	O
80	01010000	50	P
81	01010001	51	Q
82	01010010	52	R
83	01010011	53	S
84	01010100	54	T
85	01010101	55	U
86	01010110	56	V
87	01010111	57	W
88	01011000	58	X
89	01011001	59	Y
90	01011010	5A	Z
91	01011011	5B	[
92	01011100	5C	\



93	01011101	5D	]
94	01011110	5E	^
95	01011111	5F	_
96	01100000	60	`
97	01100001	61	a
98	01100010	62	b
99	01100011	63	c
100	01100100	64	d
101	01100101	65	e
102	01100110	66	f
103	01100111	67	g
104	01101000	68	h
105	01101001	69	i
106	01101010	6A	j
107	01101011	6B	k
108	01101100	6C	l
109	01101101	6D	m
110	01101110	6E	n
111	01101111	6F	o
112	01110000	70	p
113	01110001	71	q
114	01110010	72	r
115	01110011	73	s
116	01110100	74	t
117	01110101	75	u
118	01110110	76	v
119	01110111	77	w
120	01111000	78	x
121	01111001	79	y
122	01111010	7A	z



123	01111011	7B	{
124	01111100	7C	
125	01111101	7D	}
126	01111110	7E	-
127	01111111	7F	Delete
128	10000000	80	Ç
129	10000001	81	Ü
130	10000010	82	é
131	10000011	83	â
132	10000100	84	ä
133	10000101	85	à
134	10000110	86	å
135	10000111	87	ç
136	10001000	88	ê
137	10001001	89	ë
138	10001010	8A	è
139	10001011	8B	í
140	10001100	8C	î
141	10001101	8D	ì
142	10001110	8E	Ã
143	10001111	8F	Å
144	10010000	90	É
145	10010001	91	æ
146	10010010	92	Æ
147	10010011	93	ô
148	10010100	94	ö
149	10010101	95	ò
150	10010110	96	û
151	10010111	97	ù
152	10011000	98	ÿ
153	10011001	99	Ö



154	10011010	9A	Ü
155	10011011	9B	ø
156	10011100	9C	£
157	10011101	9D	Ø
158	10011110	9E	×
159	10011111	9F	f
160	10100000	A0	á
161	10100001	A1	à
162	10100010	A2	ó
163	10100011	A3	ú
164	10100100	A4	ñ
165	10100101	A5	Ñ
166	10100110	A6	ä
167	10100111	A7	ö
168	10101000	A8	¿
169	10101001	A9	®
170	10101010	AA	-
171	10101011	AB	½
172	10101100	AC	¼
173	10101101	AD	i
174	10101110	AE	«
175	10101111	AF	»
176	10110000	B0	
177	10110001	B1	
178	10110010	B2	
179	10110011	B3	
180	10110100	B4	-
181	10110101	B5	Á
182	10110110	B6	Â
183	10110111	B7	À
184	10111000	B8	©



185	10111001	B9	¶
186	10111010	BB	
187	10111011	BC	¶
188	10111100	BD	¶
189	10111101	BE	¢
190	10111110	BF	¥
191	10111111	CO	⊤
192	11000000	C1	Ł
193	11000001	C1	⊥
194	11000010	C2	⊤
195	11000011	C3	⊤
196	11000100	C4	—
197	11000101	C5	+
198	11000110	C6	ã
199	11000111	C7	Ã
200	11001000	C8	Ł
201	11001001	C9	ſ
202	11001010	CA	Ł
203	11001011	CB	ſ
204	11001100	CC	ſ
205	11001101	CD	=
206	11001110	CE	ſ
207	11001111	CF	¤
208	11010000	D0	ð
209	11010001	D1	Đ
210	11010010	D2	Ê
211	11010011	D3	Ë
212	11010100	D4	È
213	11010101	D5	í
214	11010110	D6	í
215	11010111	D7	î

216	11011000	D8	Í
217	11011001	D9	Ј
218	11011010	DA	Г
219	11011011	DB	█
220	11011100	DC	█
221	11011101	DD	Ј
222	11011110	DE	І
223	11011111	DF	█
224	11100000	E0	Ó
224	11100000	E0	Ó
225	11100001	E1	ß
226	11100010	E2	Ô
227	11100011	E3	Ò
228	11100100	E4	ð
229	11100101	E5	Õ
230	11100110	E6	µ
231	11100111	E7	þ
232	11101000	E8	þ
233	11101001	E9	Ú
234	11101010	EA	Û
235	11101011	EB	Ù

236	11101100	EC	ý
237	11101101	ED	Ý
238	11101110	EE	—
239	11101111	EF	,
240	11110000	F0	--
241	11110001	F1	±
242	11110010	F2	—
243	11110011	F3	$\frac{3}{4}$
244	11110100	F4	¶
245	11110101	F5	§
246	11110110	F6	$\div$
247	11110111	F7	,
248	11111000	F8	°
249	11111001	F9	”
250	11111010	FA	·
251	11111011	FB	‘
252	11111100	FC	$^3$
253	11111101	FD	$^2$
254	11111110	FE	■
255	11111111	FF	

## Software Básico E Software Aplicativo

Chamamos de software básico os programas que são responsáveis por tarefas básicas de um sistema de computação. Eles são listados a seguir.

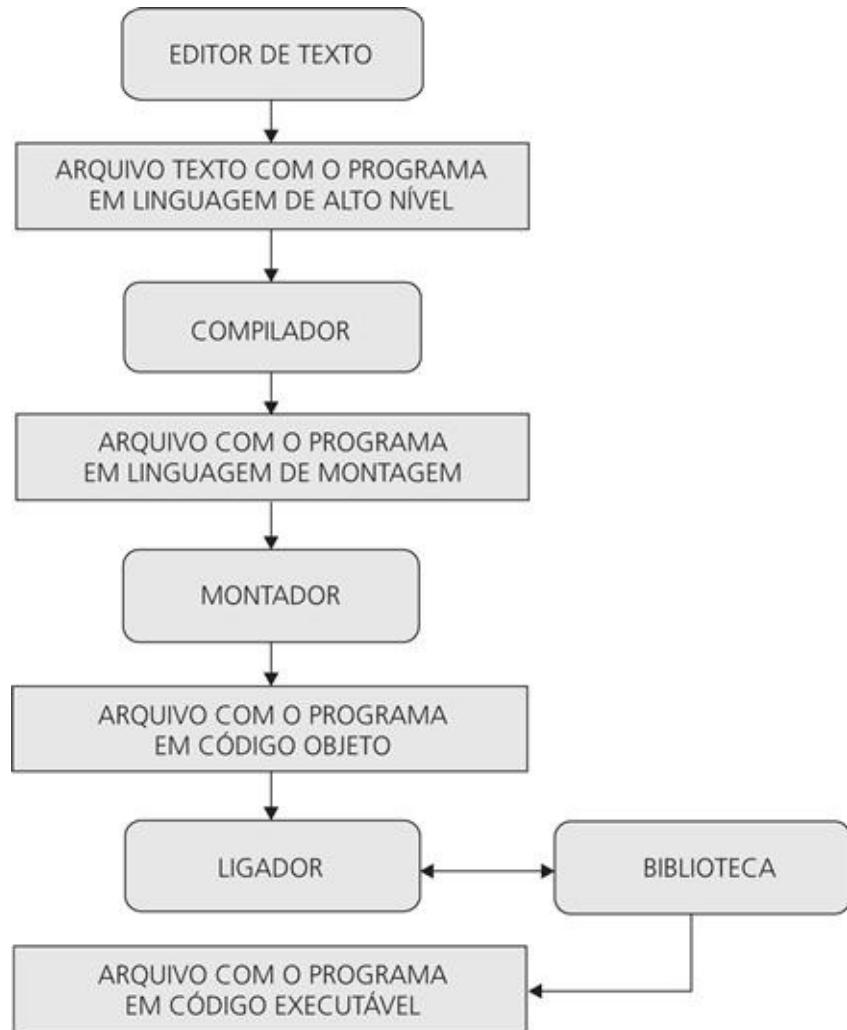
a) **Sistemas operacionais:** são um conjunto de rotinas que possibilitam a utilização dos recursos de hardware pelos usuários e pelo próprio hardware, de uma maneira mais eficiente, confortável e simples. Os sistemas operacionais são responsáveis pela interface entre o ser humano e o computador (interface humano–computador), pelo gerenciamento de

memória (recurso do hardware) e pela automatização do uso de periféricos (teclado, mouse, monitor, unidades de armazenamento secundárias etc.). São exemplos destes sistemas todas as versões do LINUX, do Windows, do MAC OS, do OS X LION e outros.

- b) **Compiladores de linguagens de programação:** são programas que traduzem um programa de usuário (aplicativo) escrito em uma linguagem de alto nível para uma linguagem de baixo nível. Uma linguagem de alto nível é aquela que está mais próxima do usuário, ou seja, de fácil entendimento. Uma linguagem de baixo nível é aquela que está mais próxima do computador, pronta ou quase pronta a ser executada. Os compiladores C e C++ são exemplos de compiladores de linguagem de alto nível.
- c) **Montadores (assemblers):** são programas que traduzem programas escritos em uma linguagem de montagem (*assembly*), ou seja, uma linguagem de baixo nível, para a chamada linguagem de máquina. Uma linguagem de máquina contém as instruções que o computador consegue executar.
- d) **Carregadores (loaders):** são rotinas que têm a função de carregar um programa, ou seja, colocá-lo na memória para a sua execução.
- e) **Ligadores (Linkers):** são rotinas que fazem a ligação entre as chamadas de funções de uma biblioteca existente em um programa e esta própria biblioteca. Estas rotinas podem fazer também a ligação entre dois ou mais programas traduzidos separadamente.
- f) **Aplicativos:** são programas desenvolvidos para uma determinada aplicação (como seu próprio nome já diz). Os editores de texto, as planilhas de cálculo e os gerenciadores de banco de dados são exemplos de aplicativos.

### ***O processo de tradução de um programa escrito em linguagem de alto nível***

Para que um programa escrito em uma linguagem de alto nível possa ser executado pelo computador, precisa passar pelas etapas mostradas na [Figura 1.9](#).



**FIGURA 1.9** Processo de tradução de um programa em linguagem de alto nível.

Inicialmente utilizamos um editor de texto para escrever o programa na linguagem de alto nível desejada. Depois, este programa é traduzido para uma linguagem intermediária (geralmente é a linguagem de montagem) pelo compilador – assim, ainda temos um arquivo de texto. O programa em linguagem de montagem é então traduzido pelo montador, gerando um arquivo em código objeto. O arquivo em código objeto contém algumas das instruções traduzidas para linguagem de máquina, mas também tem referências à biblioteca que são posteriormente resolvidas pelo ligador, gerando o código executável. O código executável é um código binário pronto para o computador executar.



## O Que Vem Depois

Vimos neste capítulo a origem de alguns conceitos que ainda utilizamos no nosso dia a dia e os principais fatos e máquinas que marcaram os primórdios da computação, assim como vimos também os conceitos fundamentais de hardware e software. Em seguida, pudemos realizar uma breve viagem pelas diversas gerações dos computadores. Com isso, terminamos a abordagem dos conceitos básicos. No próximo capítulo, vamos ver um pouco sobre os sistemas de numeração e como os dados numéricos podem ser representados de diferentes formas. Vamos lá! Bons estudos!!

---

## CAPÍTULO 2

---

# Sistemas de numeração

---

### Objetivos do capítulo

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

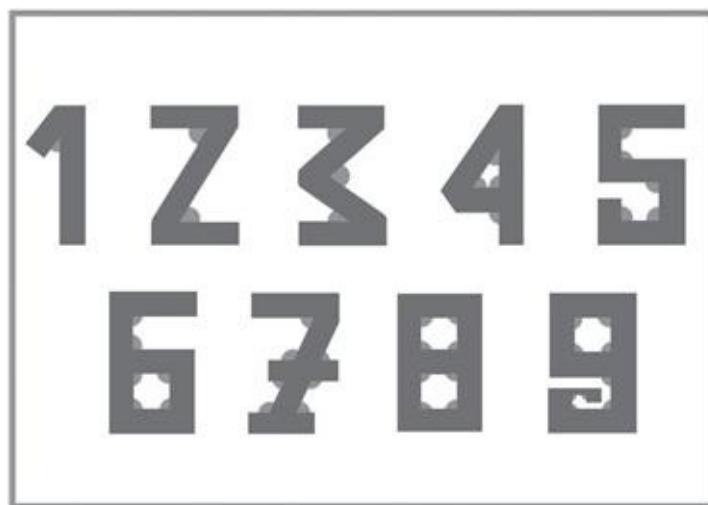
- Representar os valores em diversas bases numéricas.
- Compreender o processo de conversão entre bases e efetuar estas conversões.
- Decidir quais métodos de conversão utilizar, dadas as características dos valores a serem convertidos.



### Apresentação

Você já deve ter se questionado sobre o motivo de os números arábicos serem do jeito que são. Se não, agora é um bom momento para entender isso.

Você pode deduzir o motivo ao observar a figura a seguir.



A forma como conhecemos e utilizamos os números hoje surgiu exatamente dessas formas anteriores, que procuravam traduzir a quantidade de ângulos

existentes em cada figura. Por exemplo, o número dois, como pode ser visto na ilustração, possui dois ângulos; o número 3, três ângulos, e assim por diante.

Assim como a forma e a utilização dos números foram padronizadas em algum momento da história, a representatividade e a forma de expressão também foram estabelecidas.

Ao utilizarmos 10 dígitos diferentes, chegamos na expressão dos valores na forma que utilizamos hoje (sistema decimal). Agora, se em vez de utilizarmos 10 dígitos só utilizarmos dois, os valores ainda poderão ser expressos, mas de outra forma – usando outra combinação. Teoricamente, dizemos que os números são expressos em outra base numérica.

Neste capítulo veremos essas várias formas de padronização e como podemos migrar de uma base numérica (ou convertê-la) para outra. Será bem divertido... vamos lá!

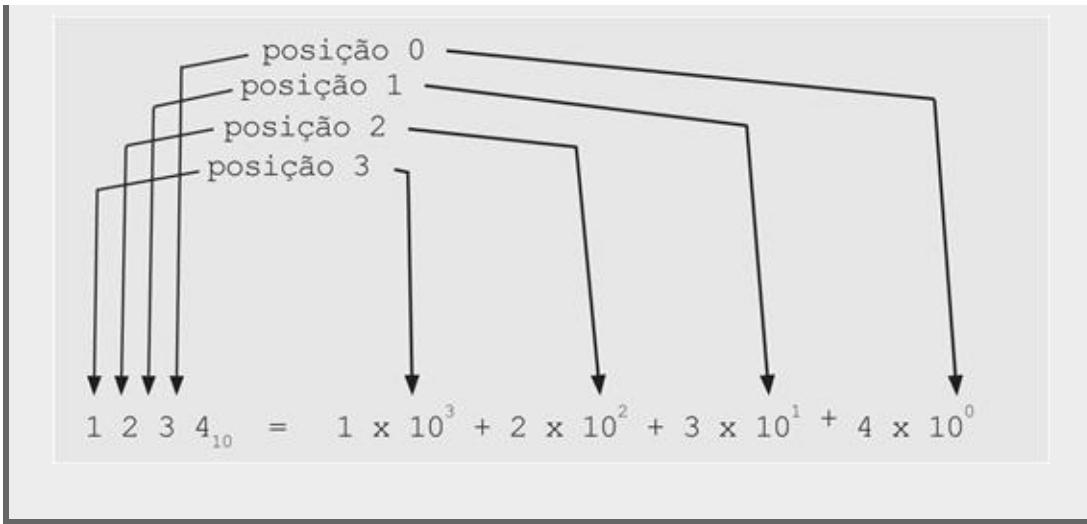
## Fundamentos

### ***Representação posicional de um número***

Os números naturais (inteiros e sem sinal) podem ser representados pela notação posicional. Esta notação permite que qualquer número **n**, em qualquer base de numeração **b**, possa ser escrito como:

$$n = \sum_{i=0}^{i=m-1} d \times b^i$$

Na expressão, **d** é o dígito que ocupa a *i*-ésima posição do número **n**, e **m** é o número de dígitos do número. A representação posicional do número 1234, na base 10, por exemplo, seria:



Neste caso,  $n = 1234$ ,  $b = 10$ ,  $m = 4$  e  $d$  admite os valores dos dígitos que formam o número (1, 2, 3 e 4, com as respectivas posições 3, 2, 1 e 0).

Vamos ver se você guardou essa informação: como ficaria a representação do número 324 na base 10?

## Sistemas De Numeração Binária, Octal E Hexadecimal

Um sistema de numeração permite que por meio de formas e símbolos (algarismos) possamos representar as quantidades (números) e efetuar as operações aritméticas (soma, subtração, multiplicação e divisão) com elas.

A quantidade de algarismos utilizados para representar um número nos diz em que base este número está. Por exemplo, o sistema decimal (base 10), que vem sendo utilizado pela humanidade há muito tempo, faz uso de 10 algarismos: 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9, para representar os números. Analogamente, o sistema binário utiliza dois algarismos: 0 e 1.

Iremos estudar os sistemas de numeração binário (base 2), octal (base 8) e hexadecimal (base 16). O sistema binário é importante, pois, como vimos anteriormente, as informações dentro do computador são codificadas. Os outros dois sistemas são úteis para o programador na representação dos números.

### ***Sistema de numeração binária***

Como já sabemos, o sistema de numeração binária (base 2) utiliza dois símbolos para a representação dos números, os algarismos 0 e 1.

Como em qualquer base, só podemos representar com um algarismo números que vão de zero até a *base-1*; no sistema binário, apenas os valores que pertencem ao intervalo [0..1] podem ser representadas com somente um dígito. Para representar os demais números, será necessário utilizarmos combinações dos algarismos, formando os números de maneira adequada, analogamente ao sistema decimal. Por exemplo:

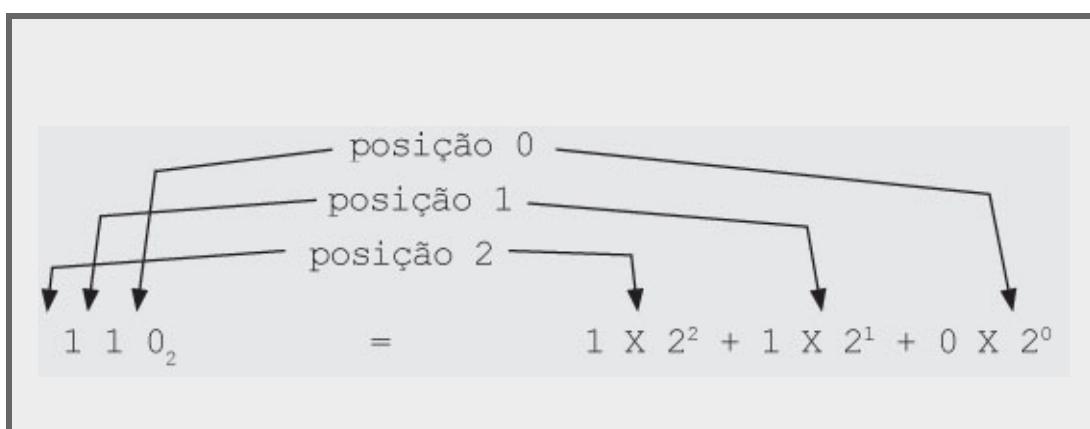
---

**Tabela 2.1**  
**Sistema decimal e sistema binário**

---

Números decimais	Representação binária
Zero	0
Um	1
Dois	10
Três	11
Quatro	100
Cinco	101
Seis	110
Sete	111
Oito	1000
...	...

Além do sistema decimal, podemos representar um número do sistema binário utilizando sua notação posicional. Por exemplo:



Como saber qual quantidade representa esse número? Como estamos acostumados ao sistema decimal, só temos a noção deste valor quando ele é representado na base 10. Portanto, devemos converter o número  $10010_2$  para base 10, e isso é feito efetuando as operações da notação posicional, como mostrado a seguir:

$$\begin{aligned}
 10010_2 &= 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\
 10010_2 &= 1 \times 16 + 0 \times 0 + 0 \times 4 + 1 \times 2 + 0 \times 1 \\
 10010_2 &= 16 + 0 + 0 + 2 + 0 \\
 10010_2 &= 18_{10}
 \end{aligned}$$

## ***Sistema de numeração octal***

O sistema de numeração octal (base 8) utiliza oito símbolos para a representação dos números: os algarismos de 0 a 7. Neste sistema, representamos com um algarismo números que vão de 0 a 7. Para valores maiores que 7, devemos combinar os algarismos de maneira adequada. Por exemplo:

---

**Tabela 2.2**

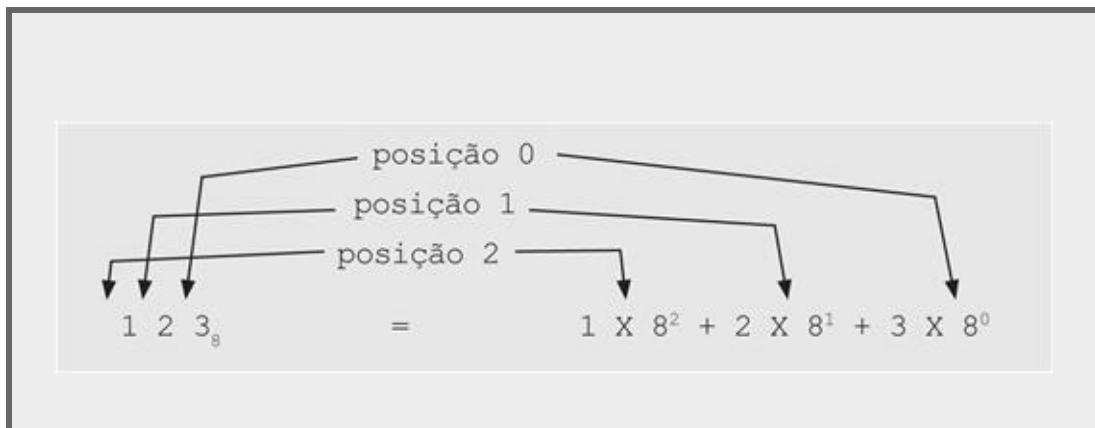
**Sistema decimal e sistema octal**

---

Números decimais	Representação octal
Zero	0
Um	1
Dois	2
Três	3
Quatro	4
Cinco	5
Seis	6
Sete	7
Oito	10

Nove	11
Dez	12
...	...

A utilização da notação posicional para o sistema octal pode ser vista no exemplo a seguir:



Para saber qual valor representa esse número, vamos convertê-lo para o sistema decimal, efetuando as operações da notação posicional na base 10:

$$\begin{aligned}
 123_8 &= 1 \times 8^2 + 2 \times 8^1 + 3 \times 8^0 \\
 123_8 &= 1 \times 64 + 2 \times 8 + 3 \times 1 \\
 123_8 &= 64 + 16 + 3 \\
 123_8 &= 83_{10}
 \end{aligned}$$

## Sistema de numeração hexadecimal

O sistema de numeração hexadecimal (base 16) utiliza 16 símbolos para a representação dos números: os algarismos de 0 a 9 e as letras maiúsculas de A a F. As letras A, B, C, D, E e F são símbolos utilizados para representar os valores

10, 11, 12, 13, 14 e 15, respectivamente. Nesta base também só podemos representar com um símbolo os valores que vão de 0 a 15 (base -1). Para valores maiores que 15, os algarismos e letras são combinados da maneira adequada. Por exemplo:

---

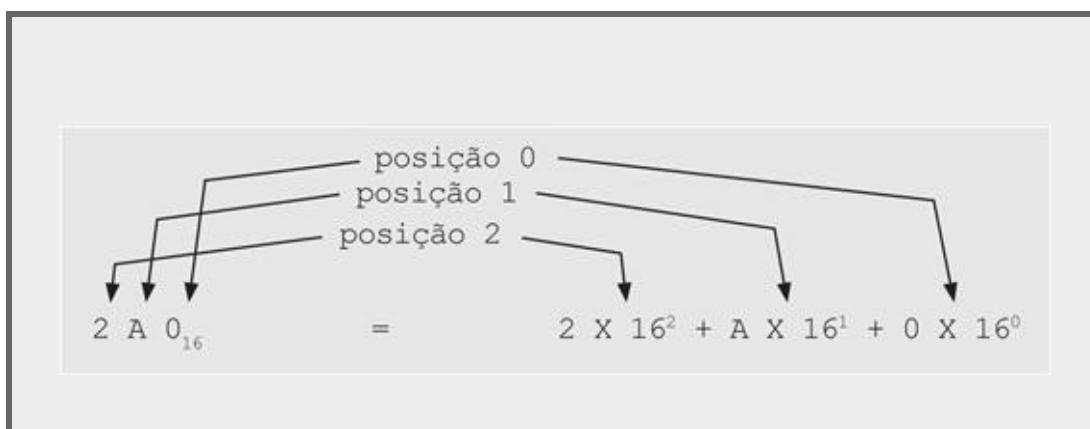
**Tabela 2.3**  
**Sistema decimal e sistema hexadecimal**

---

Números decimais	Representação hexadecimal
Zero	0
Um	1
Dois	2
Três	3
Quatro	4
Cinco	5
Seis	6
Sete	7
Oito	8
Nove	9
Dez	A
Onze	B
Doze	C
Treze	D
Catorze	E
Quinze	F

Dezesseis	10
...	...
Vinte	14
...	...
Trinta e dois	20
...	...

A representação em notação posicional do número  $2A0_{16}$ , por exemplo, é:

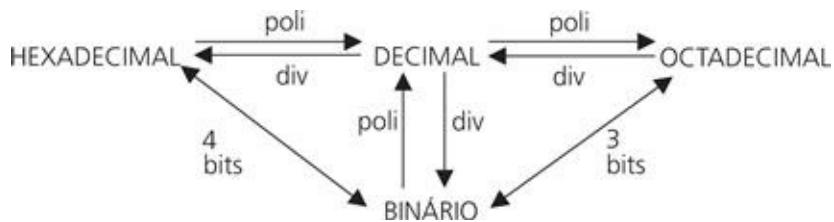


Já o valor representado por esse número é:

$$\begin{aligned}
 2A0_{16} &= 2 \times 16^2 + A \times 16^1 + 0 \times 16^0 \\
 2A0_{16} &= 2 \times 256 + 10 \times 16 + 0 \times 1 \\
 2A0_{16} &= 512 + 160 + 0 \\
 2A0_{16} &= 672_{10}
 \end{aligned}$$

## **Conversão de números entre as diversas bases**

Existem três tipos de procedimentos para efetuar as conversões entre as diversas bases: divisões sucessivas pela base pretendida; utilização do polinômio posicional e agrupamento de bits com a conversão segmentada. Podemos utilizar o esquema ilustrado na [Figura 2.1](#) para efetuar estas conversões.



**FIGURA 2.1** Esquema de procedimentos para conversões entre as bases de numeração.

A seguir, veremos cada uma dessas conversões em mais detalhes.

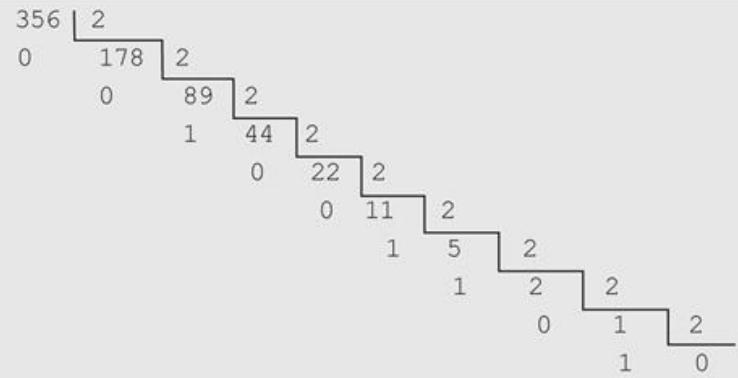
## **Conversão de um número na base decimal para qualquer outra base**

Para converter um número na base decimal para qualquer outra base, basta fazermos divisões sucessivas do número pela base desejada até o quociente ser zero. Os restos das divisões, da última até a primeira, formarão o número na referida base.

### **Exemplo:**

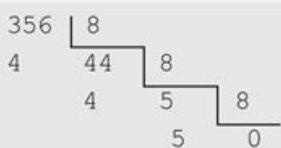
Converter o número  $356_{10}$  para as bases 2, 8 e 16.

- $356_{10}$  para a base 2



$$356_{10} = 101100100_2$$

b)  $356_{10}$  para a base 8



$$356_{10} = 544_8$$

c)  $356_{10}$  para a base 16

$$\begin{array}{r}
 356 \quad | \quad 16 \\
 4 \quad 22 \quad | \quad 16 \\
 \quad \quad 6 \quad 1 \quad | \quad 16 \\
 \quad \quad \quad 1 \quad 0
 \end{array}$$

$$356_{10} = 164_{16}$$

## **Conversão de um número em qualquer base para a base decimal**

Como vimos anteriormente, para converter um número escrito em uma base diferente da decimal para a base decimal basta fazermos as operações indicadas usadas na representação posicional deste número.

### **Exemplos:**

1. Converter o número  $101100100_2$  para a base 10.

$$\begin{aligned}
 101100100_2 &= 1 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + \\
 &= +0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = \\
 &= 256 + 0 + 64 + 32 + 0 + 0 + 4 + 0 + 0 = \\
 &= 356_{10}
 \end{aligned}$$

2. Converter o número  $544_8$  para a base 10.

$$\begin{aligned}544_8 &= 5 \times 8^2 + 4 \times 8^1 + 4 \times 8^0 = \\&= 5 \times 64 + 4 \times 8 + 4 \times 1 = \\&= 320 + 32 + 4 = \\&= 356_{10}\end{aligned}$$

3. Converter o número  $164_{16}$  para a base 10.

$$\begin{aligned}164_{16} &= 1 \times 16^2 + 6 \times 16^1 + 4 \times 16^0 \\&= 1 \times 256 + 6 \times 16 + 4 \times 1 \\&= 256 + 96 + 4 \\&= 356_{10}\end{aligned}$$

### ***Conversão de um número na base binária para uma base que é potência de 2***

Para converter um número escrito na base binária para uma base que seja potência de 2, basta agrupar os bits de **n** em **n** (considerando  $\mathbf{n} = \log_2 \mathbf{base}$ ) da direita para a esquerda, e transformar cada grupo no respectivo algarismo da nova base.

## **Exemplos:**

### **1. Conversão de um número na base binária para a base octal**

Separar os bits do número por grupos de três ( $\log_2 8 = \log_2 2^3 = 3$ ) da direita para a esquerda; cada grupo deverá ser convertido para um algarismo de base 8.

$$\begin{aligned}101100100_2 &= 101 \quad 100 \quad 100_2 = \\&= 5 \quad 4 \quad 4_8 = \\&= 544_8\end{aligned}$$

### **2. Conversão de um número da base binária para a base hexadecimal**

Separar os bits do número por grupos de quatro ( $\log_2 16 = \log_2 2^4 = 4$ ) da direita para a esquerda; cada grupo deverá ser convertido para um algarismo de base 16.

$$\begin{aligned}101100100_2 &= 0001 \quad 0110 \quad 0100_2 = \\&= 1 \quad 6 \quad 4_{16} = \\&= 164_{16}\end{aligned}$$

***Conversão de um número em uma base que é potência de***

## **2 para a base binária**

Para converter um número escrito em uma base que é potência de 2 para a base binária, cada algarismo deste número deve ser convertido para a base desejada; assim, cada um deles formará um número binário com tantos bits quanto for o **log<sub>2</sub> da base**. O número final será o encadeamento destes números binários na mesma ordem dos algarismos.

### **Exemplos:**

#### 1. Conversão de um número na base octal para a base binária

Cada dígito do número deverá ser convertido em um número binário de 3 bits ( $\log_2 8 = \log_2 2^3 = 3$ ). Ao unir este conjunto de bits, teremos o resultado final.

$$\begin{aligned} 544_8 &= 5 \quad 4 \quad 4_8 = \\ &= 101 \quad 100 \quad 100_2 = \\ &= 101100100_2 \end{aligned}$$

#### 2. Conversão de um número na base hexadecimal para a base binária

Cada dígito do número deverá ser convertido em um número binário de 4 bits ( $\log_2 16 = \log_2 2^4 = 4$ ). Ao unir este conjunto de bits, teremos o resultado final.

$$\begin{aligned} 164_{16} &= 1 \quad 6 \quad 4_{16} = \\ &= 0001 \quad 0110 \quad 0100_2 = \\ &= 101100100_2 \end{aligned}$$

A [Tabela 2.4](#) nos mostra um resumo das regras de conversão entre as bases, e a [Tabela 2.5](#) exemplifica como se apresentam os números de 0 a 20.

---

### Tabela 2.4

#### Conversão de bases

---

Conversão	Procedimento
Decimal → Binário	Divisões sucessivas por 2 até se obter zero no quociente. Os restos das divisões lidos da direita para a esquerda formarão o número binário.
Binário → Decimal	Soma dos termos da notação posicional do número; cada termo é um produto do bit com a base 2, com expoente que corresponde à posição do bit no número.
Hexadecimal → Binário	Formar grupos de quatro bits para cada dígito hexadecimal, segundo seu valor.
Binário → Hexadecimal	Compactar cada grupo de quatro dígitos binários em um único dígito hexadecimal da direita para a esquerda, segundo seu valor.
Decimal → Octal	Divisões sucessivas por 8 até se obter zero no quociente. Os restos das divisões lidos da direita para a esquerda formarão o número octal.
Octal → Decimal	Soma dos termos da notação posicional do número; cada termo é um produto do dígito octal com a base 8, com um expoente que corresponde à posição do dígito no número.
Decimal → Hexadecimal	Divisões sucessivas por 16 até se obter zero no quociente. Os restos das divisões lidos da direita para a esquerda formarão o número hexadecimal.
Hexadecimal → Decimal	Soma dos termos da notação posicional do número; cada termo é um produto do dígito hexadecimal com a base 16, com um expoente que corresponde à posição do dígito no número.

---

### Tabela 2.5

#### Exemplo de números de 0 a 20 escritos nas bases decimal, binária, octal e hexadecimal

---

Decimal	Binário	Octal	Hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B

12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	19	11
18	10010	21	12
19	10011	22	13
20	10100	23	14

## Representação De Números

### *Representação de números inteiros sem sinal*

Os números naturais, inteiros e sem sinal, são representados no computador

como uma palavra binária. Vimos anteriormente que o tamanho (número de bits) de uma palavra depende do processador que estamos usando. Portanto, se estivermos trabalhando com um processador de tamanho de palavra de 8 bits, o número 135 será representado por  $10000111_2$ . Este mesmo número, em um processador com palavra de 16 bits, será representado por  $0000000010000111_2$ .

A [Tabela 2.6](#) mostra que, ao considerarmos uma palavra de  $n$  bits, poderemos representar qualquer número decimal dentro do intervalo  $[0..2^n - 1]$ . Assim, se trabalharmos com uma palavra de 8 bits, o intervalo será  $[0..255]$ . A tabela também mostra a relação de  $n$  com o intervalo de representação.

---

**Tabela 2.6**  
**Relação de  $n$  com o intervalo de representação**

---

Número de bits ( $n$ )	Intervalo de representação
1	$[0 .. 2^1 - 1] = [0 .. 1]$
2	$[0 .. 2^2 - 1] = [0 .. 3]$
3	$[0 .. 2^3 - 1] = [0 .. 7]$
4	$[0 .. 2^4 - 1] = [0 .. 15]$
5	$[0 .. 2^5 - 1] = [0 .. 31]$
6	$[0 .. 2^6 - 1] = [0 .. 63]$
7	$[0 .. 2^7 - 1] = [0 .. 127]$
8	$[0 .. 2^8 - 1] = [0 .. 255]$
9	$[0 .. 2^9 - 1] = [0 .. 511]$
10	$[0 .. 2^{10} - 1] = [0 .. 1023]$

### ***Representação em sinal e magnitude de números inteiros sinalizados***

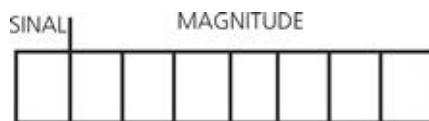
Os números sinalizados têm dois tipos de informações a serem representadas: a magnitude do número, isto é, o seu valor absoluto, e o sinal do número, positivo (+) ou negativo (-). A magnitude é representada como um número binário. Para representar o sinal do número, precisaremos reservar um bit, pois assim teremos duas possibilidades de valor (0 ou 1). Utilizamos, por convenção, o valor 1 para o bit de sinal, se o número for negativo, e 0, se for positivo. Com isso, o número  $65_{10}$ , em um processador de palavra de 8 bits, é representado por:

$$65_{10} = 01000001_2$$

Para uma palavra de n bits, poderemos representar qualquer número dentro do intervalo  $[-2^{-1} .. +2^{n-1}]$ . Assim, se trabalharmos com uma palavra de 8 bits, o intervalo será  $[-127 .. +127]$ . Há duas representações em sinal e magnitude do número 0; se utilizarmos 8 bits, por exemplo, teremos as seguintes representações:

$$\begin{aligned}-0_{10} &= 10000000_2 \text{ ou} \\ +0_{10} &= 00000000_2\end{aligned}$$

Isso não é conveniente em sistemas computacionais porque causa ambiguidade na representação do número 0, o que aumentaria a complexidade dos circuitos lógicos que tratam os números, além de nos fazer desperdiçar um padrão de bits que poderia ser utilizado para representar mais um número. Esta é uma das razões pelas quais a representação não é utilizada nos sistemas computacionais. A [Figura 2.2](#) a seguir esquematiza esta representação.



**FIGURA 2.2** Representação de sinal e magnitude.

## ***Representação em complemento de 1 de números inteiros***

## **sinalizados**

A representação em complemento de 1 também é usada para representar números sinalizados utilizando palavras binárias. Para uma palavra de  $n$  bits, poderemos representar qualquer número em complemento de 1 dentro do intervalo  $[-2^{n-1} .. +2^{n-1}]$ ; assim, se trabalharmos com uma palavra de 8 bits, o intervalo será  $[-127 .. +127]$ . Como podemos ver, o intervalo de números que podem ser representados é o mesmo do sinal e da magnitude, mas a vantagem é que as operações de adição e subtração binária podem ser feitas utilizando apenas a adição, e, por isso, quando da implementação dos circuitos, teremos apenas um circuito somador para executar tanto a adição como a subtração.

Para representarmos um número em complemento de 1, temos de inverter todos os bits a partir da representação em complemento de 1 do negativo do número. Note que os números positivos têm a mesma representação em complemento de 1 e em sinal e magnitude.

### **Exemplos:**

1. Dê a representação do número  $-65_{10}$  em complemento de 1.

$$\begin{aligned}-65_{10} &= 11000001 \text{ em sinal e magnitude} \\ +65_{10} &= 01000001 \text{ em sinal e magnitude e complemento de 2}\end{aligned}$$

Invertendo todos os bits de  $01000001$  ( $+65$ ) teremos  $10111110$ , que é a representação em complemento de 1 de  $-65_{10}$ .

Portanto:

$$-65_{10} = 10111110 \text{ em complemento de 2}$$

2. Dê a representação do número  $+65_{10}$  em complemento de 2.

$$-65_{10} = 10111110 \text{ em Complemento de 1 (exemplo anterior)}$$

Invertendo todos os bits de 10111110 teremos 01000001, que é a representação em complemento de 1 de  $+65_{10}$ . É claro que, se este é um número positivo, poderemos utilizar, diretamente, a mesma representação usada em sinal e magnitude.

Portanto:

$$+65_{10} = 01000001 \text{ em Complemento de 1}$$

Podemos observar na [Tabela 2.7](#) da seção seguinte que o zero ainda tem duas representações e que os números positivos são representados da mesma maneira para as três representações.

---

### Tabela 2.7

**Representação de números de 4 bits em sinal e magnitude,**

## complemento de 1 e complemento de 2

---

Binário	Sinal e magnitude	Complemento de 1	Complemento de 2
1111	-7	0	-1
1110	-6	-1	-2
1101	-5	-2	-3
1100	-4	-3	-4

1011	-3	-4	-5
1010	-2	-5	-6
1001	-1	-6	-7
1000	0	-7	-8
0000	0	0	0
0001	+1	+1	+1
0010	+2	+2	+2
0011	+3	+3	+3
0100	+4	+4	+4
0101	+5	+5	+5
0110	+6	+6	+6
0111	+7	+7	+7

## Representação em complemento de 2 de números inteiros sinalizados

A representação em complemento de 2 também é usada para números sinalizados, e possui as seguintes vantagens:

- Tem apenas uma representação para o zero.
- Permite representar um número negativo a mais na faixa de representação, ou seja, considerando uma palavra de  $n$  bits, poderemos representar qualquer número dentro do intervalo  $[-(2^n + 1 + 1) .. + 2^{n-1}]$  (para uma palavra com 8

bits, o intervalo será  $[-128 \dots +127]$ ).

- Utiliza apenas um circuito somador para soma e subtração, com base na propriedade da álgebra, segundo a qual  $A - B = A + (-B)$ .

Para representarmos um número em complemento de 2, temos de inverter todos os bits e somar 1, a partir da representação em complemento de 2 do negativo do número. Como na representação em complemento de 1, aqui os números positivos também têm a mesma representação que em sinal e magnitude.

### **Exemplos:**

1. Dê a representação do número  $-65_{10}$  em complemento de 2.

$$\begin{aligned}-65_{10} &= 11000001 \text{ em sinal e magnitude} \\ +65 &= 01000001 \text{ em sinal e magnitude e em complemento de 2}\end{aligned}$$

Invertendo todos os bits de 01000001 (+65) teremos 1011110, e depois de somar 1 o resultado será 1011111, que é a representação em complemento de 2 de  $-65_{10}$ .

Portanto:

$$-65_{10} = 10111111 \text{ em complemento de 2}$$

2. Dê a representação do número  $+65_{10}$  em complemento de 2.

$$-65_{10} = 10111111 \text{ (exemplo anterior)}$$

Invertendo todos os bits de 10111111 teremos 01000000, e depois de somar 1 o resultado será 01000001, que é a representação em complemento de 2 de  $+65_{10}$ . É claro que, se esse é um número positivo, podemos utilizar, diretamente, a mesma representação usada em sinal e magnitude.

Portanto:

$$+65 = 01000001 \text{ em sinal e magnitude e em complemento de 2}$$

Podemos observar na Tabela 2.7 (mais adiante) que com quatro bits conseguimos representar o  $-8$ , o que não é possível nas representações em complemento de 1 e nem na de sinal e magnitude.

Um número sinalizado **n** de **m** bits, representado em complemento de 2, pode também ser escrito como:

$$n = -(d_{m-1} \times 2^{m-1}) + (d_{m-2} \times 2^{m-2}) + \dots + (d_1 \times 2^1) + (d_0 \times 2^0)$$

Nessa equação,  $d_i$  é o i-ésimo dígito do número  $n$ .

Essa equação serve para converter um número em complemento de 2 para o sistema decimal.

Já o exemplo a seguir mostra a conversão (utilizando a equação anterior) de um número representado em complemento de 2 para o sistema decimal.

$$\begin{aligned}10111111_2 &= -(1 \times 2^7) + (0 \times 2^6) + (1 \times 2^5) + \\&(1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^0) = \\&= -128 + 0 + 32 + 16 + 8 + 4 + 2 + 1 = -65_{10}\end{aligned}$$

### A representação BCD – *binary coded decimal*

A representação BCD (*Binary Coded Decimal*) permite representar cada algarismo de um número decimal por um binário tomado separadamente, utilizando quatro bits.

Considere a representação do número  $1234_{10}$  em BCD:

$$1 \ 2 \ 3 \ 4_{10} = 0001 \ 0010 \ 0011 \ 1000 \text{ em BCD}$$

Nesse caso, cada um dos algarismos decimais são transformados no código BCD correspondente.

A Tabela 2.8 a seguir mostra o código BCD correspondente a cada dígito

decimal.

---

**Tabela 2.8**  
**Representação BCD (*binary coded decimal*)**

---

Dígito decimal	Código BCD	Dígito decimal	Código BCD
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

A representação BCD é útil quando necessitamos manter a precisão em resultados de cálculos matemáticos - precisão que se perderia pelo arredondamento de outras representações. Isto evita também um possível *overflow* (transbordamento, que será explicado no próximo capítulo) decorrente de outras representações e situações.

## Conversão De Números Fracionários

### ***Conversão de um número fracionário no sistema de numeração decimal para o sistema binário***

Antes de mostrarmos como um número fracionário é representado em um sistema computacional, vamos ver como se faz a conversão de números fracionários decimais em números fracionários binários e vice-versa.

A conversão de um número fracionário decimal para o sistema binário é feita da seguinte maneira:

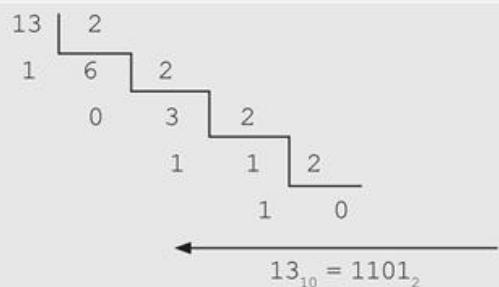
1. Com a parte inteira utilizamos o método das divisões sucessivas, visto anteriormente.
2. Com a parte fracionária, utilizamos o método das multiplicações sucessivas.

## **Exemplo:**

Representar no sistema binário os números:

a) 13,25

Parte inteira =  $13_{10}$



Parte fracionária =  $0,25_{10}$

The diagram illustrates the conversion of the decimal fraction 0,25 to binary. It shows two rows of multiplication by 2. The first row starts with 0,25 and multiplies by 2 to get 0,50. The second row starts with 0,50 and multiplies by 2 to get 1,00. An arrow points from the result 1,00 to the equation  $0,25_{10} = 0,01_2$ .

$0,25$   
 $\times 2$   
 $\hline$   
 $0,50$   
 $\times 2$   
 $\hline$   
 $1,00$

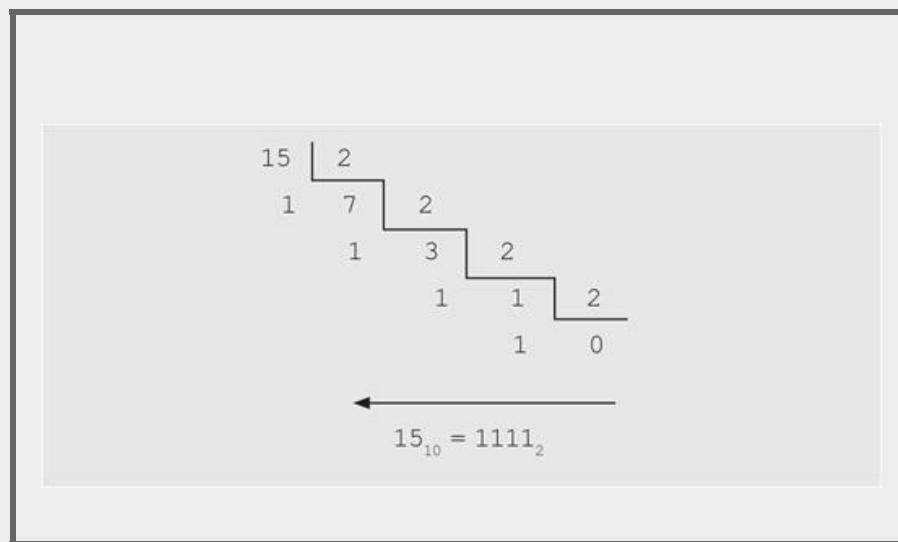
$0,25_{10} = 0,01_2$

Como podemos ver neste exemplo, a parte fracionária é sucessivamente multiplicada por 2, separando-se sempre a parte inteira. A condição de parada é o valor zero na parte fracionária ou quando completarmos o número de bits da precisão desejada. A parte fracionária no sistema binário será formada pelas partes inteiras dos resultados das multiplicações, lidas de cima para baixo.

Portanto,  $13,25_{10} = 1101,01_2$

b)  $15,43_{10}$  com cinco dígitos de precisão.

Parte inteira =  $15_{10}$



Parte fracionária =  $0,43_{10}$

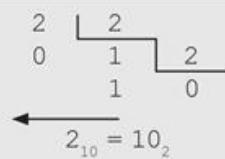
$$\begin{array}{r}
 0,43 \\
 \times 2 \\
 \hline
 0,86 \\
 \times 2 \\
 \hline
 1,72 \\
 \times 2 \\
 \hline
 1,44 \\
 \times 2 \\
 \hline
 0,88 \\
 \times 2 \\
 \hline
 1,76
 \end{array}$$

$0,4310 = 0,011012$  (com cinco dígitos de precisão)

Portanto,  $15,43_{10} = 1111,01101_2$  (com cinco dígitos de precisão)

c)  $2,1_{10}$ .

Parte inteira =  $2_{10}$



Parte Fracionária =  $0,2_{10}$

$$\begin{array}{r}
 \begin{array}{r}
 0,2 \\
 \times 2 \\
 \hline
 0,4 \\
 \times 2 \\
 \hline
 0,8 \\
 \times 2 \\
 \hline
 1,6 \\
 \times 2 \\
 \hline
 1,2
 \end{array}
 &
 \begin{array}{r}
 1,2 \\
 \times 2 \\
 \hline
 0,4 \\
 \times 2 \\
 \hline
 0,8 \\
 \times 2 \\
 \hline
 1,6 \\
 \times 2 \\
 \hline
 1,2 \\
 \text{etc.}
 \end{array}
 \end{array}$$

$$0,1_{10} = 0,00110011\dots_2 \text{ (dízima periódica)}$$

Portanto,  $2,1_{10} = 10,00110011\dots_2$ , que é uma dízima periódica.

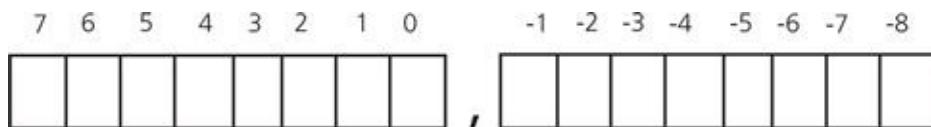
### **Conversão de um número fracionário no sistema de numeração binária para o sistema decimal**

Da mesma forma que um número inteiro, um número fracionário também pode ser escrito em sua representação posicional. Portanto, um número fracionário pode ser escrito da seguinte forma:

$$n = \sum_{i=-j}^{i=m-1} d_i \times b^i$$

Nas equações anteriores, **d** é o dígito que ocupa a *i*-ésima posição do número **n**, **m** é o número de dígitos da parte inteira e **j** é o número de dígitos da parte fracionária do número.

A [Figura 2.3](#) ilustra a posição dos bits em um número fracionário:



**FIGURA 2.3** Posição dos bits em um número fracionário.

Vejamos, por exemplo, a representação posicional do número  $123,45_{10}$ :

$$n = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2}$$

O número  $101,011_2$ , na base binária, pode ser escrito como:

$$n = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

Portanto, para converter um número fracionário do sistema binário para o sistema decimal, basta representá-lo em sua notação posicional e efetuar a soma dos fatores na base 10.

### **Exemplo:**

Que número no sistema decimal é representado pelo número  $101,011_2$ ?

$$101,011_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

Executando as operações, no sistema decimal, o número será:

$$\begin{aligned} n &= 1 \times 4 + 0 \times 2 + 1 \times 1 + 0 \times 0,5 + 1 \times 0,25 + 1 \times 0,125 \\ n &= 4 + 1 + 0,25 + 0,125 = 5,375_{10} \end{aligned}$$

Portanto,  $101,011 = 5,375_{10}$

## **Representação De Números Em Ponto Flutuante**

Para a representação de números fracionários, também chamada de representação em ponto flutuante, é necessário representarmos a mantissa, o sinal da mantissa, o expoente e o sinal do expoente. A padronização desta representação é dada pelo padrão IEEE 754, que, por ser um padrão, tem a função de uniformizar a representação destes números.

O padrão IEEE 754 parte do princípio segundo o qual um número em ponto flutuante tem de estar representado na forma:

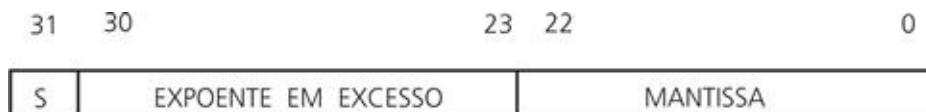
$$n = (-1)^s \times 1.m \times 2^e$$

Na equação, o número a ser representado é **n**, a mantissa normalizada é **m** e o expoente é representado por **e**.

Normalizar a mantissa significa deslocar a vírgula do número para a esquerda ou para a direita de maneira que fique apenas um dígito significativo antes da vírgula. Esse 1 antes da vírgula não é representado no padrão IEEE 754, pois partimos do princípio de que ele sempre existe. Ao deslocarmos a vírgula para a esquerda ou a direita, é necessário somar ou subtrair do expoente o número referente à quantidade de deslocamentos feitos. Por exemplo:

$$101,11001_2 = 101,11001_2 \times 2^0 = 1,0111001_2 \times 2^2$$

A [Figura 2.4](#) mostra o padrão IEEE 754 para a precisão simples. O padrão utiliza uma palavra de 32 bits. O bit mais significativo é o sinal da mantissa, e os 8 bits subsequentes correspondem ao expoente em excesso 127 (polarização); os demais estão relacionados à parte da mantissa normalizada encontrada à direita da vírgula.



**FIGURA 2.4** Representação de números de ponto flutuante – padrão IEEE 754 – precisão simples.

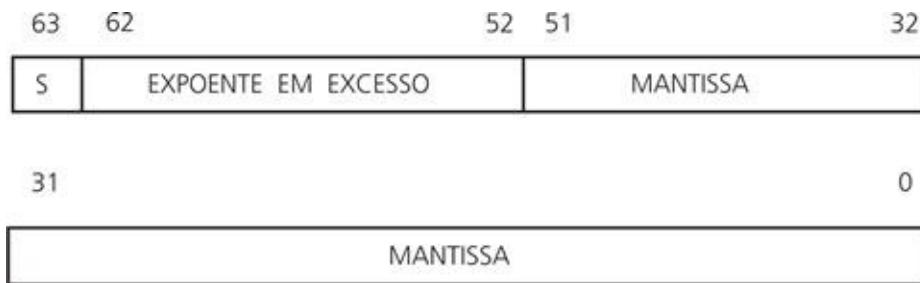
A polarização do expoente para a precisão simples é 127, ou seja, o expoente é

representado como o expoente real do número somado a 127. Com esta polarização é possível representar os expoentes reais no intervalo [-127 .. + 128], e com isso temos que os expoentes polarizados estarão no intervalo [0..255]. Por isso, precisaremos de 8 bits para representar o expoente em excesso. A polarização é usada para facilitar as operações de ponto futuante feitas pelos circuitos, pois ela permite que o expoente seja manipulado apenas como número positivo.

Tomando o expoente polarizado por **exp**, podemos interpretar um número nessa representação da seguinte maneira:

- Se  $0 < \text{exp} < 255$ , então  $n = (-1)^s \times 1.m \times 2^{\text{exp}-127}$
- Se  $\text{exp} = 0$  e  $m \neq 0$ , então  $n = (-1)^s \times 0.m \times 2^{-126}$
- Se  $\text{exp} = 0$  e  $m = 0$ , então  $n = (-1)^s \times 0.0 \times 2^{-126} = \text{zero}$
- Se  $\text{exp} = 255$ , então  $n = \pm \infty$

A [Figura 2.5](#) mostra o padrão para a precisão dupla, que utiliza 64 bits. A precisão dupla permite representar um número com maior precisão, pois tem mais bits para a mantissa, além de permitir também a representação de um número maior ou menor dependendo do expoente, pois também tem mais bits para a representação do expoente. Neste caso, como temos 11 bits para o expoente, a polarização é de 1023, portanto, podemos representar os expoentes reais no intervalo [-1023 .. +1024], e os expoentes polarizados estarão no intervalo [0 .. 2047]



**FIGURA 2.5** Representação de números de ponto futuante – padrão IEEE 754 – precisão dupla.

Um número nessa representação pode ser interpretado da seguinte maneira:

- Se  $0 < \text{exp} < 2047$ , então  $n = (-1)^s \times 1, m \times 2^{\text{exp}-1023}$
- Se  $\text{exp} = 0$  e  $m \neq 0$ , então  $n = (-1)^s \times 0, m \times 2^{-1022}$
- Se  $\text{exp} = 0$  e  $m = 0$ , então  $n = (-1)^s \times 0.0 \times 2^{-1022} = \text{zero}$
- Se  $\text{exp} = 2047$ ,  $n = \pm \infty$

## **Exemplos:**

1. Represente o número  $-0,75_{10}$  no padrão IEEE 754, precisão simples.

$$-0,75_{10} = -0,11_2$$

Normalizando  $\rightarrow -0,11 = -1,1 \times 2^{-1}$

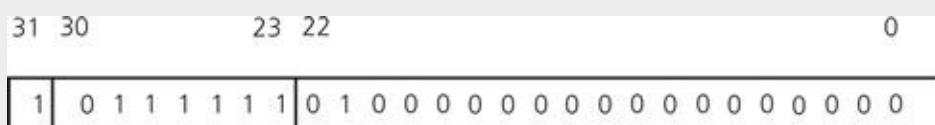
Portanto, temos:

mantissa negativa pois  $s = 1$

$$m = 1$$

$$\text{exp} = -1 + 127 = 126_{10} = 01111110_2$$

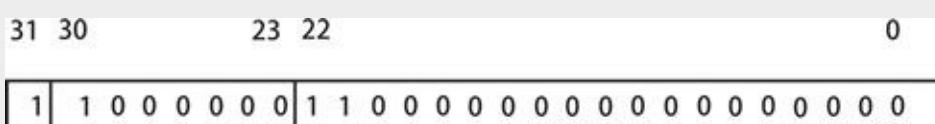
A representação fica:



**FIGURA 2.6** Representação do número  $-0,75_{10}$  no padrão IEEE 754 – precisão simples.

**Observação:** Note que o 1 antes da vírgula, depois de normalizado, não é representado.

2. Determine qual número no sistema decimal representa a palavra na figura a seguir, sabendo que aqui há uma representação de um número de ponto flutuante no padrão IEEE 754 de precisão simples.



**FIGURA 2.7** Representação de número de ponto futuante no padrão IEEE 754 – precisão simples.

Analizando a representação, podemos concluir que:

$$s = 1 \rightarrow \text{mantissa negativa}$$

$$m = 1$$

$$\exp = 10000001_2 = 129$$

$$n = (-1)^1 \times 1,1_2 \times 2^{129-127}$$

$$n = -(1 + 0,5) \times 2^2 = -1,5 \times 4 = -6,0$$



## O Que Vem Depois

Apresentamos neste capítulo os sistemas de numeração utilizados em computação, as respectivas leis de formação de números e a maneira como eles são representados no computador. Veremos a seguir como se faz as operações aritméticas em binário. Deve estar claro que focamos estas operações em binário porque, internamente, o computador trabalha nesta base. Bons estudos!

---

## CAPÍTULO 3

# Aritmética binária

---

### Objetivos do capítulo

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Realizar operações matemáticas elementares com números binários, nas diversas representações vistas no [Capítulo 2](#), como sinal e magnitude, complemento de 2 e codificação BCD.
- Identificar erros nas operações, como o transbordamento (overflow).



### Apresentação

Antes de falarmos sobre as operações aritméticas básicas envolvendo os números binários, seria interessante mostrar como a evolução tecnológica transformou palavras como “cálculo”, que originalmente designava uma mera forma de contagem de ovelhas, em uma das palavras mais importantes na área computacional.



Cálculo vem do latim *calculus*, que significa “pedra”. Os pastores antigamente tinham de controlar a quantidade de ovelhas que eram levadas para pastar, assim como as que retornavam. Isto era importante para que eles não se esquecessem de uma ou outra ovelha, e também servia para verificar se eles não as estavam perdendo para outros pastores ou animais selvagens. Uma das formas de praticar este controle era reservar uma quantidade de pedras correspondente ao número de ovelhas. Assim, surgiu a palavra “calcular”, ou seja, usar pedras para controlar (contar) a quantidade de ovelhas.

Porém, conforme a quantidade de ovelhas aumentava, os pastores viram que era necessária uma tecnologia de contagem mais à mão, e começaram a utilizar os dedos, palavra derivada do latim *digitus*. Assim surgiram os códigos (dígitos), que representavam unidades, dezenas, centenas etc.

Depois disso surgiram os sistemas de numeração e algumas máquinas que vieram auxiliar o ser humano a desenvolver ferramentas cada vez mais efetivas nas tarefas de cálculo e computação.

As tecnologias e as ferramentas evoluíram, mas o método de calcular continua o mesmo – e com aplicabilidade em várias bases, inclusive a binária. As quatro operações aritméticas, adição, subtração, multiplicação e divisão, têm os mesmos princípios, não importando a base em que se trabalha. Por exemplo, na base decimal, quando se soma o 1 ao 9, o resultado passa a ser 0 unidade e 1 dezena, e assim temos o vai-um (*carry-out*). Isto acontece quando o resultado é maior que a base com a qual estamos trabalhando. Analogamente, quando somamos 1 com 1 na aritmética binária, temos o número  $10_2$  ( $2_{10}$ ), ou seja, 0 e vai-um. Em outras palavras, as quatro operações, não importando a base, têm sempre o mesmo algoritmo como princípio.

Vamos agora ver como isso ocorre! Bons estudos!

## Fundamentos

### ***Adição e subtração de números binários não sinalizados***

Em qualquer base que trabalhemos, as operações aritméticas seguirão as mesmas regras às quais estão sujeitas no sistema decimal – isso obviamente considerando a base em que estamos trabalhando. Em outras palavras, fazer uma operação na base binária é a mesma coisa que fazê-lo na base decimal; deve-se somente lembrar que na base binária uma ordem superior de um número tem duas unidades da ordem imediatamente inferior, e não 10, como no sistema decimal.

## **Exemplos:**

1. Tomemos como exemplo a soma de dois números de 8 bits não sinalizados:

$$7_{10} + 6_{10} = 00000111_2 + 00000110_2$$

$$\begin{array}{r}
 & & & & & & \text{vai-um} \\
 & & & & 1 & 1 \\
 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & (7_{10}) \\
 + & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & +(6_{10}) \\
 \hline
 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & (13_{10})
 \end{array}$$

O que podemos observar neste exemplo é que, na operação dos segundos dígitos (da direita para a esquerda) dos números (1 + 1), o resultado é  $10_2$ , ou seja, fica o 0 e vai-um. Na operação dos terceiros dígitos (1 + 1 + vai-um) o resultado é  $11_2$ , ou seja, fica 1 e vai-um.

2. Para a subtração, usaremos também dois números de 8 bits não sinalizados:

$$6_{10} - 5_{10} = 00000110_2 - 00000101_2$$

$$\begin{array}{r}
 & & & & & & & \text{"empréstimo"} \\
 & 0 & 0 & 0 & 0 & 0 & 1 & \cancel{\not{}}^{\phantom{0}10} \\
 - & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
 \hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
 \end{array}
 \quad
 \begin{array}{l}
 (6_{10}) \\
 + (5_{10}) \\
 \hline
 (1_{10})
 \end{array}$$

Neste caso, considerando os bits menos significativos, não podemos subtrair 1 de 0, então “emprestamos” o 1 da posição à esquerda; como ele é de uma ordem superior, esse 1 vem como  $10_2$ , ao qual subtraindo 1 resulta 1. O 1 à esquerda da ordem menos significativa se torna 0.

## **Adição e subtração de números binários sinalizados**

A operação de adição de números binários sinalizados, representados em complemento de 2, é executada da mesma maneira que descrevemos em “Adição e subtração de números binários não sinalizados”, respeitando os sinais dos números. A subtração é feita como uma adição. Os exemplos a seguir ilustram

estes procedimentos.

## **Exemplos:**

## 1. Soma de dois números positivos

$$+7_{10} + (+6_{10}) = 00000111_2 + 00000110_2$$

$$\begin{array}{r}
 & & & & & & \text{vai-um} \\
 & & & & 1 & 1 \\
 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & (+7_{10}) \\
 + & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
 \hline
 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
 & & & & & & & (+13_{10})
 \end{array}$$

## 2. Soma de um número positivo com um número negativo

$$(+7) + (-3) = 00000111 + 11111101$$

$$\begin{array}{r}
 & & & & & & & & \text{vai-um} \\
 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & (+7_{10}) \\
 + & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
 \hline
 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & (+4_{10}) \\
 & & & & & & & & (-3_{10})
 \end{array}$$

O 1 do vai-um na posição do sinal (bit mais significativo) é desprezado.

### 3. Soma de dois números negativos

$$(-7_{10}) + (-3_{10}) = 11111001_2 + 11111101_2$$

$$\begin{array}{r}
 & & & & & & & \text{vai-um} \\
 & 1 & 1 & 1 & 1 & 1 & 1 & \\
 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & (-7_{10}) \\
 + & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & + (-3_{10}) \\
 \hline
 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & (-10_{10})
 \end{array}$$

O 1 do vai-um na posição do sinal (bit mais significativo) é desprezado.

#### 4. Subtração de dois números positivos

Neste caso, a operação a ser realizada é uma soma de um número positivo com um número negativo.

$$+7_{10} - (+3_{10}) = +7_{10} + (-3_{10}) = 00000111_2 + 1111101_2$$

$$\begin{array}{r}
 & & & & & & & \\
 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \text{vai-um} \\
 + & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
 \hline
 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
 \end{array}
 \quad
 \begin{array}{r}
 & & & \\
 & & & (+7_{10}) \\
 & & & + (-3_{10}) \\
 \hline
 & & & (+4_{10})
 \end{array}$$

O 1 do vai-um na posição do sinal (bit mais significativo) é desprezado.

#### 5. Subtração de um número positivo com um número negativo.

Aqui, a operação será uma soma de dois números positivos.

$$+7_{10} - (-6_{10}) = +7_{10} + (+6_{10}) = 00000111_2 + 00000110_2$$

$$\begin{array}{r}
 & & & & & & \\
 & 1 & 1 & & & & \\
 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \text{vai-um} \\
 + & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
 \hline
 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1
 \end{array}
 \quad
 \begin{array}{r}
 & & & \\
 & & & (+7_{10}) \\
 & & & + (+6_{10}) \\
 \hline
 & & & (+13_{10})
 \end{array}$$

#### 6. Subtração de dois números negativos.

Neste exemplo, a operação que será realizada é a soma de um número negativo com um número positivo.

$$-7_{10} - (-6_{10}) = -7_{10} + (+6_{10}) = 11111001_2 + 00000110_2$$

$$\begin{array}{r}
 & & & & & & \\
 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
 & + & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
 \hline
 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
 \end{array}
 \quad
 \begin{array}{r}
 & & & \\
 & & & (-7_{10}) \\
 & & & + (+6_{10}) \\
 \hline
 & & & (-1_{10})
 \end{array}$$

## Multiplicação E Divisão De Números Binários

Assim como a adição e a subtração, as operações de multiplicação e divisão, em qualquer base, também seguem os mesmos algoritmos utilizados na base decimal.

### Exemplos:

## 1. Multiplicação de dois números binários não sinalizados.

$$8_{10} \times 9_{10} = 1000_2 \times 1001_2 = 1001000_2$$

multiplicando	1 0 0 0	(8 <sub>10</sub> )
multiplicador	$\times$ 1 0 0 1	(9 <sub>10</sub> )
	1 0 0 0	
	0 0 0 0	
	0 0 0 0	+
	1 0 0 0	
produto	1 0 0 1 0 0 0	(72 <sub>10</sub> )

## 2. Divisão de dois números binários não sinalizados.

$$74_{10} \div 8_{10} = 1001000_2 \div 1000_2 = 1001_2 \text{ e resto } 0010_2$$

dividendo	(74 <sub>10</sub> ) (8 <sub>10</sub> )	divisor
	1 0 0 1 0 1 0	1 0 0 0
	- 1 0 0 0	1 0 0 1
	0 0 0 1 0	quociente
		(9 <sub>10</sub> )
	1 0 1	
	1 0 1 0	
	- 1 0 0 0	
	0 0 1 0	resto
		(2 <sub>10</sub> )

**Observação:** Na multiplicação de números sinalizados binários, deve-se proceder como se os números não fossem sinalizados. O resultado será positivo se os sinais dos números forem iguais; caso contrário, será negativo. Cabe considerar também o algoritmo de Booth (em [David Patterson e John Hennessy, 2000](#); veja as Referências Bibliográficas), que permite a multiplicação de números sinalizados, representados em complemento de 2, sem a necessidade de se trabalhar com a magnitude e o sinal em separado.

# Erro De Transbordamento Ou Erro De Overflow

Erro de overflow ou erro de transbordamento ocorre sempre quando o número de bits da palavra com a qual estamos trabalhando em uma dada operação aritmética não permite que o resultado seja representado.

Para operações com números não sinalizados, só ocorrerá overflow na adição, uma vez que, se podemos representar os dois termos da operação na subtração, certamente seremos capazes de representar o resultado, pois este será sempre menor que o maior dos termos. A condição para que haja overflow em uma adição de dois números binários não sinalizados é a ocorrência do vai-um (*carry-out*) na posição do bit mais significativo.

Para ilustrar com um exemplo, vamos fazer a seguinte operação, utilizando palavras de 4 bits:

$$\begin{array}{r}
 & 10_{10} + 6_{10} = 1010_2 + 0010_2 \\
 \text{vai-um} & \begin{array}{r}
 \begin{array}{c} 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ + & 0 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 \end{array} & (10_{10}) \\
 & + (6_{10}) \\
 & \hline (16_{10})
 \end{array}
 \end{array}$$

O que podemos ver é que o resultado da operação  $(16_{10})$  não pode ser representado com quatro bits, por isso dizemos que ali ocorreu o overflow ou erro de transbordamento. Para representar o  $(16_{10})$  necessitamos de cinco bits:  $10000_2$ .

Para que o overflow não ocorra na multiplicação, o resultado deverá ser representado por um número de **2n** bits, quando multiplicarmos dois números de **n** bits; nos circuitos multiplicadores, isso é o que acontece na prática. Por fim, cabe notar que na divisão não temos o problema do overflow.

Para números sinalizados em complemento de 2, na adição e subtração, a ocorrência de overflow pode ser detectada de duas maneiras. Podemos analisar, na posição do sinal (MSB), o vem-um (emprestimo – *carry-in*) e o vai-um (*carry-out*). O overflow não ocorrerá caso eles sejam iguais. Se forem diferentes, haverá a ocorrência do erro de transbordamento.

A seguir, alguns exemplos considerando números de 4 bits:

1. Na operação a seguir, o vai-um da posição mais significativa do número é igual ao vem-um, por isso não ocorreu overflow.

$$\begin{array}{r}
 1 & 1 & 1 & 1 \\
 0 & 1 & 1 & 1 \\
 + & 1 & 1 & 0 & 1 \\
 \hline
 0 & 1 & 0 & 0
 \end{array}
 \quad
 \begin{array}{r}
 (+7_{10}) \\
 + (-3_{10}) \\
 \hline
 (+4_{10})
 \end{array}$$

2. Na operação seguinte, o vai-um da posição MSB é igual ao vem-um, por isso ocorreu overflow.

$$\begin{array}{r}
 0 & 0 & 1 & 1 \\
 0 & 0 & 1 & 1 \\
 + & 1 & 0 & 0 & 1 \\
 \hline
 1 & 1 & 0 & 0
 \end{array}
 \quad
 \begin{array}{r}
 (+3_{10}) \\
 + (-7_{10}) \\
 \hline
 (-4_{10})
 \end{array}$$

3. Já na próxima operação, o vai-um da posição MSB do número é diferente do vem-um, por isso ocorreu overflow.

$$\begin{array}{r}
 1 & 0 & 0 & 1 \\
 1 & 0 & 0 & 1 \\
 + & 1 & 1 & 0 & 1 \\
 \hline
 0 & 1 & 1 & 0
 \end{array}
 \quad
 \begin{array}{r}
 (-7_{10}) \\
 + (-3_{10}) \\
 \hline
 (-10_{10})
 \end{array}$$

4. Nessa operação, o vai-um da posição MSB do número é diferente do vêm-um, por isso ocorreu overflow.

$$\begin{array}{r}
 \begin{array}{r}
 \begin{array}{r}
 0 & 1 & 1 & 1 \\
 0 & 1 & 1 & 1 \\
 + & 0 & 0 & 0 & 1 \\
 \hline
 1 & 0 & 0 & 0
 \end{array}
 \end{array}
 \end{array}
 \quad
 \begin{array}{r}
 \begin{array}{r}
 \begin{array}{r}
 (+7_{10}) \\
 + (+1_{10}) \\
 \hline
 (+8_{10})
 \end{array}
 \end{array}
 \end{array}$$

Outro modo de verificarmos se ocorreu ou não overflow em adição e subtração de números em complemento de 2 é a análise dos sinais dos números e do sinal do resultado. Sejam A e B os números e R o resultado. A [Tabela 3.1](#) nos dá as condições de ocorrência de overflow.

---

**Tabela 3.1**  
**Condições de ocorrência de overflow**

---

Operação	Número A	Número B	Resultado R
A + B	$\geq 0$	$\geq 0$	$< 0$
A + B	$< 0$	$< 0$	$\geq 0$
A - B	$\geq 0$	$< 0$	$< 0$
A - B	$< 0$	$\geq 0$	$\geq 0$

## Operações Com Números Em BCD

A operação com números em BCD utiliza os mesmos algoritmos das operações

no sistema decimal.

### Exemplos:

1.  $32_{10} + 24_{10} = 00110010 + 00100100 = 00110110 = 56_{10}$

$$\begin{array}{r} 0011\ 0010 \\ + 0010\ 0100 \\ \hline 0011\ 0110 \end{array}$$

2.  $82_{10} + 39_{10} = 10000010 + 00111001 = 000100100001 = 121_{10}$

$$\begin{array}{r} 1000\ 0010 \\ + 0011\ 1001 \\ \hline 1011 \end{array}$$



Este dígito é maior que 9  
(resultou 11), portanto, para  
representá-lo em BCD, consideramos  
 $11_{10}$  em BCD = 0001 0001, que resultará um "vai-um":

$$\begin{array}{r} 1\ 1\ 1 \\ 1000\ 0010 \\ + 0011\ 1001 \\ \hline 1100\ 0001 \end{array}$$



Este dígito também é maior que 9  
(resultou 12), portanto, para  
representá-lo em BCD, consideramos  
 $12_{10}$  em BCD = 0001 0010, que resultará em:

$$\begin{array}{r} 1000\ 0010 \\ + 0011\ 1001 \\ \hline 1\ 0010\ 0001 \end{array}$$

Este é o resultado final =  $121_{10}$

Na prática, quando um resultado é maior que 9, somase 6 (em binário) ao dígito, para que tenhamos o número codificado novamente em BCD.

No caso anterior:  $1011_2 + 0110_2 = 1\ 0001$ , ou seja, a representação BCD do número  $11_{10}$ .

**Observação:** Somamos o número 6 porque a “distância entre” o número  $10001_2$  e o número  $1011_2$  é 0110, ou seja,

$$17_{10} - 11_{10} = 6_{10}.$$

$$3. 32_{10} - 24_{10} = 00110010 - 00100100 = 00001000 = 08_{10}$$

$\begin{array}{r} 0011 \ 0010 \\ - 0010 \ 0100 \\ \hline \end{array}$  ← Este número ( $2_{10}$ ) é menor que este ( $4_{10}$ ). Como nas operações decimais, vamos “emprestar 1” do dígito à esquerda (dígito de ordem superior).

$\begin{array}{r} 0010 \ 1100 \\ - 0010 \ 0100 \\ \hline 0000 \ 1000 \end{array}$  ← O dígito da ordem superior que era  $3_{10}$  passa a ser  $2_{10}$ , e o da inferior que era  $2_{10}$  passa a ser  $2 + 10 = 12_{10}$

$$4. 15_{10} \times 2_{10} = 00010101 \times 0010 = 00110000 = 30_{10}$$

Multiplicando inicialmente o dígito menos significativo pelo multiplicador:

$$\begin{array}{r}
 0001 \\
 \times 0010 \\
 \hline
 0000 \\
 0001 \\
 0000 \\
 0000 \\
 \hline
 00000010
 \end{array}$$

$$\begin{array}{r}
 0101 \\
 \times 0010 \\
 \hline
 0000 \\
 0101 \\
 0000 \\
 0000 \\
 \hline
 00001010
 \end{array}$$

$$\begin{aligned}
 00001010 &\rightarrow 1010 + 0110 = \\
 &= 1\ 0000
 \end{aligned}$$

vai-um para o dígito mais significativo  
do resultado

Portanto, para o dígito mais significativo teremos:  
0010 + 0001 (vai-um do resultado da multiplicação  
do dígito menos significativo) = 0011

Portanto, para o dígito mais significativo teremos:  
0010 + 0001 (vai-um do resultado da multiplicação  
do dígito menos significativo) = 0011

Portanto, o resultado é 0011 0000.

5.  $31_{10} \div 2_{10} = 00110001 \div 0010 = 00010101$  e resto 0001 =  $15_{10}$  e  
resto  $1_{10}$

A divisão do dígito mais significativo fica:

A divisão do dígito mais significativo fica:

11    10  
 10    1 ← digito mais significativo do  
            
 01              Resultado - 0001

O resto é somado ao dígito menos significativo, lembrando que o resto é de uma ordem superior; portanto, teremos  $1010$  (1 de ordem superior =  $10_{10}$  na ordem inferior) + 0001 (dígito menos significativo do dividendo) = 1011

$\begin{array}{r} 1011 \\ 10 \\ \hline 0011- \end{array}$	$\begin{array}{c} 10 \\ \hline 101 \end{array}$	dígito menos significativo do resultado - 0101 resto - 0001
---	---	--



# O que vem depositar

A Parte 1 deste livro apresentou, além do histórico e evolução dos computadores, importantes conceitos que ajudam a entender as operações aritméticas com números binários, o que nos dá uma ideia de como podem ser os circuitos digitais que implementam estas operações. Na próxima parte deste livro iremos

nos aprofundar nos conceitos de variáveis, funções e circuitos lógicos, o que permitirá nossa iniciação ao mundo complexo do hardware. Mas antes de passar para a segunda parte deste livro, faça uma recapitulação nos três capítulos da Parte 1 e desenvolva os exercícios propostos no site [www.elsevier.com.br/organizacaobasica](http://www.elsevier.com.br/organizacaobasica). Vamos lá!! Bons estudos!

---

## CAPÍTULO 4

# Variáveis e funções lógicas

---

### Objetivos do capítulo

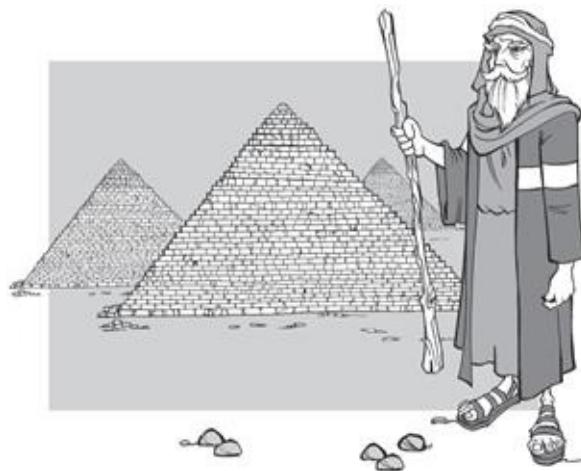
Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Compreender o conceito de variável lógica.
- Compreender o conceito de função lógica.
- Identificar as funções lógicas básicas e derivadas.
- Representar funções lógicas por meio de tabelas-verdade, equações e símbolos.



### Apresentação

Agora, um pouco de matemática! As variáveis são objeto do nosso interesse, e normalmente envolvem números. Funções operam variáveis segundo certas regras.



Essas regras podem ser expressas por uma fórmula matemática que permite o

cálculo de um resultado, que por sua vez também é uma variável. Nas seções a seguir vamos rever esses conceitos e aplicá-los a um campo novo: as variáveis e funções lógicas.

## Fundamentos

### O conceito de variável e função

O conceito de variável e de função nos é familiar pela experiência que temos com a matemática. O campo de definição de uma variável, isto é, o intervalo de valores que uma variável  $x$  pode assumir pode ser especificado de inúmeras maneiras:  $x$  pode variar sobre o campo dos números reais de menos infinito até mais infinito, por exemplo, ou pode ser restrito ao intervalo dos números reais compreendido entre  $-1.000$  e  $+1.000$ ; a variável pode ainda ser restrita a números inteiros de  $0$  a  $100$ , e assim por diante.

Uma função  $f$  é uma regra ou relação pela qual se determina o valor de uma variável dependente  $y$  a partir do valor de uma variável independente  $x$ , e é representada por  $y = f(x)$ .

Seja uma função matemática descrita por  $y = 3x^2 - 2$ . A relação funcional entre  $x$  e  $y$  pode ser dada por:

$x$	$y = f(x)$
0	-2
1	1
2	10
3	25

As variáveis dependente ( $y$ ) e independente ( $x$ ) não precisam ser numéricas como na tabela anterior, mas podem ser descritas por meio de palavras ou sentenças textuais. Veja a seguir dois exemplos.

Acionamento de uma lâmpada	
$x$	$Y = f(x)$
Chave ligada	Luz acesa
Chave desligada	Luz apagada

Cores do semáforo	
x	$y = f(x)$
Verde	Prossiga
Amarelo	Devagar
Vermelho	Pare

As funções são classificadas como contínuas, quando podem assumir quaisquer valores dentro de um dado intervalo, e discretas, caso só possam assumir alguns valores dentro de um conjunto finito.

Em sistemas digitais como computadores encontramos variáveis particulares que só podem assumir dois valores distintos, geralmente denominados verdadeiro ou falso, aos quais se associam os números binários 1 e 0, respectivamente. Este tipo de variável é denominado variável lógica ou binária.

Uma variável lógica é uma variável que tem três propriedades:

- A variável lógica só pode assumir um dentre dois valores possíveis. O exemplo do semáforo define uma variável x que pode assumir três valores, logo ela não é uma variável lógica.
- Os valores são expressos por afirmações declarativas como “aceso”, “apagado”, “ligado”, “desligado”.
- Os dois valores possíveis, expressos por afirmações declarativas, devem ser tais que, com base no raciocínio humano, ou seja, com base na lógica, sejam mutuamente exclusivos. Assim “ligado” exclui “desligado” e vice-versa.

Suponha que o semáforo do exemplo anterior só pudesse estar “verde” ou “vermelho”. Neste caso, a variável x da tabela do semáforo seria uma variável lógica. As hipóteses possíveis seriam:

- “O semáforo está verde”, o que seria representado por “ $x = \text{verde}$ ” ou;
- “O semáforo está vermelho”, representado por “ $x = \text{vermelho}$ ”. Devido à mútua exclusividade, ao afirmarmos que “ $x = \text{vermelho}$ ” também estamos dizendo que “ $x = \text{não verde}$ ”.

Quando consideramos matematicamente a relação funcional entre as variáveis de um problema, não nos interessa o que é representado por elas. Assim, devemos dar nomes aos dois valores possíveis de uma variável lógica para que possamos considerar a variável independentemente do que ela possa representar. Por exemplo:

A	$Y = f(A)$
Verde	Prossiga
Vermelho	Pare
equivalente a	
1	1
0	0

Os valores lógicos 1 e 0 na tabela da direita são atribuídos arbitrariamente: no caso de A, são as cores “verde” e “vermelho”, e no caso da variável Y correspondem aos comandos “prossiga” e “pare”. Normalmente associamos ao valor lógico 1 as ideias de verdadeiro, positivo, ativo, ligado etc., e ao valor lógico 0, as ideias de falso, inativo, desligado etc. A [Tabela 4.1](#) lista diversos valores de variáveis lógicas que podem ser associados aos valores lógicos 1 e 0 genéricos.

---

### Tabela 4.1

#### Exemplos de valores de variáveis lógicas

---

1 Lógico	0 Lógico
Verdadeiro	Falso
Ligado	Desligado
Alto	Baixo
Ativado	Desativado
Aceso	Apagado
SIM	NÃO

Portanto, inicialmente poderemos usar variáveis lógicas para resolver apenas problemas cujas variáveis tenham essa natureza.

Por exemplo:

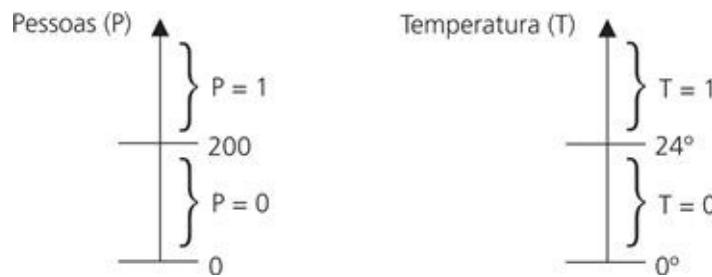
*Um ventilador é “ligado” se houver pessoas dentro de uma sala e a temperatura for alta.*

Nesse exemplo, as variáveis são descritas conforme a tabela a seguir.

Valor declarativo	Valor lógico
Ventilador “desligado”	0
Ventilador “ligado”	1
“Não há” pessoas na sala	0
“Há” pessoas na sala	1

Temperatura “baixa”	0
Temperatura “alta”	1

Observe que, apesar de as variáveis lógicas admitirem apenas dois valores distintos, elas podem representar qualquer coisa (temperatura, pressão, distância, velocidade, tempo etc.), mesmo aquelas que aparentam ter mais de dois valores. No exemplo considerado, quando nos referimos “logicamente” às variáveis “pessoas” e “temperatura”, não estávamos interessados na quantidade ou no valor. O problema se resolve perfeitamente se nos limitarmos em saber se “há” ou “não há” pessoas dentro da sala (parece razoável desligar o ventilador quando houver ZERO pessoa na sala), ou se a temperatura está “alta” ou “baixa” (em relação a uma referência de, digamos, 24º C). Portanto, a lógica de dois valores pode ser adaptada para representar grandezas multivaloradas, bastando definir um limite de decisão, tal como mostrado na [Figura 4.1](#).



**FIGURA 4.1** Definição de variáveis lógicas.

Existem várias funções lógicas, mas apenas umas poucas são necessárias para expressar a maioria dos problemas. Considere a sentença a seguir:

*A iluminação interna de um escritório é ligada se a luminosidade externa cair abaixo de um valor mínimo e houver pessoas dentro do escritório.*

Para transformarmos essa sentença em um problema lógico, devemos associar a cada afirmação uma variável lógica para depois compormos a função que a representa.

Variável A = luminosidade externa. Como A só pode assumir dois valores, vamos definir que:

A = 1 significa que a luminosidade externa está abaixo do valor mínimo (usamos a variável A para detectar a condição de interesse para o problema, ou seja, a situação em que a iluminação externa não é suficiente).

A = 0 significa que a luminosidade externa está acima do valor mínimo.

Variável B = existência de pessoas dentro do escritório. De forma análoga ao caso anterior, podemos estipular que:

B = 1 significa que existem pessoas dentro do escritório.

B = 0 significa que não existem pessoas dentro do escritório.

Variável Y = iluminação interna

Y = 1 significa ligar iluminação interna.

Y = 0 significa desligar iluminação interna.

Ao montar uma tabela com todas as possibilidades, descobriremos que a relação entre as variáveis A e B é tal que:

$$Y = 1 \text{ somente se } (A = 1) \wedge (B = 1)$$

Essa é uma função AND.

A	B	Y	Significado
0	0	0	A iluminação interna não é ligada, pois a iluminação externa é suficiente e não existem pessoas no escritório.
0	1	0	A iluminação interna não é ligada, pois a iluminação externa é suficiente.
1	0	0	A iluminação interna não é ligada, pois não existem pessoas no escritório.
1	1	1	A iluminação interna é ligada, pois a iluminação externa não é suficiente e existem pessoas no escritório.

Vejamos agora outro exemplo:

*O alarme de segurança de uma empresa é acionado se o portão da frente for aberto ou se a porta dos fundos for aberta.*

Variável A = portão da frente:

A = 1 significa que o portão da frente está aberto.

A = 0 significa que o portão da frente está fechado.

Variável B = porta dos fundos:

B = 1 significa que a porta dos fundos está aberta.

B = 0 significa que a porta dos fundos está fechada.

Variável Y = alarme de segurança

Y = 1 significa ligar o alarme de segurança.

Y = 0 significa desligar o alarme de segurança.

A	B	Y	Significado
0	0	0	O alarme de segurança é desligado, pois o portão da frente e a porta dos fundos estão fechados.
0	1	1	O alarme de segurança é ligado, pois a porta dos fundos está aberta.
1	0	1	O alarme de segurança é ligado, pois o portão da frente está aberto.
1	1	1	O alarme de segurança é ligado, pois o portão da frente e a porta dos fundos estão abertos.

Na tabela anterior, percebemos que a relação entre as variáveis A e B é tal que:

$$Y = 1 \text{ se } (A = 1) \text{ OU } (B = 1) \text{ OU } (\text{ambas} = 1)$$

Esse é um exemplo de uma função OR.

Nos exemplos anteriores, as variáveis A e B são denominadas variáveis de entrada (ou simplesmente entradas), e a variável Y é denominada variável de saída (ou simplesmente saída). Uma dada função lógica e suas respectivas variáveis pode ser representada graficamente, como mostrado na [Figura 4.2](#).



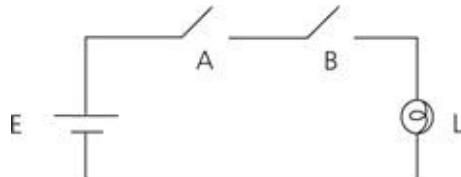
**FIGURA 4.2** Representação gráfica de uma função lógica.

# Funções Lógicas Básicas

Uma função básica é aquela que, devido a sua forma primária, serve como elemento construtivo para funções mais complexas. Na matemática, as operações aritméticas (soma, subtração, multiplicação e divisão) podem ser consideradas básicas, e são amplamente utilizadas para definir funções de maior complexidade. A seguir, são definidas as funções lógicas básicas na sua forma generalizada.

## Função AND

Imagine o circuito da [Figura 4.3](#), em que A e B são as variáveis de entrada e Y é a variável de saída correspondente ao estado luminoso da lâmpada L:



**FIGURA 4.3** Circuito ilustrativo da função AND.

Neste circuito, temos as seguintes convenções:

- Chave aberta = 0
- Chave fechada = 1
- Lâmpada apagada = 0
- Lâmpada acesa = 1

Observa-se que a lâmpada L só acende se as chaves A e B (daí o nome da função *and*) estiverem fechadas ao mesmo tempo, tal como mostrado na tabela a seguir, também conhecida como tabela-verdade.

Chave A	Chave B	Lâmpada L
Aberta	Aberta	Apagada
Aberta	Fechada	Apagada
Fechada	Aberta	Apagada
Fechada	Fechada	Acesa

A tabela-verdade é um mapeamento de todas as possíveis combinações das variáveis de entrada com seus respectivos resultados da variável de saída. A

tabela-verdade da função AND é dada na [Tabela 4.2](#).

---

### Tabela 4.2

#### Tabela-verdade da função AND

---

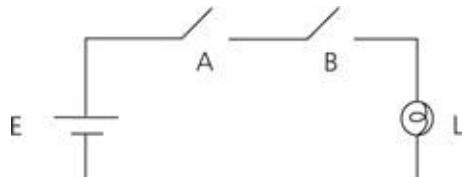
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

A equação lógica que representa a função AND tem a seguinte forma:

$$Y = (A \text{ AND } B) \text{ ou } Y = A \cdot B \text{ ou } y = AB$$

### Função OR

Imagine o circuito da [Figura 4.4](#) (valem as mesmas convenções usadas na função AND):



**FIGURA 4.4** Circuito ilustrativo da função OR.

Observa-se que neste caso a lâmpada L acende se *ao menos uma* das chaves, A e B (daí por que o nome da função vem de *or*, que significa “ou” em inglês) estiver fechada, tal como mostrado na tabela a seguir.

Chave A	Chave B	Lâmpada L
Aberta	Aberta	Apagada
Aberta	Fechada	Acesa
Fechada	Aberta	Acesa
Fechada	Fechada	Acesa

A [Tabela 4.3](#) apresenta a tabela-verdade da função.

---

### Tabela 4.3

#### Tabela-verdade da função OR

---

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

A equação lógica que representa a função OR tem a seguinte forma:

$$Y = (A \text{ OR } B) \text{ ou } Y = A + B$$

### Função NOT

A função NOT, também referida como negação (do inglês *not*, o nome da função), inversão ou complemento é uma função de apenas uma entrada que troca o valor da variável lógica presente nessa entrada. Lembre-se que uma variável lógica possui apenas dois valores. A tabela-verdade da função NOT é dada na [Tabela 4.4](#).

---

### Tabela 4.4

#### Tabela-verdade da função NOT

---

A	Y
0	0
1	0

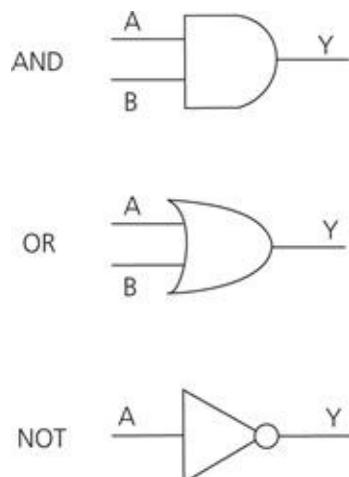
A equação lógica que representa a função NOT tem a seguinte forma:

$$Y = (\text{NOT } A) \text{ ou } Y = \bar{A}$$

A função NOT pode parecer inicialmente sem sentido, mas ela será muito utilizada em outros capítulos ao longo deste livro.

## Representação Simbólica

Uma função lógica básica é um conceito abstrato, porém iremos verificar ao longo deste capítulo que funções mais elaboradas serão compostas pela combinação de funções básicas. A forma como esta combinação se dá é melhor descrita por meio de um diagrama, ilustrando como as variáveis são conectadas às funções. Logo, torna-se interessante dispor de símbolos que representem estas funções. Estes símbolos também estão associados aos circuitos eletrônicos que executam estas funções no nível do hardware, denominados portas lógicas. A [Figura 4.5](#) mostra os símbolos adotados para as portas AND, OR e NOT.



**FIGURA 4.5** Símbolos lógicos AND, OR e NOT.

## Funções lógicas de N entradas

Pode-se generalizar a descrição de uma função AND ou OR para N entradas. O número de combinações possíveis das N entradas é igual a  $2^N$  ( $N$  é o número de variáveis e entradas), que é também o número de linhas da tabela-verdade. Veja na [Tabela 4.5](#) exemplos de tabelas-verdade de funções AND e OR de três entradas.

---

### Tabela 4.5

#### Exemplos de tabelas-verdade de funções AND e OR de três variáveis de entrada

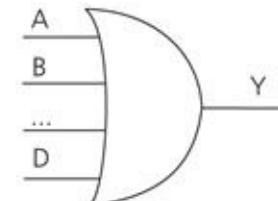
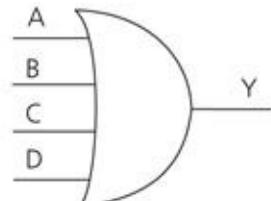
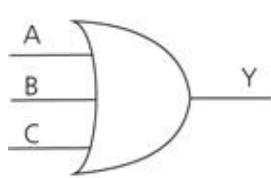
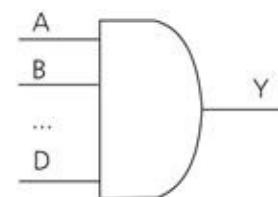
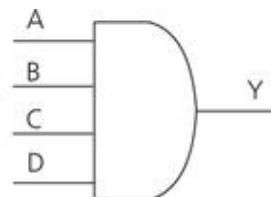
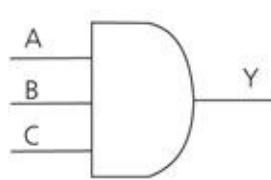
---

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

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

Na função AND, a saída Y terá valor 1 se (e somente se) todas as N entradas forem simultaneamente iguais a 1, e nas demais combinações terá valor 0. Na função OR, Y terá valor 1 em qualquer combinação das entradas em que pelo menos uma delas tenha valor 1, e apenas quando todas as N entradas forem 0 é que Y terá valor 0.

Observando as tabelas-verdade das funções de duas e três variáveis, procure imaginar como generalizar esses resultados para qualquer valor de N. Note que a função AND se relaciona à ideia de simultaneidade (... todas as entradas iguais a 1...), enquanto a função OR se relaciona à ideia de alternativa (... qualquer das entradas em valor 1...). A [Figura 4.6](#) ilustra os símbolos para portas lógicas de três, quatro e N entradas.



**FIGURA 4.6** Símbolos lógicos para funções de três, quatro e N entradas.

Observe ainda que as funções lógicas AND e OR podem ser completamente definidas por qualquer um desses elementos considerados individualmente: tabela-verdade, equação lógica ou símbolo. Estas três formas de representação são equivalentes, e esse é um resultado genérico extensível para qualquer função lógica com qualquer número de entradas.

## Funções Lógicas Derivadas

São funções lógicas que podem ser definidas por alguma associação das funções lógicas básicas.

### Função NAND

A função NAND é a função AND combinada com a função NOT. A tabela-verdade da função NAND corresponde à [Tabela 4.6](#).

---

#### Tabela 4.6

#### Tabela-verdade da função NAND

---

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

A equação lógica que representa a função NAND tem a seguinte forma:

$$Y = (A \text{ NAND } B) \text{ ou } Y = \overline{A \cdot B} \text{ ou } y = \overline{AB}$$

### Função NOR

A função NOR é a função OR combinada com a função NOT. A [Tabela 4.7](#) apresenta a tabela-verdade da função NOR.

---

#### Tabela 4.7

#### Tabela-verdade da função NOR

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

A equação lógica que representa a função NOR tem a seguinte forma:

$$Y = (A \text{ NOR } B) \text{ ou } Y = \overline{A + B}$$

## Função XOR

A função XOR é a conhecida como função OU-exclusivo (eXclusive-OR). A tabela-verdade da função XOR é dada na [Tabela 4.8](#). Note que Y vale 1 somente se apenas uma das entradas (exclusivamente) for 1, ou seja, quando as entradas forem diferentes entre si. É uma sutil diferença em relação à função OR.

---

### Tabela 4.8

#### Tabela-verdade da função XOR

---

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

A equação lógica que representa a função XOR tem a seguinte forma:

$$Y = (A \text{ XOR } B) \text{ ou } Y = A \oplus B$$

## Função XNOR

A função XNOR é uma associação da função XOR combinada com a função NOT. A [Tabela 4.9](#) mostra a tabela-verdade da função XNOR. Note que Y vale 1 somente se as entradas forem iguais entre si. Esta função também é conhecida como função equivalência.

---

**Tabela 4.9****Tabela-verdade da função XNOR**

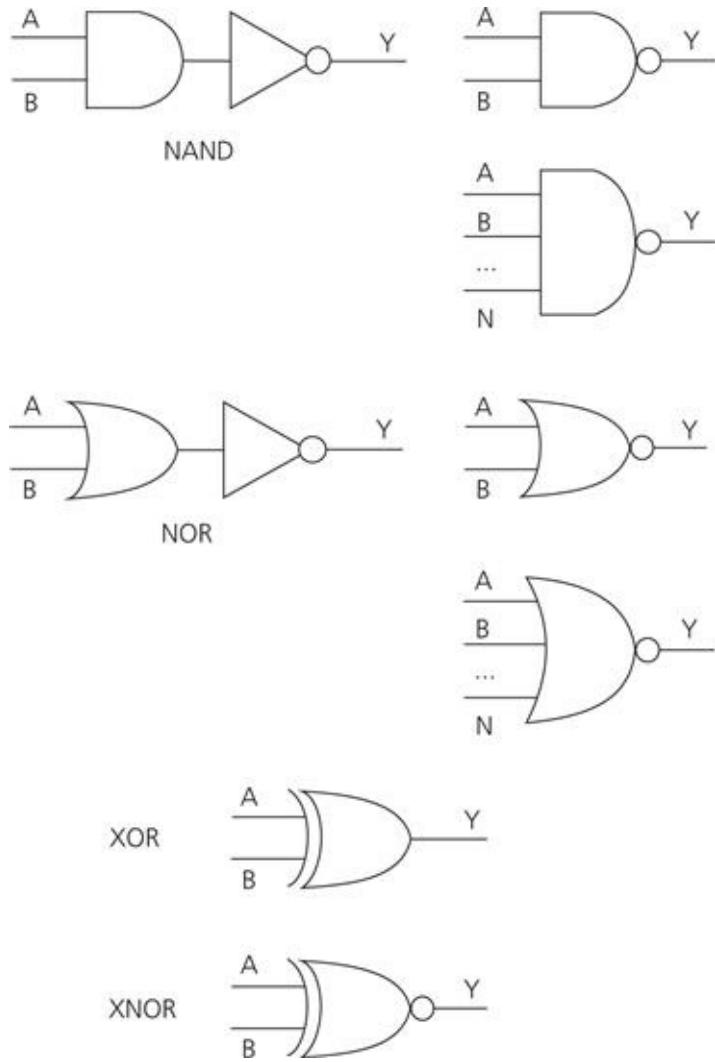
---

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

A equação lógica que representa a função XNOR tem a seguinte forma:

$$Y = (A \text{ XNOR } B) \text{ ou } Y = A \oplus B$$

Veja na [Figura 4.7](#) os diagramas lógicos equivalentes e os respectivos símbolos das funções NAND, NOR, XOR e XNOR.



**FIGURA 4.7** Símbolos lógicos NAND, NOR, XOR e XNOR.



## O que vem depois

Funções lógicas básicas são os blocos construtivos de qualquer função mais complexa. As funções derivadas são construídas com funções básicas. Entretanto, elas recebem uma distinção especial, pois aparecem com muita frequência em diversas aplicações práticas, como veremos nos capítulos a seguir. Uma vez entendidas as variáveis e funções lógicas vistas neste capítulo, estaremos aptos a avançar em nossos estudos e aprender sobre as leis, teoremas e propriedades da álgebra booleana, com as quais iremos manipular as funções

lógicas. Vamos lá!

---

## CAPÉTULO 5

# A álgebra booleana

---

## Objetivos do capítulo

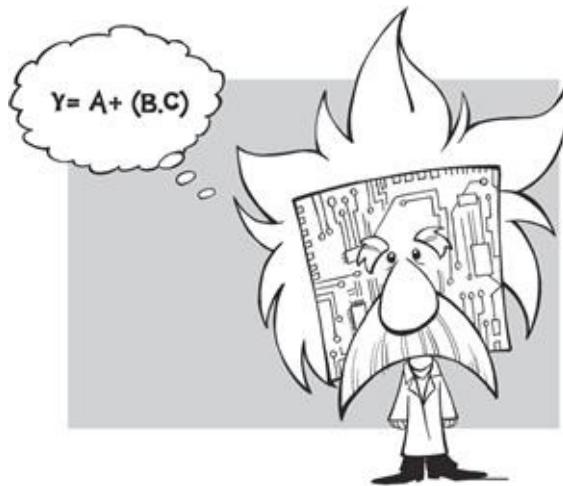
Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Identificar as propriedades das variáveis e funções lógicas básicas.
- Identificar as leis da álgebra booleana e os teoremas de DeMorgan.
- Reconhecer o princípio da dualidade.
- Comprovar a igualdade de expressões lógicas.
- Representar funções lógicas por meio de equações escritas a partir de tabelas-verdade e pelos todos da soma de produtos e do produto de somas.
- Obter um diagrama de circuito lógico referente a uma equação lógica e desenhar o respectivo diagrama de formas de onda.



## Apresentação

Na matemática convencional, entendese álgebra como um conjunto de regras, propriedades e teoremas baseados nas propriedades das operações aritméticas básicas (soma, subtração, multiplicação e divisão).



Ela é empregada para se manipular as variáveis de uma equação a fim de melhor representar a relação entre as variáveis e obter a representação mais simples da função. No domínio das variáveis lógicas, existe uma álgebra particular que opera sobre as variáveis e funções lógicas conhecida como álgebra booleana.

## Fundamentos

### ***Propriedades das variáveis lógicas***

A álgebra booleana foi desenvolvida no século XIX pelo matemático inglês George Boole. Ela se baseia nas propriedades de uma variável lógica e nas propriedades das funções lógicas básicas, e também em leis e teoremas.

Como definido no [Capítulo 4](#), uma variável lógica pode assumir apenas dois valores distintos. Disso se pode definir o postulado básico da complementação (ou negação):

$$\begin{aligned} A = 0 &\rightarrow \bar{A} = 1 \\ A = 1 &\rightarrow \bar{A} = 0 \end{aligned}$$

A negação de uma variável 0 só pode resultar em 1 e vice-versa. Em consequência, uma variável duplamente negada resulta nela mesma:

$$\begin{array}{l} A = 0 \rightarrow \overline{\overline{A}} = 0 \\ A = 1 \rightarrow \overline{\overline{A}} = 1 \\ \text{Ou seja} \quad \overline{\overline{A}} = A \end{array}$$

## Propriedades Das Funções Lógicas Básicas

Uma observação atenta da função OR discutida no [Capítulo 4](#) nos mostra algumas propriedades importantes. Veja como referência a tabela-verdade da função OR reproduzida a seguir:

**Tabela 5.1**

**Tabela-verdade da função OR**

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

Observa-se que se a variável B for fixada em 1 (ou seja, em vez de ser uma variável livre, ela é considerada uma constante de valor imutável), as possibilidades de combinação se reduzem a duas, restando apenas a 2<sup>a</sup> e 4<sup>a</sup> linhas da tabela-verdade, para as quais Y vale sempre 1. Disso se conclui que:

$$Y = A + 1 = 1 \rightarrow A + 1 = 1$$

Refazendo o mesmo raciocínio agora para B fixo em 0 (restrinindo a tabela-verdade a 1<sup>a</sup> e à 3<sup>a</sup> linhas) e em seguida para B igual a A (restando a 1<sup>a</sup> e 4<sup>a</sup> linhas), obtém-se:

$$\begin{array}{l} Y = A + 0 = A \rightarrow A + 0 = A \\ Y = A + A = A \rightarrow A + A = A \end{array}$$

E se B for igual a A negado? Observando-se as 2<sup>a</sup> e 3<sup>a</sup> linhas da tabela-verdade:

$$Y = A + \bar{A} = 1 \rightarrow A + \bar{A} = 1$$

Esse processo de dedução pode ser realizado também com a função AND.  
O resultado é a lista de propriedades (ou regras da álgebra booleana) mostrada a seguir:

1.  $A + 1 = 1$
2.  $A + 0 = A$
3.  $A + A = A$
4.  $A + \bar{A} = 1$  (Sempre uma das variáveis vale 1)
5.  $A \cdot 0 = 0$
6.  $A \cdot 1 = A$
7.  $A \cdot A = A$
8.  $A \cdot \bar{A} = 0$  (Sempre uma das variáveis vale 0)
9.  $\bar{\bar{A}} = A$  (A-barra-barra é igual a A)

A propriedade 9 também é conhecida como dupla inversão.

## Leis Da Álgebra Booleana

Comutar, associar ou distribuir variáveis na matemática convencional são operações bem conhecidas, justificadas pela validade das operações aritméticas. Da mesma forma, pode-se definir essas mesmas leis para as operações com variáveis lógicas.

1) Lei comutativa:

$A \cdot B = B \cdot A$	AND
$A + B = B + A$	OR
$A \oplus B = B \oplus A$	XOR

2) Lei associativa:

$$(A \cdot B) \cdot C = A \cdot (B \cdot C) \quad \text{AND}$$

$$(A + B) + C = A + (B + C) \quad \text{OR}$$

$$(A \oplus B) \oplus C = A \oplus (B \oplus C) \quad \text{XOR}$$

3) Lei distributiva:

$$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$$

$$A + (B \cdot C) = (A + B) \cdot (A + C)$$

Salienta-se neste caso a fatoração como a operação inversa da distribuição:

$$(A \cdot B) + (A \cdot C) = A \cdot (B + C)$$

$$(A + B) \cdot (A + C) = A + (B \cdot C)$$

É interessante notar que essas leis podem ser comprovadas com base nas operações lógicas envolvidas. Vamos então provar, por exemplo, que a segunda expressão da lei distributiva é verdadeira, pois à primeira vista ela parece estranha com base na nossa experiência anterior com a matemática convencional.

Partimos da ideia de que uma igualdade é verdadeira se o cálculo feito pela expressão de cada lado produzir o mesmo resultado – como na expressão  $(3 + 5) - 4 = 2 \times 2$ .

No caso de funções lógicas, utilizamos a tabela-verdade para calcular ambos os lados da igualdade com base em todas as possíveis combinações das entradas. Colunas de resultados iguais ponto a ponto provam que a igualdade é válida.

A	B	C	B · C	Y1	A + B	A + C	Y2
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

Dividimos a igualdade a ser comprovada em duas expressões:

$$A + (B \cdot C) = (A + B) \cdot (A + C) \rightarrow Y_1 = Y_2$$

$$Y_1 = A + (B \cdot C)$$

$$Y_2 = (A + B) \cdot (A + C)$$

Iniciamos desenvolvendo as oito combinações das variáveis A, B e C, e calculamos as operações dentro dos parênteses, prestando atenção apenas nas variáveis usadas em cada operação. No cálculo de (B · C), por exemplo, desconsidera-se A. Combinamos então as colunas dos termos entre parênteses (funções intermediárias) para compor os resultados finais de Y<sub>1</sub> e Y<sub>2</sub>. Ao observar as colunas Y<sub>1</sub> e Y<sub>2</sub> e considerar o objetivo do exemplo, poderemos ver que elas realmente são idênticas ponto a ponto. Logo, a igualdade de está provada. Provas similares podem ser feitas para as outras leis. Esse procedimento é conhecido como prova por tabela-verdade.

## Resultados Notáveis Da Álgebra Booleana

Esses são resultados de aplicação genérica, cuja validade pode ser provada, como visto no item anterior, e que resultam em simplificação da expressão lógica, ou seja, o lado direito das igualdades possui menor número de operações lógicas que o lado esquerdo.

10.  $A + (A \cdot B) = A$
11.  $A \cdot (A + B) = A$
12.  $(A \cdot B) + (A \cdot \bar{B}) = A$
13.  $(A + B) \cdot (A + \bar{B}) = A$
14.  $A + (\bar{A} \cdot B) = A + B$
15.  $A \cdot (\bar{A} + B) = A \cdot B$
16.  $(A + B) \cdot (A + C) = A + (B \cdot C)$
17.  $(A \cdot B) + (A \cdot C) = A \cdot (B + C)$
18.  $(A \cdot B) + (\bar{A} \cdot C) = (A + C) \cdot (\bar{A} + B)$
19.  $(A + B) \cdot (\bar{A} + C) = (A \cdot C) + (\bar{A} \cdot B)$

Esses resultados notáveis podem ser utilizados para simplificar expressões lógicas caso uma estrutura similar seja identificada.

## Teoremas De DeMorgan

As funções lógicas AND e OR são como água e vinho: muito diferentes em essência. Os teoremas de DeMorgan são dois resultados inusitados que definem condições para que uma função “AND modificada” seja equivalente a uma “função OR modificada” e vice-versa.

- **Teorema 1:** O complemento de um produto lógico (AND) de duas ou mais variáveis é igual à soma lógica (OR) dos complementos de cada variável individualmente.

$$(\overline{A \cdot B}) = \bar{A} + \bar{B}$$

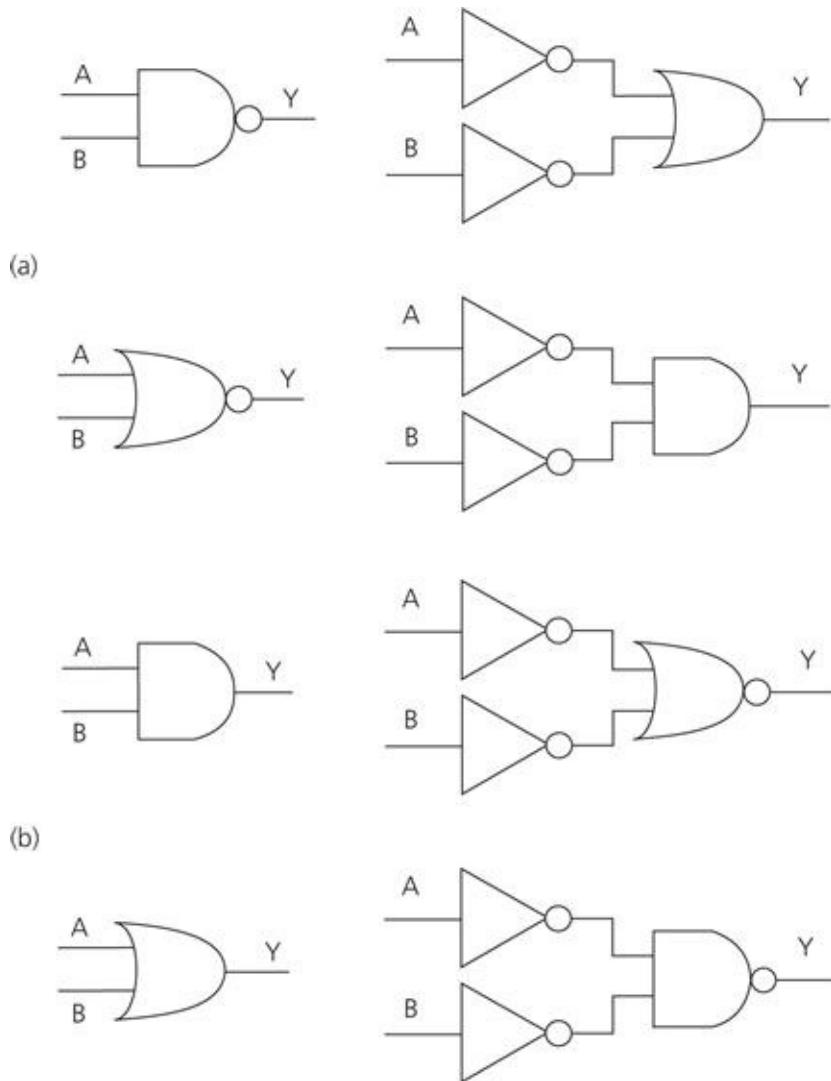
Ou seja, o NAND de A e B é igual ao OR de A-barra e B-barra.

- **Teorema 2:** O complemento de uma soma lógica (OR) de duas ou mais variáveis é igual ao produto lógico (AND) dos complementos de cada variável individualmente.

$$(\overline{A + B}) = \bar{A} \cdot \bar{B}$$

Ou seja, o NOR de A e B é igual ao AND de A-barra e B-barra.

Os teoremas de DeMorgan podem ser provados pelo método da tabela-verdade discutido na seção “Leis da Álgebra Booleana”. A [Figura 5.1](#) ilustra as equivalências possíveis entre expressões e diagramas lógicos que respeitam os teoremas de DeMorgan.



**FIGURA 5.1** Equivalências de DeMorgan: (a) obtidas diretamente dos teoremas e (b) equivalências derivadas.

Esses resultados também podem ser utilizados para simplificar expressões ou circuitos lógicos (veja mais detalhes no item “Simplificação Lógica Usando Mapas de Karnaugh” do [Capítulo 6](#)). Caso uma estrutura similar a alguma das existentes no lado direito da [Figura 5.1](#) seja identificada, basta substituí-la pela função correspondente à esquerda. Uma função AND, por exemplo, é a substituta ideal à configuração de uma função NOR com entradas negadas.

## O Princípio Da Dualidade

Analizando-se as leis, os teoremas e os resultados notáveis da álgebra \

Booleana, observamos uma semelhança estrutural entre pares de equações. Considere as equações (10) e (11) da seção “Resultados Notáveis da Álgebra Booleana”, neste mesmo capítulo, por exemplo. Elas diferem apenas pelos sinais “+” (OR) e “.”. (AND), que aparecem trocados em cada uma delas. Isso ocorre também nas equações das leis comutativa, associativa e distributiva, assim como nas duas equações dos teoremas de DeMorgan. Veja agora mais exemplos, nos quais, além das operações AND e OR, trocamos os 0s ou 1s da equação à esquerda para resultar na equação à direita:

$$A \cdot 0 = 0 \rightarrow A + 1 = 1$$

$$A \cdot 1 = A \rightarrow A + 0 = A$$

A essa semelhança estrutural denomina-se dualidade. O princípio da dualidade estabelece que se trocarmos as operações AND e OR entre si (dada uma equação booleana qualquer) e fizermos o mesmo com os valores lógicos de 0 e 1, obteremos uma equação igualmente válida.

## Como Surgem As Equações Lógicas?

Veremos no [Capítulo 7](#) como projetar circuitos lógicos a partir de uma especificação, ou seja, a partir da funcionalidade que desejamos do mesmo. Esta especificação é capturada na forma de uma tabela-verdade, e a partir desta existem métodos que podem ser empregados para se escrever a equação lógica, simplificá-la e por fim obter o diagrama de circuito lógico. Por ora, vamos imaginar que a tabela-verdade já existe.

### ***Expressões booleanas de soma de produtos***

Considere a tabela-verdade de três variáveis (A, B e C) mostrada a seguir. Observe que só duas combinações das variáveis de entrada geram uma saída 1.

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

0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

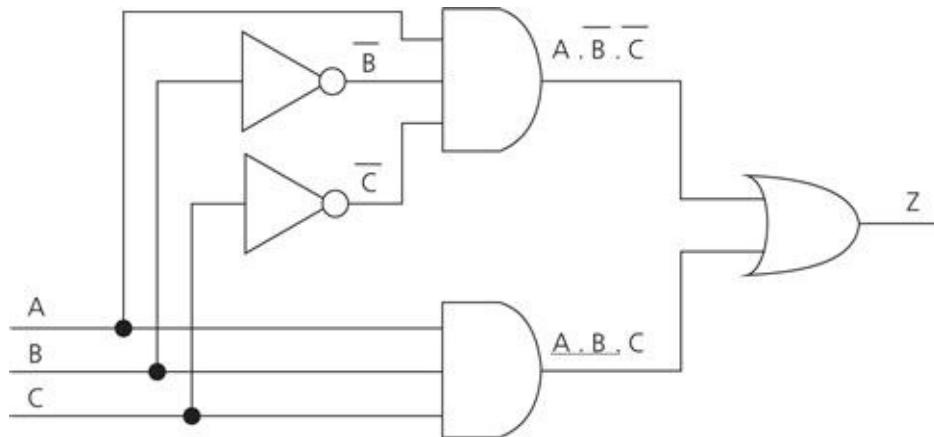
(A .  $\bar{B}$  .  $\bar{C}$ )

(A . B . C)

Na 5<sup>a</sup> linha da tabela anterior, podemos observar que a combinação de entradas (A = 1) E (B = 0) E (C = 0) gera uma saída (Z = 1). Isso é mostrado do lado da linha, com a expressão booleana (A. /B. /C). A outra combinação de variáveis que irá gerar uma saída 1 é mostrada na 8<sup>a</sup> linha, onde a combinação de entradas (A = 1) E (B = 1) E (C = 1) também gera uma saída (Z = 1). A expressão booleana equivalente é mostrada do lado da linha: (A. B. C). Essas duas combinações possíveis constituem alternativas, e portanto podem ser unidas por uma operação OR para formar a equação booleana completa da tabela-verdade.

$$Z = (A \cdot \bar{B} \cdot \bar{C}) + (A \cdot B \cdot C)$$

Esta forma de apresentar uma equação lógica é chamada de método da soma de produtos, conhecida também como forma de *termos mínimos*, ou *mintermos*. O diagrama lógico da [Figura 5.2](#) implementa a lógica contida na tabela-verdade inicial.



**FIGURA 5.2** Diagrama lógico obtido a partir de uma equação escrita pelo método de soma de produtos.

A função AND (produto lógico) que contenha todas as variáveis de entrada e que corresponda a uma posição da tabela-verdade na qual a saída vale 1 é denominada *produto fundamental*. Portanto, todo termo mínimo é também um produto fundamental.

### **Expressões booleanas de produto de somas**

Considere a tabela-verdade da função básica OR mostrada a seguir.

**Tabela 5.2**

**Tabela-verdade da função OR**

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

$$(A + B)$$

A expressão booleana dessa tabela-verdade pode ser escrita de duas formas. A primeira é a expressão de soma de produtos obtida a partir dos 1's de saída da tabela-verdade, como já visto no item anterior. Cada 1 na coluna de saída torna-se um termo a ser submetido a uma operação OR na expressão de termos

mínimos. A expressão de termos mínimos para a tabela-verdade é:

$$Z = (\bar{A} \cdot B) + (A \cdot \bar{B}) + (A \cdot B)$$

Também é possível obter a expressão lógica por meio dos termos *máximos*, ou *maxtermos*, forma também conhecida como método do *produto de somas*. Essa expressão é desenvolvida a partir dos 0 s de saída da tabela-verdade. Para cada 0 na coluna de saída é desenvolvido um termo OR, e então as variáveis de entrada são negadas e submetidas à operação OR. Assim, todos os termos OR obtidos são unidos por uma função AND para constituir a expressão final.

A expressão de termos máximos para a função básica OR é:

$$Z = (A + B)$$

A expressão booleana de termos máximos para a tabela-verdade desse exemplo revelase a mais simples. Tanto as expressões de termos mínimos quanto as de termos máximos descrevem a mesma função lógica expressa pela tabela-verdade, portanto são representações equivalentes.

Os métodos de *soma de produtos* ou *produto de somas* são ferramentas para se obter uma expressão booleana a partir de uma tabela-verdade. Estas expressões muitas vezes podem ser simplificadas. Note que estes dois métodos são duais.

## Diagramas Lógicos E Formas De Onda

Como vimos, cada função lógica tem um símbolo associado. Da mesma forma, uma expressão lógica composta por diversos termos como AND, OR e XOR (entre outros) pode também ser representada por um diagrama de símbolos lógicos, cuja conexão tem relação direta com o fluxo de variáveis pela expressão. Veja a seguir um exemplo. Considere a expressão:

$$Y = (A \cdot \bar{B}) + (\bar{A} \cdot B) + (A \cdot B)$$

Os termos entre parênteses são diferentes funções AND das variáveis A e B. Podemos calculá-los individualmente:

$$Y_1 = (A \cdot \bar{B})$$

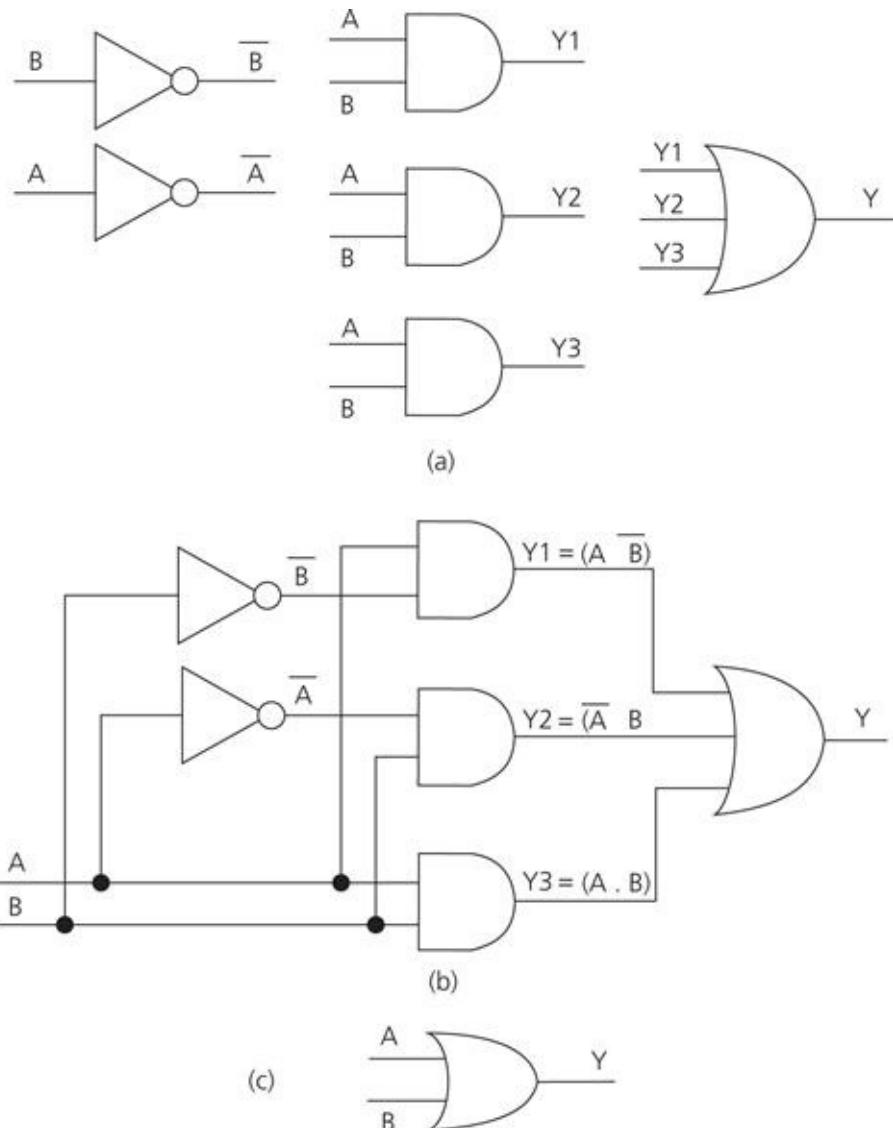
$$Y_2 = (\bar{A} \cdot B)$$

$$Y_3 = (A \cdot B)$$

Então, Y pode ser representada por:

$$Y = Y_1 + Y_2 + Y_3$$

Graficamente, as partes constituintes da função Y estão mostradas na [Figura 5.3.a](#); basta conectá-las, como mostrado na [Figura 5.3.b](#). Note que as variáveis de mesmo nome são ligadas pela mesma linha, mesmo que esta tenha de se ramificar. Pode-se ainda anotar sobre o diagrama as variáveis intermediárias (aqueles que são calculadas na saída de cada símbolo lógico), que ajudam no entendimento do circuito.

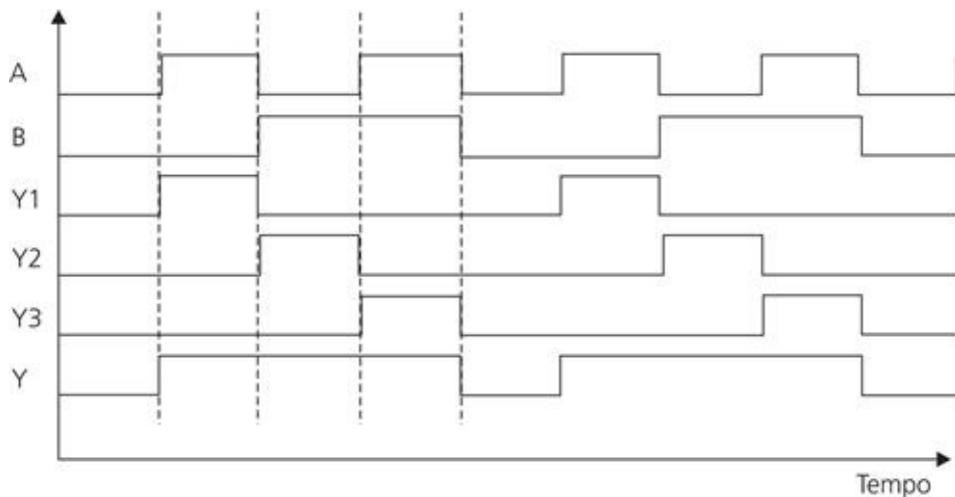


**FIGURA 5.3** Diagrama lógico de uma função  $Y$ : (a) partes constituintes, b) conexão das variáveis e (c) circuito simplificado.

Na prática, um diagrama lógico auxilia na montagem de circuito eletrônico que implementa a função lógica nele descrita. Os símbolos lógicos indicam quais portas lógicas devem ser utilizadas, e as ligações das variáveis são implementadas por fios condutores. Linhas que simplesmente se cruzam não significam conexão elétrica. Na [Figura 5.3.b](#), uma conexão elétrica é simbolizada por um ponto cheio, que pode ser entendido como uma solda entre dois fios.

Outro recurso muito empregado para facilitar o entendimento de um circuito lógico é o diagrama de formas de onda. Neste diagrama, mostrado na [Figura 5.4](#),

as modificações dos valores lógicos em função do tempo são simbolizados por linhas que alternam sua altura: “embaixo” para o 0 lógico e “em cima” para o 1 lógico. A mudança de 0 para 1 ou de 1 para 0 é sempre abrupta, porque uma variável lógica não pode apresentar valor diferente de 0 ou 1. Costuma-se listar verticalmente as variáveis de entrada e de saída, e marcar horizontalmente a evolução dos respectivos valores lógicos por linhas que transitam entre alto e baixo, cujo desenho final se denomina forma de onda.



**FIGURA 5.4** Diagrama de formas de onda correspondente ao circuito mostrado na Figura 5.3.

Para as variáveis de entrada, procura-se produzir todas as combinações presentes na tabela-verdade do circuito. Obviamente, as variáveis de saída devem indicar, neste caso, os respectivos valores das colunas de saída da tabela-verdade.

Na [Figura 5.4](#) temos as variáveis de entrada A e B, que necessitam de quatro combinações lógicas: as três variáveis intermediárias (Y1, Y2 e Y3) e a variável de saída Y. Note que as formas de onda de A e B se alternam de tal forma que, nos intervalos entre as linhas tracejadas, a combinação lógica é estável. O eixo vertical representa o valor lógico de cada variável, que não precisa de uma escala numérica ou ser anotado, pois basta perceber se a linha está “em cima” ou “embaixo”. O eixo horizontal representa o tempo, sem escala numérica por enquanto, porque ainda não precisamos atribuir essa dimensão ao diagrama.

Cabe ressaltar, finalmente, que como o diagrama de formas de onda de uma função contém todas as combinações lógicas possíveis das entradas, também

descreve totalmente a função à qual se refere. Portanto, temos agora quatro vistas equivalentes de uma função lógica: a tabela-verdade, a equação lógica, o símbolo ou diagrama lógico e o diagrama de formas de onda. Qualquer um deles, em princípio, é suficiente.

No final do item “Leis da Álgebra Booleana”, vimos que duas equações lógicas de formatos diferentes são equivalentes se apresentarem a mesma tabela-verdade. Estendendo este conceito, podemos verificar que essas mesmas duas equações também possuem as mesmas formas de onda nas variáveis de saída.

Analizar formas de onda é um procedimento muito comum quando se projeta um circuito digital. Atualmente, esse projeto é auxiliado por ferramentas computacionais chamadas simuladores lógicos, que a partir de uma descrição do diagrama de circuito lógico realizam a previsão das combinações lógicas das saídas e de variáveis intermediárias a partir da definição das formas de onda das entradas ou estímulos. Os resultados da simulação lógica são exibidos em diagramas de formas de onda.

Em outra situação, quando consideramos o circuito já montado com portas lógicas, é comum utilizar osciloscópios ou analisadores lógicos para observar as formas de onda de sinais elétricos reais medidos em qualquer ponto do circuito. Neste método, a medição também é exibida em diagramas de formas de onda.

Veremos mais à frente que certos circuitos, devido a sua maior complexidade e dependência direta da variável tempo, necessitam de diagramas de formas de onda para facilitar o entendimento de sua funcionalidade.



## O Que ven Deposis

Uma vez conhecidas as propriedades, as leis e os teoremas da álgebra booleana, e as funções lógicas descritas no [Capítulo 4](#), iremos construir funções lógicas mais complexas e aprender a escrever equações lógicas a partir de tabelas-verdade, que podem muitas vezes (e devem!) ser simplificadas.

---

## CAPITULO 6

# Funções lógicas e simplificação

---

### Objetivos do capítulo

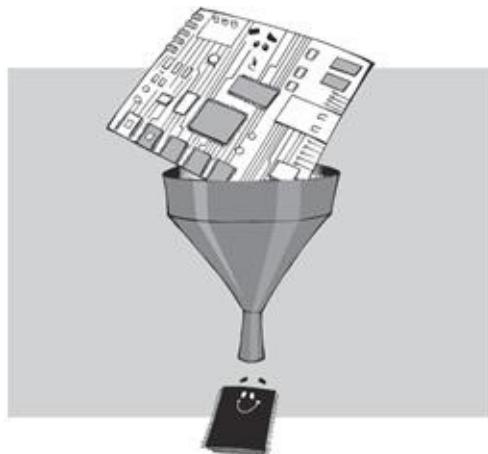
Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Simplificar expressões lógicas aplicando os princípios, as leis e os teoremas da álgebra booleana.
- Aplicar o método do mapa de Karnaugh para obter de forma sistemática a equação lógica mais simples a partir de uma tabela-verdade.



### Apresentação

Tabelas-verdade, equações lógicas, diagramas de circuito lógico e diagramas de formas de onda são formas equivalentes de representação de funções lógicas. Uma tabela-verdade representa uma única função, mas pode equivaler a diversas equações lógicas.



Obter a equação lógica mais simples torna-se uma necessidade para a construção de um circuito lógico mais otimizado e menos complexo. Estes serão

os temas discutidos neste capítulo.

## Fundamentos

### **Simplificação de expressões lógicas**

A expressão da função Y mostrada na figura do item anterior pode ser simplificada se as funções lógicas forem convenientemente manipuladas segundo as propriedades, as leis e os teoremas da álgebra booleana.

Veja a seguir:

$$\begin{aligned}Y &= (A \cdot \bar{B}) + (\bar{A} \cdot B) + (A \cdot B) \\&= (A \cdot B) + (A \cdot B) + (\bar{A} \cdot B) + (A \cdot B) \\&= [A \cdot (\bar{B} + B)] + [B \cdot (\bar{A} \cdot A)] \\&= (A \cdot 1) + (B \cdot 1) \\&= A + B\end{aligned}$$

Observe que no processo de simplificação a equação é reescrita linha após linha utilizando-se os seguintes recursos da álgebra booleana:

- a) Duplicando-se o último termo e comutando sua posição.
- b) Fatorando os pares de termos AND de acordo com a parte comum.
- c) Aplicando uma propriedade da função OR.
- d) Aplicando uma propriedade da função AND.

Observe que o processo de simplificação lógica, mostrado anteriormente, é muito dependente do grau de conhecimento e experiência do projetista na área da álgebra booleana. Por exemplo, a percepção da necessidade da etapa (a) deve ser imediata, pois sua realização significa aumentar inicialmente a quantidade de termos da equação para então se beneficiar da operação de fatoração (lei distributiva ao contrário).

A simplificação de expressões lógicas leva a uma implementação física bem mais econômica (no exemplo citado, foram economizadas três portas AND e duas portas NOT, sem contar a redução da porta OR para apenas duas entradas). No final do capítulo anterior, a [Figura 5.3.c](#) mostrou o diagrama lógico simplificado da função Y. Você deve ter notado que os circuitos lógicos das [Figuras 5.3.b](#) e [5.3.c](#) executam exatamente a mesma função lógica; um projetista obviamente escolheria o circuito mais simples e menos dispendioso.

Veremos a seguir que existem métodos sistemáticos de simplificação mais

simples e seguros do que a álgebra booleana, com os quais podemos obter a expressão lógica mais simples de uma função a partir de sua tabela-verdade.

## Mapas De Karnaugh

Um mapa de Karnaugh (ou mapa K) é uma forma gráfica de representação de uma tabela-verdade. É um método que auxilia na simplificação da equação lógica associada a esta tabela-verdade, a partir da deteção de emparelhamentos de 1s. A equação lógica simplificada apresenta-se na forma de produto de somas. Vejamos a seguir como construir os mapas K para funções de duas, três e quatro variáveis.

### **Mapas de Karnaugh para funções de duas variáveis**

Considere a função lógica definida pela tabela-verdade seguinte.

A	B	W	Posição
0	0	0	(0)
0	1	1	(1)
1	0	1	(2)
1	1	0	(3)

A construção do mapa K dessa tabela-verdade é mostrada a seguir; seu preenchimento deve ser segundo a ordem indicada pelo número de posição:

	$\bar{B}$	B
$\bar{A}$	(0)	(1)
A	(2)	(3)

	$\bar{B}$	B
$\bar{A}$	0	1
A	1	0

Cada 1 ou 0 da coluna W da tabela-verdade (variável de saída) possui “coordenadas” (A, B) no mapa K que dependem das combinações das variáveis de entrada A e B. Por exemplo, W = 1, na 2<sup>a</sup> linha da tabela-verdade, tem “coordenadas” A = 0 e B = 1; portanto, o 1 é colocado na intersecção da linha A-barra com a coluna B, ou seja, na posição (A-barra, B). O outro 1 existente na 3<sup>a</sup> linha desta tabela-verdade é alocado na posição (A, B-barra). Damos prioridade à alocação de todos os 1’s da variável de saída. As demais posições do mapa K são preenchidas com 0’s.

### **Mapas de Karnaugh para funções de três variáveis**

Considere a função lógica de três entradas definida pela tabela-verdade seguinte.

A	B	C	Z	Posição
0	0	0	0	(0)
0	0	1	0	(1)
0	1	0	1	(2)
0	1	1	0	(3)
1	0	0	0	(4)
1	0	1	0	(5)
1	1	0	1	(6)
1	1	1	1	(7)

A construção do mapa K de uma tabela-verdade de três variáveis é mostrada a seguir, e seu preenchimento deve obedecer a ordem indicada pelo número de posição:

	$\bar{C}$	C
$\bar{A} \bar{B}$	(0)	(1)
$\bar{A} B$	(2)	(3)
A B	(6)	(7)
A $\bar{B}$	(4)	(5)

	$\bar{C}$	C
$\bar{A} \bar{B}$	0	0
$\bar{A} B$	1	0
A B	1	1
A $\bar{B}$	0	0

Note que a ordem de preenchimento “salta” a 3<sup>a</sup> linha. A ordem como as variáveis A, B e C estão organizadas nas linhas e colunas do mapa K assegura que posições adjacentes, tanto na vertical como na horizontal, difiram por uma única variável (ou seja, nestas posições, apenas uma variável muda de valor lógico). Esta propriedade garante que o emparelhamento de 1's em qualquer posição do mapa K indique uma possibilidade de simplificação lógica, como veremos mais adiante.

### **Mapas de Karnaugh para funções de quatro variáveis**

Considere agora uma função lógica Y de quatro entradas, que possui uma tabela-verdade da qual se pode escrever a seguinte equação lógica por soma de produtos:

$$Y = (\bar{A} \cdot \bar{B} \cdot \bar{C} \cdot \bar{D}) + (A \cdot B \cdot \bar{C} \cdot \bar{D}) + \\ (A \cdot B \cdot \bar{C} \cdot D) + (A \cdot B \cdot C \cdot D)$$

Isso significa que cada termo AND da equação corresponde a um determinado 1 da tabela-verdade, cuja “coordenada” no mapa K é dada diretamente pela

combinação das variáveis. Por exemplo, o 1º termo da equação posiciona um 1 na intersecção da linha ( $A$ -barra,  $B$ ) e da coluna ( $C$ -barra,  $D$ -barra). O mapa K desta função é:

	$\bar{C} \bar{D}$	$\bar{C} D$	$C \bar{D}$	$C D$
$\bar{A} \bar{B}$	0	0	0	0
$\bar{A} B$	1	0	0	0
$A \bar{B}$	1	1	1	0
$A B$	0	0	0	0

Para tornar esse exemplo completo, a tabela-verdade relacionada à equação lógica fornecida seria:

---

**Tabela 6.1**  
**Tabela-verdade da equação lógica**

---

A	B	C	D	Y	Posição
0	0	0	0	0	(0)
0	0	0	1	0	(1)
0	0	1	0	0	(2)
0	0	1	1	0	(3)
0	1	0	0	1	(4)
0	1	0	1	0	(5)
0	1	1	0	0	(6)
0	1	1	1	0	(7)
1	0	0	0	0	(8)
1	0	0	1	0	(9)
1	0	1	0	0	(10)
1	0	1	1	0	(11)
1	1	0	0	1	(12)
1	1	0	1	1	(13)
1	1	1	0	0	(14)
1	1	1	1	1	(15)

A ordem de preenchimento de um mapa K de quatro variáveis é mostrada a seguir.

	$\bar{C} \bar{D}$	$\bar{C} D$	$C \bar{D}$	$C D$
$\bar{A} \bar{B}$	(0)	(1)	(3)	(2)
$\bar{A} B$	(4)	(5)	(7)	(6)
$A \bar{B}$	(13)	(14)	(16)	(15)
$A B$	(8)	(9)	(12)	(11)

Note que, quando lidamos com quatro variáveis, a ordem de preenchimento

“salta” não apenas a 3<sup>a</sup> linha, mas também a 3<sup>a</sup> coluna de cada linha. Novamente, a ordem como as variáveis A, B, C e D estão organizadas nas linhas e colunas do mapa K assegura que posições adjacentes, tanto na vertical como na horizontal, difram por uma única variável (ou seja, nestas posições, apenas uma variável muda de valor lógico). Esta propriedade garante que o emparelhamento de 1's em qualquer posição do mapa K indique uma possibilidade de simplificação lógica, como veremos mais adiante.

## Simplificação Lógica Usando Mapas De Karnaugh

A existência de 1's adjacentes no mapa K permite definir agrupamentos ou emparelhamentos. Vamos procurar reconhecer agrupamentos regulares de 1's que contenham dois, quatro, oito (ou mais) desses 1's, dispostos apenas nas direções horizontal e vertical, formando, por sua vez, pares, quadras, octetos etc.

Considere o mapa K, cuja variável de saída é W:

	$\bar{C}$	C
$\bar{A} \bar{B}$	0	0
$\bar{A} B$	1	0
A B	1	1
A $\bar{B}$	0	0

	$\bar{C}$	C
$\bar{A} \bar{B}$	0	0
$\bar{A} B$	1	0
A B	1	1
A $\bar{B}$	0	0

Observamos dois possíveis pares de 1's, mostrados graficamente pelas áreas sombreadas e devidamente circundados. No par definido verticalmente na tabela

da esquerda, observamos que as variáveis B e C-barra são comuns a ambos os termos AND, e que portanto podem ser fatoradas. Na sequência, a variável A pode ser eliminada, tal como demonstrado a seguir:

$$\begin{aligned}
 \dots &= (A \cdot B \cdot C) + (A \cdot B \cdot \bar{C}) = \\
 &= (\bar{B} \cdot \bar{C}) \cdot (\bar{A} + A) = \\
 &= (\bar{B} \cdot \bar{C}) \cdot 1 = \\
 &= (\bar{B} \cdot \bar{C})
 \end{aligned}$$

Portanto, um par (vertical ou horizontal) elimina uma variável, exatamente aquela que dentro do par aparece na sua forma normal e na forma negada. As demais variáveis que ficam constantes são mantidas.

No exemplo anterior, podemos observar dois pares: um par vertical que elimina a variável A e um par horizontal que elimina a variável C (confira!).

A expressão simplificada é composta pelo OR de todos os pares simplificados, e portanto fica assim:

$$W = (\bar{B} \cdot \bar{C}) + (A \cdot B)$$

Os agrupamentos de 1's podem ocorrer em quadras, que eliminam duas variáveis, em octetos, que eliminam três variáveis, e ainda em grupos de 16 1's, que eliminam quatro variáveis – além de poderem formar figuras regulares ainda maiores, com maior poder de eliminação.

Observe a quadra a seguir.

	$\bar{C}$	C
$\bar{A} \bar{B}$	0	0
$\bar{A} B$	1	1
A B	1	1
A $\bar{B}$	0	0

Quadra quadrada  
A e C são eliminadas

$$W = B$$

Escrevendo os termos AND referentes a cada 1 do mapa K e realizando a fatoração e a eliminação:

$$\begin{aligned}
 W &= (\overline{A} \cdot B \cdot \overline{C}) + (A \cdot B \cdot \overline{C}) + (\overline{A} \cdot B \cdot C) + (A \cdot B \cdot C) = \\
 &= (B \cdot \overline{C}) \cdot (\overline{A} \cdot A) + (B \cdot C) \cdot (\overline{A} \cdot A) = \\
 &= (B \cdot \overline{C}) \cdot 1 + (B \cdot C) \cdot 1 = \\
 &= (B \cdot \overline{C}) + (B \cdot C) = \\
 &= B \cdot (\overline{C} + C) = \\
 &= B
 \end{aligned}$$

A seguir, observe mais alguns exemplos de mapas K e suas respectivas expressões simplificadas.

### **Exemplos De Mapas K:**

- Este mapa K possui um par de 1's, que elimina a variável B.

	$\overline{B}$	B
$\overline{A}$	0	0
A	1	1

$$S = A$$

- Aqui, pode-se facilmente perceber a ocorrência de dois pares.

	$\overline{C}$	C
$\overline{A} \overline{B}$	1	0
$\overline{A} B$	1	0
A B	0	0
A $\overline{B}$	1	1

$$W = (\overline{A} \cdot \overline{C}) + (A \cdot \overline{B})$$

3. Neste mapa K, note a ocorrência de uma quadra em linha (que elimina as variáveis A e B), além de uma quadra quadrada e um par.

	$\bar{C} \bar{D}$	$\bar{C} D$	$C \bar{D}$	$C D$
$\bar{A} \bar{B}$	0	0	0	1
$\bar{A} B$	1	1	0	1
$A \bar{B}$	1	1	0	1
$A B$	0	1	0	1

$$M = (A \cdot \bar{C} \cdot D) + (B \cdot \bar{C}) + (C \cdot \bar{D})$$

Note que para fins de simplificação lógica podemos sobrepor parcialmente os emparelhamentos, mas não é válido encontrar um emparelhamento menor totalmente contido em um emparelhamento maior, como um par contido dentro de uma quadra.

## Casos Especiais De Mapas De Karnaugh

Como foi dito no item anterior, a existência de 1's adjacentes no mapa K permite definir agrupamentos que resultam em simplificações da equação lógica final. A forma como os mapas de Karnaugh são construídos também garante que possamos considerar adjacentes a primeira e a última linhas, as colunas mais à esquerda e mais à direita e as posições localizadas nos quatro cantos do mapa. Veja a seguir alguns exemplos dessas situações.

### Exemplos De Mapas K (Cont.):

4. Um par.

	$\bar{C}$	C
$\bar{A} \bar{B}$	0	
$\bar{A} B$	0	0
A B	0	0
A $\bar{B}$	0	

$$Y = \bar{B} \cdot C$$

5. Uma quadra quadrada e um par.

	$\bar{C} \bar{D}$	$\bar{C} D$	C D	$C \bar{D}$
$\bar{A} \bar{B}$	0		1	0
$\bar{A} B$		0	0	
A B	0	0	0	0
A $\bar{B}$	0		1	0

$$N = (\bar{A} \cdot B \cdot \bar{D}) + (\bar{B} \cdot D)$$

6. Um octeto vertical.

	$\bar{C} \bar{D}$	$\bar{C} D$	C D	$C \bar{D}$
$\bar{A} \bar{B}$		0	0	
$\bar{A} B$	1	0	0	1
A B	1	0	0	1
A $\bar{B}$		0	0	

$$P = \overline{D}$$

7. Um octeto horizontal.

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

$$Q = \overline{B}$$

8. Uma quadra de cantos.

	$\overline{C} \overline{D}$	$\overline{C} D$	C D	$C \overline{D}$
$\overline{A} \overline{B}$	1	0	0	1
$\overline{A} B$	0	0	0	0
A B	0	0	0	0
A $\overline{B}$	1	0	0	1

$$W = \overline{B} \cdot \overline{D}$$

Todos esses casos podem também ocorrer juntamente com agrupamentos internos regulares e sobreposições parciais, como no exemplo a seguir.

9. Confira os emparelhamentos!

	$\bar{C} \bar{D}$	$\bar{C} D$	$C D$	$C \bar{D}$
$\bar{A} \bar{B}$	1	0	0	1
$\bar{A} B$	1	1	0	1
$A \bar{B}$	1	1	0	0
$A B$	0	1	0	1

$$M = (\bar{A} \cdot \bar{C} \cdot D) + (\bar{B} \cdot C \cdot \bar{D}) + (B \cdot \bar{C}) + (\bar{A} \cdot \bar{D})$$

O que ocorre se um determinado 1 do mapa K não emparelha com nenhum outro? Simples: esse 1 não participará de simplificações lógicas e seu termo mínimo não será reduzido (não ocorre redução do número de variáveis lógicas do termo). O exemplo seguinte ilustra essa situação.

#### 10. Ocorrência de um 1 isolado.

	$\bar{C} \bar{D}$	$\bar{C} D$	$C D$	$C \bar{D}$
$\bar{A} \bar{B}$	0	0	0	0
$\bar{A} B$	1	1	1	1
$A \bar{B}$	0	1	1	0
$A B$	0	0	0	1

$$J = (\bar{A} \cdot B) + (B \cdot D) + (A \cdot \bar{B} \cdot C \cdot \bar{D})$$

Em todos os exemplos discutidos, os emparelhamentos são feitos apenas nas direções horizontal e vertical. Não são válidos emparelhamentos na direção diagonal.

#### 11. Mapa K de uma função OU-exclusivo:

	$\bar{B}$	B
$\bar{A}$	0	1
A	1	0

$$J = (\bar{A} \cdot B) + (A \cdot \bar{B}) = A + B$$

Observe que não ocorre emparelhamento de 1's na horizontal ou na vertical. A existência de 1's na diagonal é característica da função XOR. O mapa K não resulta em simplificação lógica. Entretanto, o reconhecimento dos termos AND na expressão lógica anterior permite ao projetista a simplificação manual da equação.



## O que vem depois

Neste capítulo, aprendemos a simplificar expressões lógicas pela utilização das propriedades, das leis e dos teoremas da álgebra booleana. Entretanto, vimos também que o melhor método de simplificação se baseia no uso do mapa de Karnaugh. A seguir, iremos definir os circuitos lógicos combinacionais e estudar vários tipos desses circuitos, utilizados em diversas situações práticas.

---

## CAPÍTULO 7

---

# Circuitos lógicos combinacionais (primeira parte)

---

## Objetivos Do Capítulo

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Compreender e reconhecer circuitos lógicos combinacionais.
- Compreender o princípio de operação e saber projetar circuitos multiplexadores, demultiplexadores, decodificadores e codificadores.
- Entender o processo de projeto lógico de um circuito combinacional.

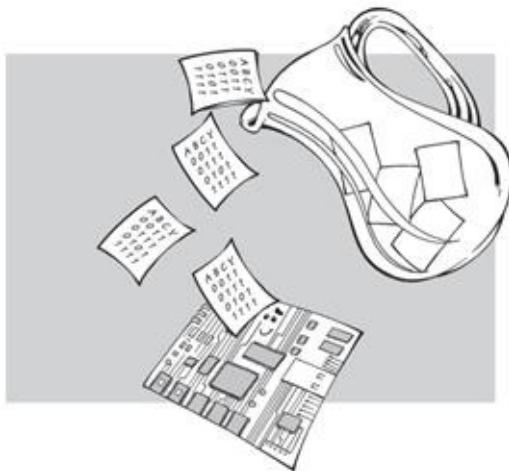


## Apresentação

Circuitos lógicos combinacionais são aqueles em que as variáveis de saída dependem apenas e exclusivamente das combinações lógicas das variáveis de entrada de acordo com uma função lógica específica. Eles são compostos por portas lógicas AND, OR, NOT, NAND, NOR, XOR ou XNOR interligadas. Por exemplo:

$$Y = A \cdot (B + C)$$

$$Y = A \cdot (B + C)$$



Vamos ver a seguir como projetar tais circuitos combinacionais para resolver alguns problemas práticos.

## Fundamentos

### **Circuitos baseados em uma especificação**

Vimos nos capítulos anteriores que qualquer função lógica pode ser representada de quatro formas distintas e totalmente equivalentes: tabela-verdade, equação lógica, circuito lógico e diagrama de formas de onda.

Mas como surge uma função lógica? Ela surge pela especificação de um problema cuja solução seja possível com a utilização de variáveis lógicas (que possuem apenas dois valores), às quais pode se aplicar a álgebra booleana.

Quando há um problema a ser resolvido, normalmente se utiliza um procedimento definido como projeto lógico combinacional, composto dos seguintes passos:

- Especificação do problema
- Definição das variáveis lógicas de entrada e saída
- Preenchimento da tabela-verdade
- Obtenção da equação lógica simplificada
- Desenho do diagrama lógico

Inicialmente, procuramos representar o problema a ser resolvido por meio de um diagrama de bloco (mostrado de forma genérica na [Figura 7.1](#)), no qual são indicadas as entradas, saídas, principais funções e informações complementares que são obtidas a partir da especificação.



**FIGURA 7.1** Diagrama de bloco genérico de um projeto lógico.

Como exemplo, vamos considerar a seguinte especificação:

Projete um circuito lógico para auxiliar na votação da diretoria de uma empresa com três diretores, de tal modo que cada diretor tenha acesso a um botão para a seleção da votação favorável (SIM). No caso de votação desfavorável, o fato de não apertar o botão seleciona a opção NÃO. Uma lâmpada deve acender quando houver maioria simples favorável ao SIM.

Nessa especificação, podemos identificar as seguintes variáveis lógicas:

a) Variáveis de entrada: diretores A, B e C

Estas variáveis são implementadas por meio de botões, de tal modo que:

Botão apertado = 1, quando um diretor votar SIM

Botão desapertado = 0, quando o mesmo votar NÃO

b) Variáveis de saída: lâmpada Y

Lâmpada acesa = 1, se houver maioria simples (caso dois ou mais diretores votem SIM)

Lâmpada apagada = 0, se não houver maioria simples (quando nenhum ou apenas um diretor votar SIM).

Observe que na definição das variáveis de entrada e saída é possível descrever a funcionalidade da função lógica (por exemplo, quando anotamos a afirmação “caso dois ou mais diretores votem SIM”).

O próximo passo é o preenchimento da tabela-verdade. Como temos três variáveis de entrada, essa tabela deve conter oito linhas para acomodar todas as combinações possíveis de A, B e C; veja a tabela a seguir como exemplo. A coluna Y representa a saída, e o seu preenchimento com 1's deve ser feito linha a linha considerando que a situação de maioria simples corresponde às combinações das entradas que contenham ao menos dois 1's, ou seja, nas posições 3, 5, 6 e 7 da tabela. Por exemplo, na linha 6 o valor de Y é 1, pois os diretores A e B apertam o botão (dois SIM definem a maioria em um conjunto de

três votos possíveis).

**Tabela 7.1**

**Tabela-verdade do exemplo**

A	B	C	Y	Posição
0	0	0	0	(0)
0	0	1	0	(1)
0	1	0	0	(2)
0	1	1	1	(3)
1	0	0	0	(4)
1	0	1	1	(5)
1	1	0	1	(6)
1	1	1	1	(7)

A tabela-verdade anterior corresponde à função lógica de detecção de maioria simples. Observando a distribuição dos bits na coluna Y, percebe-se que Y não corresponde a nenhuma função lógica básica ou derivada.

Portanto, para se obter a equação lógica associada a Y, temos de escrevê-la por meio da técnica de soma de produtos seguida de simplificação lógica manual baseada nas leis da álgebra booleana (procure fazer isso como exercício).

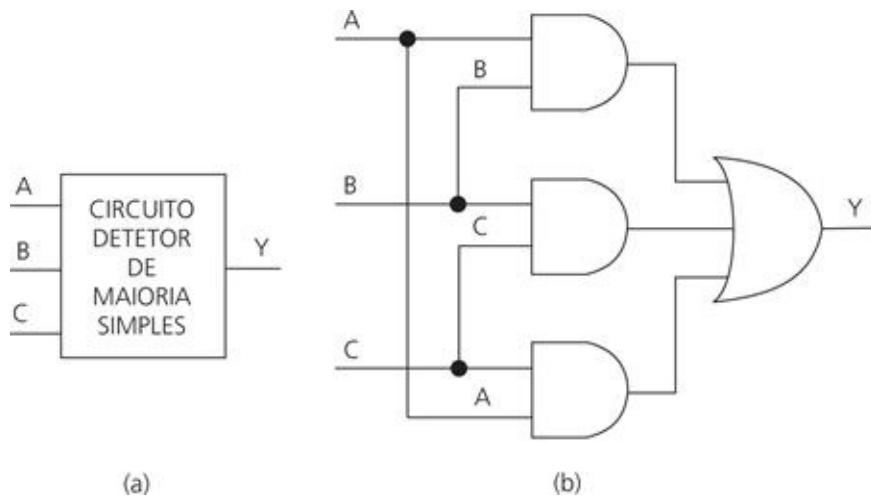
Podemos também utilizar diretamente um mapa de Karnaugh para obter Y na sua forma simplificada. O mapa K desta função é mostrado a seguir.

	$\bar{C}$	C
$\bar{A} \bar{B}$	0	0
$\bar{A} B$	0	1
$A \bar{B}$	1	1
$A B$	0	1

Identificam-se no mapa três pares, que resultam na equação simplificada seguinte:

$$Y = (A \cdot B) + (B \cdot C) + (A \cdot C)$$

Finalmente, a [Figura 7.2](#) mostra o circuito lógico correspondente à função maioria.



**FIGURA 7.2** Função maioria simples: (a) diagrama de bloco e (b) circuito lógico simplificado.

O exemplo que foi descrito aqui demonstra o procedimento a ser seguido para se projetar um circuito lógico combinacional de qualquer funcionalidade a partir de sua descrição em uma especificação. A seguir, outros tipos de projetos lógicos combinacionais serão descritos. Eles constituem classes de circuitos com funções padronizadas, tais como multiplexação, demultiplexação, codificação, decodificação etc., e são muito utilizados em processamento de dados.

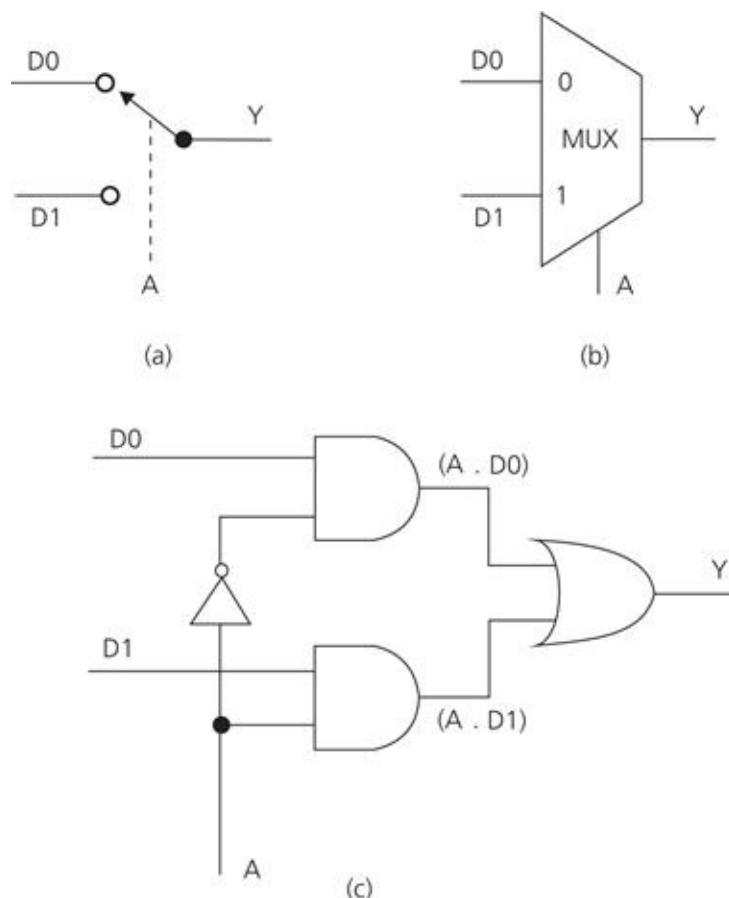
Nesse contexto, um dado é uma coleção de bits (ou variáveis lógicas de entrada) modificados ou analisados por uma função lógica, que podem conter algumas entradas auxiliares que configuram essa função.

## Multiplexadores

Por definição, um multiplexador é um circuito que seleciona uma dentre várias entradas de dados quando uma das possíveis combinações das entradas de

seleção é ativada. A tabela-verdade a seguir descreve um multiplexador 2 para 1. Quando a entrada de seleção A é 0, a entrada D0 é selecionada e a saída Y reflete os valores lógicos de D0, ou seja,  $Y = D0$ . Por outro lado, quando  $A = 1$ , D1 é selecionada fazendo  $Y = D1$ .

Note que “multiplexar” é o mesmo que “selecionar”, ou seja, o multiplexador (MUX) tem função equivalente ao circuito seletor com chave de duas posições mostrado na [Figura 7.3.a](#).



**FIGURA 7.3** Circuito multiplexador 2 para 1 (MUX 2:1): (a) circuito seletor equivalente, (b) símbolo lógico e (c) circuito lógico interno.

A tabela-verdade de um multiplexador 2 para 1 pode ser resumida assim:

A	Y	
0	D0	$A = 0$ seleciona D0, e $Y = D0$
1	D1	$A = 1$ seleciona D1, e $Y = D1$

O símbolo lógico de um MUX 2:1 (lê-se Multiplexador 2 para 1) é mostrado na [Figura 7.3.b](#).

O circuito interno de um MUX 2:1 pode ser projetado a partir de sua tabela-verdade completa. Aqui, temos três entradas – portanto, oito combinações.

---

**Tabela 7.2**

**Tabela-verdade de um MUX 2:1**

---

A	D0	D1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	2

Observe na tabela anterior que, quando  $A = 0$ , a sequência de 1's e 0's da coluna D0 aparece na saída Y. De forma similar, para  $A = 1$ , a parte inferior da coluna D1 aparece em Y, tal como mostrado pelos retângulos inseridos sobre a tabela. O mapa K é mostrado a seguir.

	$\overline{D1}$	D1
$\overline{A} \ D0$	0	0
$\overline{A} \ D0$	1	1
$A \ D0$	0	1
$A \ \overline{D0}$	0	1

Identificam-se aí dois pares, que resultam na equação seguinte:

$$Y = (\bar{A} \cdot D0) + (A \cdot D1)$$

Veja na [Figura 7.3.c](#) o circuito lógico interno de um MUX 2:1.

Podemos generalizar o resultado do projeto de um MUX 2:1 para um número maior de entradas. O símbolo lógico de um MUX 4:1 é mostrado na [Figura 7.4.a](#). Como agora temos quatro entradas de dados, D0, D1, D2 e D3, necessitamos de duas entradas de seleção A e B de forma a dispor de quatro combinações de seleção, uma para cada entrada. A entrada D0, por exemplo, é selecionada quando as entradas de seleção forem respectivamente AB = 00. A tabela-verdade a seguir resume a funcionalidade de um MUX 4:1.

---

**Tabela 7.3**

**Tabela-verdade do MUX 4:1**

---

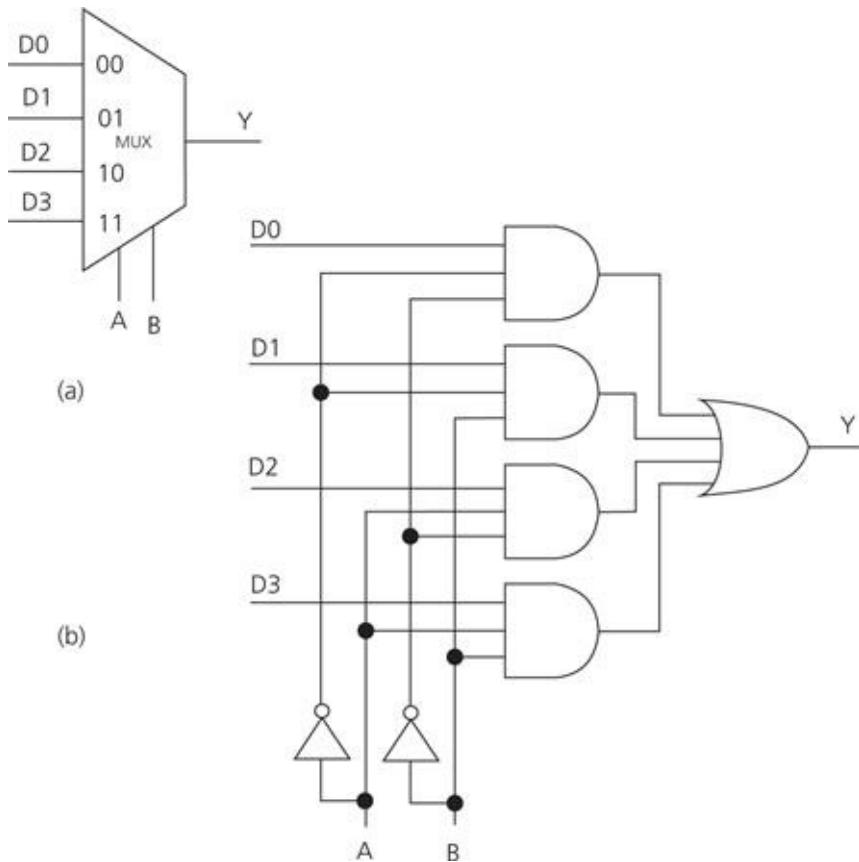
A	B	Y
0	0	<b>D0</b>
0	1	<b>D1</b>
1	0	<b>D2</b>
1	1	<b>D3</b>

AB = 00 seleciona D0, e Y = D0

AB = 01 seleciona D1, e Y = D1

AB = 10 seleciona D2, e Y = D2

AB = 11 seleciona D3, e Y = D3



**FIGURA 7.4** Circuito multiplexador 4 para 1 (MUX 4:1): (a) símbolo lógico e (b) circuito lógico interno.

O projeto lógico detalhado na forma apresentada no item “Circuitos baseados numa especificação” torna-se complexo, pois o MUX 4:1 possui seis entradas (a tabela-verdade teria 32 linhas e o mapa K de seis variáveis seria multidimensional). Por isso, podemos tentar perceber a lei de formação da equação e do circuito lógico do MUX 2:1 e procurar estendêla para o MUX 4:1. Como a saída Y do exemplo do MUX 2:1 é um OR de termos AND entre parênteses, cada termo AND deve ter uma das variáveis de entrada de dados unida AND com a respectiva combinação de seleção. A entrada D0, por exemplo, é selecionada quando a combinação é AB = 00, assim como D1 é selecionada pela combinação AB = 01, D2 por AB = 10 e D3 por AB = 11. A equação lógica se modifica para:

$$Y = (\overline{A} \cdot \overline{B} \cdot D_0) + (\overline{A} \cdot B \cdot D_1) + (A \cdot \overline{B} \cdot D_2) + (A \cdot B \cdot D_3)$$

Agora, procure relacionar essa equação com o circuito lógico da [Figura 7.4.b](#).

De forma similar, podemos definir multiplexadores 8:1, 16:1, 32:1 e assim por diante, desde que a quantidade de entradas E seja uma potência de 2, para as quais serão necessárias, respectivamente, 3, 4 e 5 (e assim por diante) entradas de seleção S. A relação entre E e S é definida pela fórmula:

$$E = 2^S$$

A quantidade de ANDs é igual à quantidade de entradas E, e cada AND deve ter  $E + 1$  entradas – e pelo menos uma delas tem de ser uma das entradas de dados. Para cada entrada de seleção S, devese gerar a variável S-barra, e distribuir ambas para as proximidades das entradas dos ANDs. Cada AND deve completar suas entradas com uma combinação das entradas de seleção obedecendo a tabela-verdade resumida. Por fim, a saída Y é um OR de todas as saídas dos ANDs intermediários.

Pode-se ainda definir um MUX com quantidade de entradas diferente de uma potência de 2. Para isso basta utilizar uma configuração baseada em potência de 2 com um valor imediatamente acima à quantidade de entradas desejada e remover o circuito das combinações não necessárias.

A [Figura 7.5](#) mostra o projeto de um MUX 3:1 cuja tabela-verdade e equação lógica são mostrados a seguir.

#### Tabela 7.4

#### Tabela-verdade do MUX 3:1

A	B	Y
0	0	D0
0	1	D1
1	0	D2
1	1	-

AB = 00 seleciona D0, e Y = D0

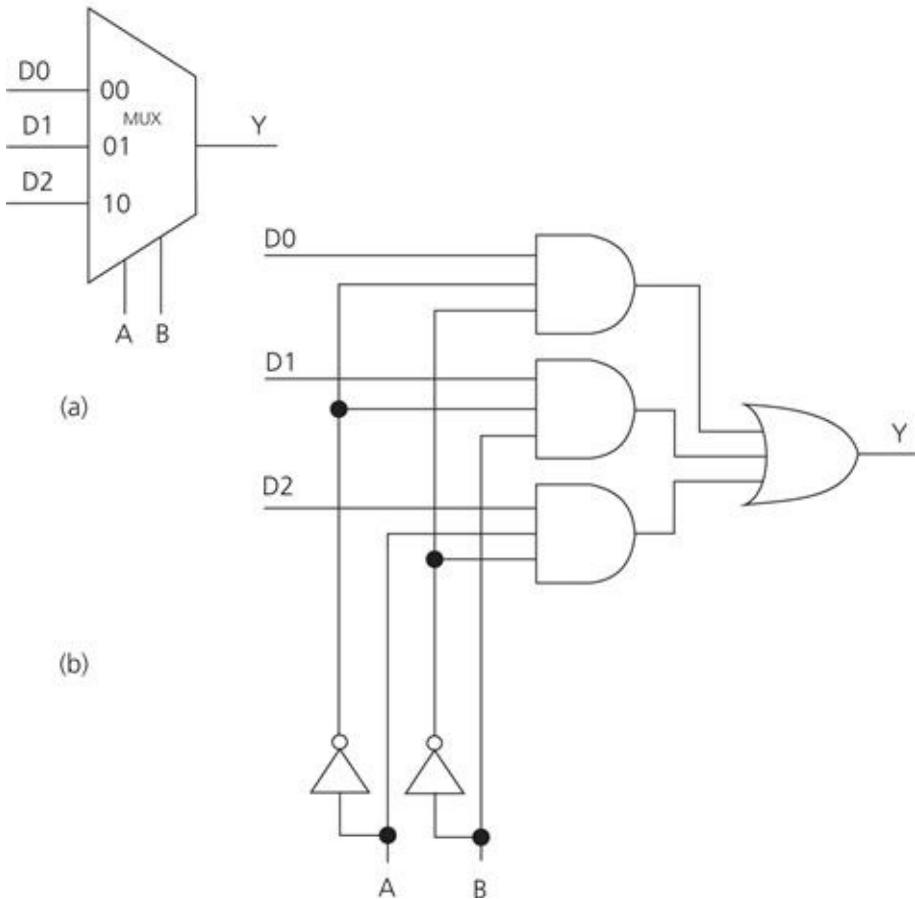
AB = 01 seleciona D1, e Y = D1

AB = 10 seleciona D2, e Y = D2

Não utilizada

$$Y = (\overline{A} \cdot \overline{B}) \cdot D0 + (\overline{A} \cdot B) \cdot D1 + (A \cdot \overline{B}) \cdot D2$$

$$Y = (\overline{A} \cdot \overline{B} \cdot D0) + (\overline{A} \cdot B \cdot D1) + (A \cdot \overline{B} \cdot D2)$$



**FIGURA 7.5** Circuito multiplexador 3 para 1 (MUX 3:1): (a) símbolo lógico e (b) circuito lógico interno.

Compare os circuitos das [Figuras 7.4](#) e [7.5](#) para perceber que a entrada não utilizada é removida, assim como o AND associado.

Como exemplo adicional, poderíamos considerar o projeto de um MUX 5:1, partindo de um MUX 8:1 e removendo daí três entradas e suas respectivas combinações. Mesmo assim, para cinco entradas de dados são necessárias três entradas de seleção. É impossível fazer com apenas duas. Comprove!

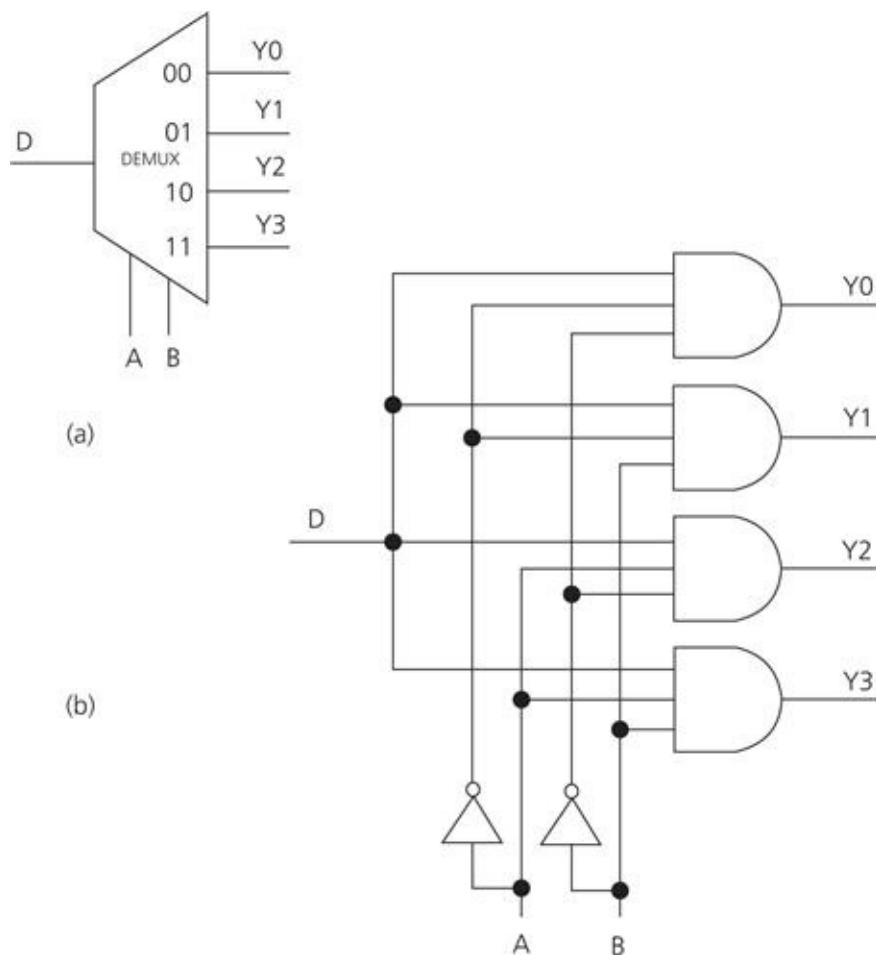
## Demultiplexadores

Um demultiplexador é um circuito que faz a operação inversa de um multiplexador, ou seja, seleciona uma dentre várias saídas quando uma das possíveis combinações das entradas de seleção é ativada.

A tabela-verdade seguinte descreve um demultiplexador 1 para 2. Quando a

entrada de seleção A é 0, a saída Y0 é selecionada para receber os valores lógicos da entrada D, ou seja,  $Y_0 = D$  e  $Y_1 = 0$ . Por outro lado, quando  $A = 1$ ,  $Y_1$  é selecionada para resultar  $Y_1 = D$ , fazendo agora  $Y_0 = 0$ . Note que a saída não selecionada fica sempre nula.

Note que “demultiplexar” é o mesmo que “distribuir”, ou seja, o demultiplexador (DEMUX) tem função equivalente ao circuito distribuidor com chave de duas posições mostrado na [Figura 7.6.a](#).



**FIGURA 7.6** Circuito demultiplexador 1 para 2 (DEMUX 1:2): (a) circuito distribuidor equivalente, (b) símbolo lógico e (c) circuito lógico interno.

A tabela-verdade de um DEMUX também pode ser resumida:

A	Y0	Y1
0	D	0
1	0	D

A = 0 distribui D para Y0 e faz Y1 = 0

A = 1 distribui D para Y1 e faz Y0 = 0

O símbolo lógico de um DEMUX 1:2 (lê-se Demultiplexador 1 para 2) é mostrado na [Figura 7.6.b](#). O circuito interno de um DEMUX 2:1 pode ser projetado a partir de sua tabela-verdade completa.

### Tabela 7.5

#### Tabela-verdade do DEMUX 1:2

A	D	Y0	Y1
0	0	0	0
0	1	1	0
1	0	0	0
1	1	0	1

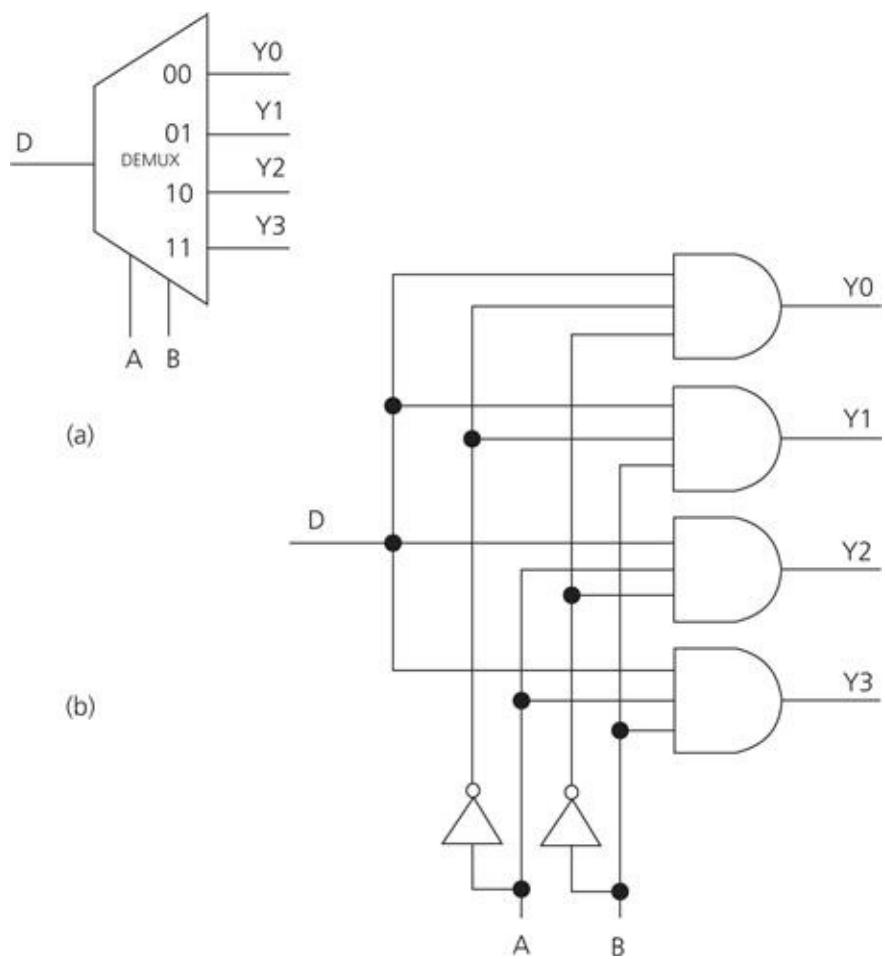
As duas saídas Y0 e Y1 são independentes, portanto podemos considerá-las como dois circuitos independentes. Inspecionando-se as colunas Y0 e Y1, conseguimos facilmente escrever as equações por soma de produtos, pois ambas possuem apenas um único 1.

$$Y0 = (\bar{A} \cdot D) \quad Y1 = (A \cdot D)$$

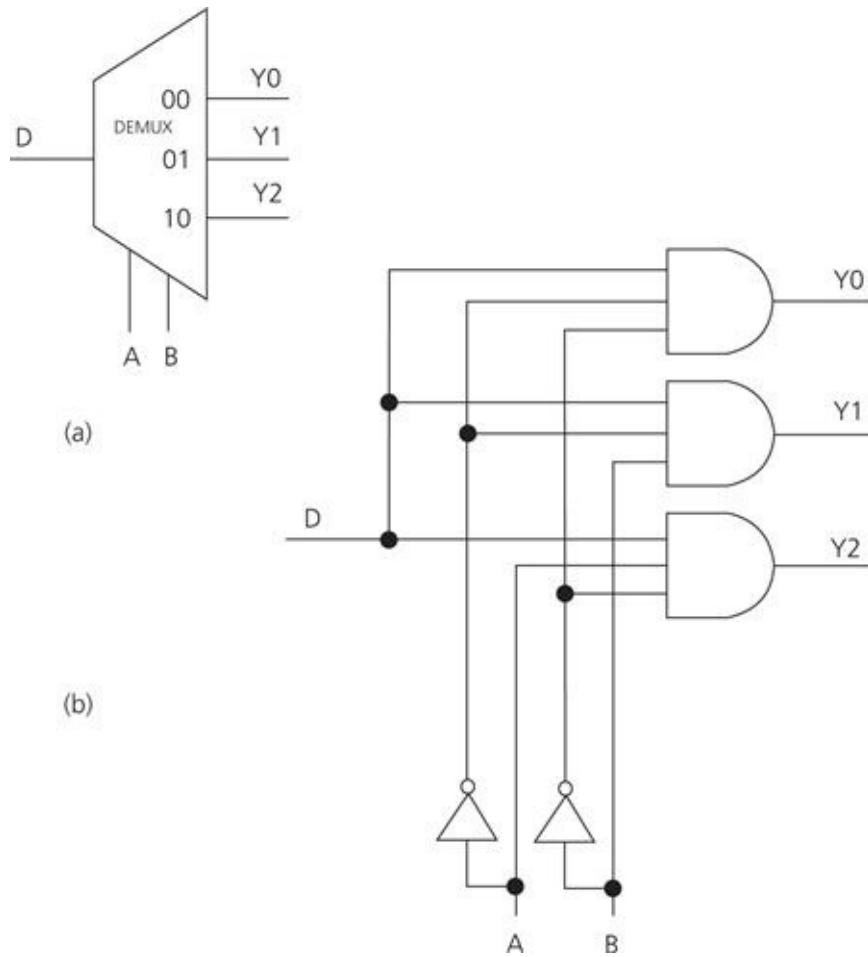
A [Figura 7.6.c](#) mostra o circuito interno de um DEMUX 1:2.

Assim como no caso dos multiplexadores, podemos generalizar o resultado do projeto de um DEMUX 1:2 para um número maior de entradas.

As [Figuras 7.7](#) e [7.8](#) mostram os símbolos e os circuitos lógicos internos para um DEMUX 1:4 e um DEMUX 1:3, respectivamente.



**FIGURA 7.7** Circuito demultiplexador 1 para 4 (DEMUX 1:4): (a) símbolo lógico e (b) circuito lógico interno.



**FIGURA 7.8** Circuito demultiplexador 1 para 3 (DEMUX 1:3): (a) símbolo lógico e (b) circuito lógico interno.

Note que o método discutido no item anterior para a determinação da lei de formação da equação lógica e do circuito lógico dos multiplexadores é similar no caso dos circuitos tipo DEMUX. Repare apenas que a quantidade de sinais de seleção S passa a ser determinada em função da quantidade de saídas O pela fórmula:

$$O = 2^S$$

De forma similar, para se definir um DEMUX com quantidade de saídas diferente de uma potência de 2, basta utilizar uma configuração baseada em potência de 2 de valor imediatamente acima à quantidade de saídas desejada e remover as não utilizadas.

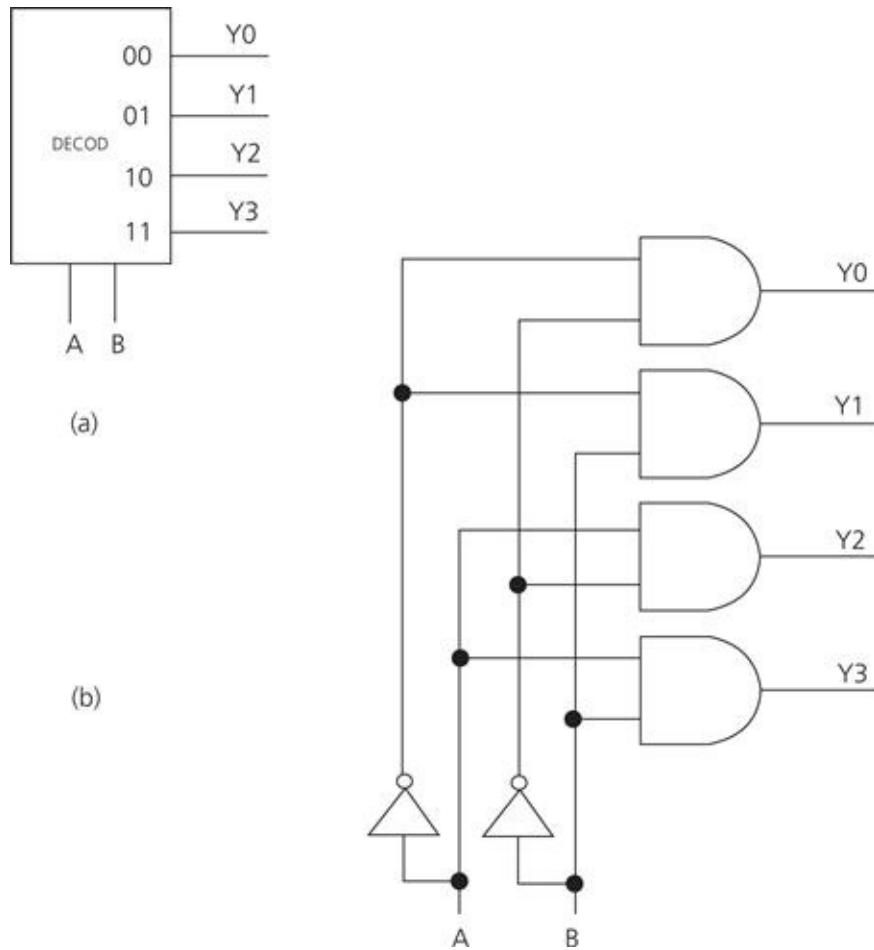
Finalmente, cabe observar que, apesar de um DEMUX realizar a função

inversa de um MUX, os seus circuitos de seleção são os mesmos (portas AND conectadas às combinações das entradas de seleção).

## Decodificadores

Um decodificador é um circuito lógico contendo várias entradas e várias saídas, que a partir da detecção de uma combinação lógica das entradas aciona apenas uma de suas saídas. Como exemplo inicial, vejamos um decodificador 1 de 4 cujo símbolo lógico é mostrado na [Figura 7.9.a](#), o qual apresenta a seguinte tabela-verdade:

A	B	Y0	Y1	Y2	Y3	
0	0	1	0	0	0	Apenas Y0 é acionado
0	1	0	1	0	0	Apenas Y1 é acionado
1	0	0	0	1	0	Apenas Y2 é acionado
1	1	0	0	0	1	Apenas Y3 é acionado



**FIGURA 7.9** Circuito decodificador 1 de 4 (DECOD 1 de 4): (a) símbolo lógico e (b) circuito lógico interno.

Um DECOD 1 de 4 possui duas entradas de seleção e quatro saídas independentes, das quais apenas uma assume nível lógico 1 (ou seja, é acionada) quando uma combinação lógica das entradas de seleção se verifica. Quando uma saída estiver acionada, as demais permanecem não acionadas (em nível lógico 0).

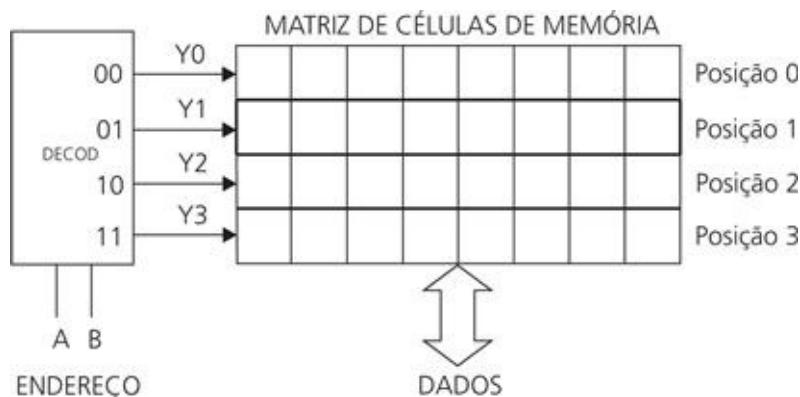
A Figura 7.9.c ilustra o circuito lógico interno que pode ser obtido diretamente da tabela-verdade anterior por meio da técnica de soma de produtos, pois cada um dos 1's das colunas são produtos fundamentais das entradas:

$$\begin{aligned} Y_0 &= (\bar{A} \cdot \bar{B}) & Y_1 &= (\bar{A} \cdot B) \\ Y_2 &= (A \cdot \bar{B}) & Y_3 &= (A \cdot B) \end{aligned}$$

Observa-se novamente que o circuito de seleção construído em torno das

portas AND é similar aos encontrados nos circuitos MUX e DEMUX. De forma análoga aos circuitos anteriormente descritos, podemos generalizar o resultado do projeto de um DECOD 1 de 4 para um número maior ou menor de entradas. Por exemplo, um DECOD 1 de 8 deve conter três entradas de seleção, e um DECOD 1 de 3 deve conter as mesmas duas entradas de seleção do circuito da [Figura 7.9.b](#), mas aí a saída Y3 e seu respectivo AND devem ser eliminados.

Circuitos decodificadores são muito empregados para detectar determinadas combinações das variáveis de entrada. A [Figura 7.10](#) mostra como acionar uma determinada posição de memória utilizando um decodificador. Neste exemplo, é mostrado um arranjo matricial de células de memória de quatro linhas de 8 bits, no qual um DECOD 1 de 4 seleciona a linha cujo conteúdo pode ser lido ou escrito de acordo com o tipo de memória. Nesse caso, as variáveis de seleção são denominadas “endereço”. Por exemplo, ao se selecionar o endereço referente à posição 1, deve-se aplicar AB = 01 de modo que apenas as células de memória correspondentes sejam acionadas (estas estão indicadas na [Figura 7.10](#) dentro do retângulo em negrito). Memórias tipo RAM permitem operações de escrita e leitura, ao passo que memórias tipo ROM permitem apenas operações de leitura. Porém, estes tipos de memórias geralmente utilizam circuitos decodificadores de endereço para definir a posição de acesso de tais operações. Maiores detalhes operacionais sobre os circuitos de memória serão vistos nos próximos capítulos.



**FIGURA 7.10** Circuito decodificador 1 de 4 (DECOD 1 de 4) empregado para acionar uma posição de memória.

Circuitos decodificadores podem também ser empregados para implementar circuitos combinacionais. No item “Expressões booleanas de soma de produtos” do [Capítulo 5](#) vimos que podemos escrever equações lógicas a partir dos 1's na

coluna de saída de uma tabela-verdade utilizando a técnica de **soma de produtos**. Considere a tabela-verdade:

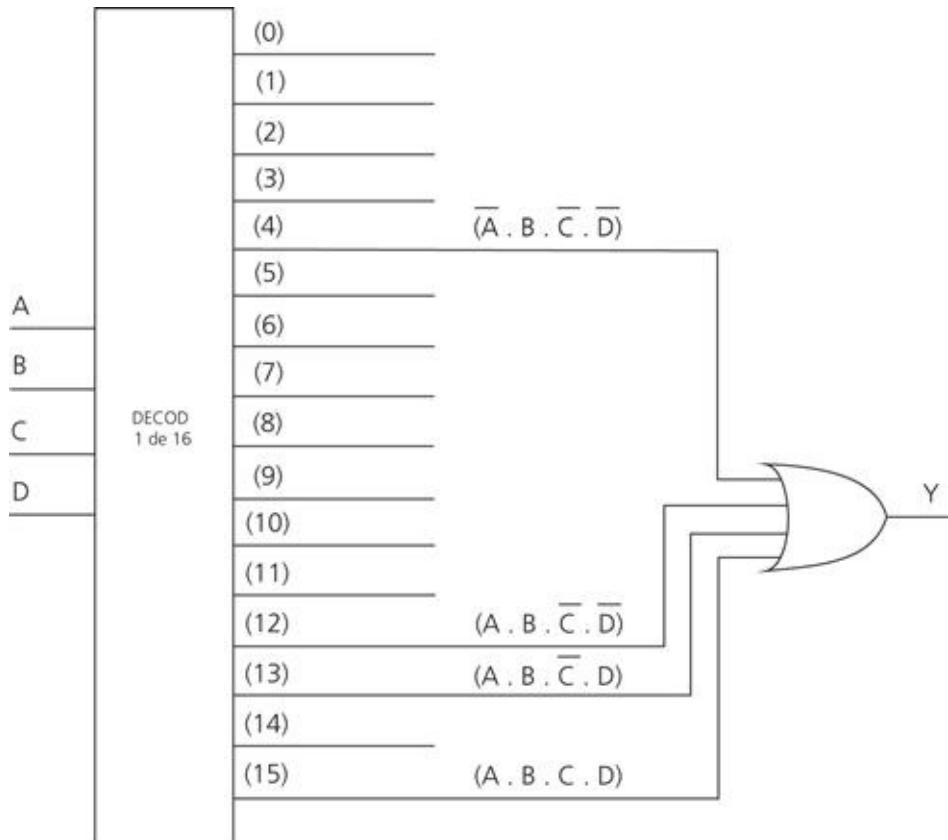
A	B	C	D	R	Posição
0	0	0	0	0	(0)
0	0	0	1	0	(1)
0	0	1	0	0	(2)
0	0	1	1	0	(3)
0	1	0	0	1	(4)
0	1	0	1	0	(5)
0	1	1	0	0	(6)
0	1	1	1	0	(7)
1	0	0	0	0	(8)
1	0	0	1	0	(9)
1	0	1	0	0	(10)
1	0	1	1	0	(11)
1	1	0	0	1	(12)
1	1	0	1	1	(13)
1	1	1	0	0	(14)
1	1	1	1	1	(15)

Pelo método de soma de produtos, pode-se escrever a seguinte equação lógica:

$$R = (\bar{A} \cdot B \cdot \bar{C} \cdot \bar{D}) + (A \cdot B \cdot \bar{C} \cdot \bar{D}) + (A \cdot B \cdot \bar{C} \cdot D) + (A \cdot B \cdot C \cdot D)$$

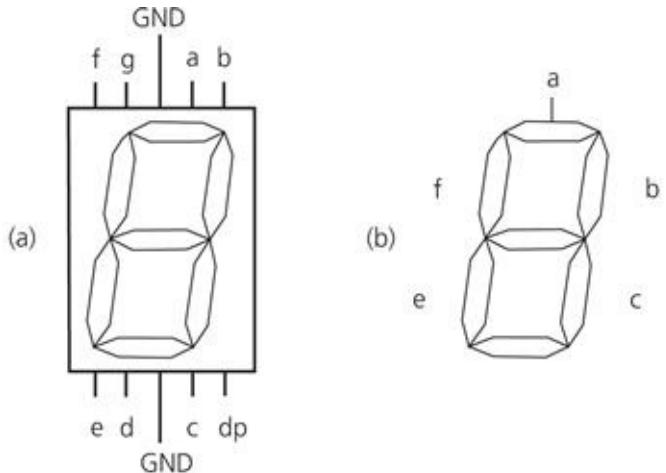
Podemos construir o circuito lógico relativo a essa equação utilizando decodificadores, e sem simplificar e utilizar portas AND, OR ou NOT. A [Figura 7.11](#) demonstra como utilizar um decodificador 1 de 16 (normalmente disponível na forma de um circuito integrado comercial) para gerar os produtos

fundamentais e apenas uni-los em uma função OR final para produzir a saída R. Essa solução para a implementação de um circuito lógico é conhecida como lógica de decodificador. Não é uma solução otimizada que utiliza a menor quantidade de portas lógicas, mas constitui uma forma prática muito empregada em bancada, devido a sua rápida execução.



**FIGURA 7.11** Circuito decodificador 1 de 16 (DECOD 1 de 16) utilizado para implementar lógica de decodificador.

Segundo a definição de circuitos decodificadores, apenas *uma* saída deve ser ativada para cada combinação das entradas. Porém, existe uma exceção, por questão histórica e de uso, nos decodificadores aplicados aos mostradores de sete segmentos, conforme mostra a [Figura 7.12](#).



**FIGURA 7.12** Mostrador de sete segmentos: (a) *layout* exterior na versão anodo comum, (b) identificação dos segmentos e (c) utilização para escrita de algarismos decimais de 0 a 9.

O mostrador indicado na [Figura 7.12.a](#) é construído com segmentos de LED, que quando acesos montam na superfície a imagem de um certo número decimal; os segmentos não utilizados para desenhar o número devem permanecer apagados, e elemento “dp” (*decimal point*) pode ser aceso em aplicações que exibam valores fracionários. O mostrador é ligado a um terminal comum GND, disponível em ambos os lados, que é utilizado como terminal de retorno para alimentação dos LEDs (este é um assunto não coberto neste texto, mas o estudante encontrará facilmente maiores detalhes em livros de eletrônica básica).

Antes de mais nada, vamos observar um detalhe muito importante: até o momento nada foi definido quanto ao formato das combinações das variáveis de entrada de um decodificador. Aparentemente a escolha é livre. Entretanto, aplicações envolvendo números sugerem que se produza uma conversão de decimal para binário ao se fornecer uma informação numérica à entrada de um circuito lógico. Na verdade, decodificadores (e codificadores, como veremos no próximo item) estão associados à manipulação de códigos, que são combinações lógicas universalmente padronizadas segundo determinadas aplicações. Veja mais detalhes sobre alguns dos códigos mais utilizados no [Capítulo 8](#).

A tabela a seguir define o código BCD (*Binary Coded Decimal*), utilizado na representação dos números decimais de 0 a 9 em aplicações envolvendo mostradores de sete segmentos.

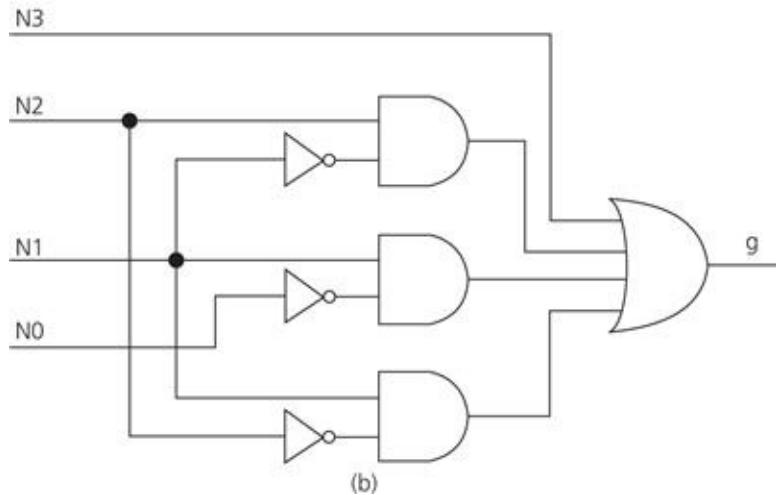
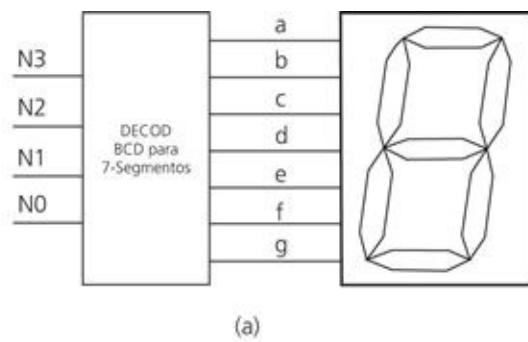
**Tabela 7.6**  
**Tabela do código BCD**

N3	N2	N1	N0	Número decimal
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	Não BCD
1	0	1	1	Não BCD
1	1	0	0	Não BCD
1	1	0	1	Não BCD
1	1	1	0	Não BCD
1	1	1	1	Não BCD

Observe que o código BCD é composto pela sequência {N3, N3, N1, N0}, e só é definido nas 10 primeiras combinações. As demais seis combinações são denominadas não BCD, e portanto nunca podem ocorrer em aplicações numéricas.

Nesse contexto, o decodificador BCD para sete segmentos deve ser projetado para acionar os diversos segmentos, para que no mostrador a forma dos números decimais de 0 a 9 seja desenhada quando os respectivos segmentos forem acesos, de acordo com a [Figura 7.12.c](#).

Vamos exemplificar projetando o circuito lógico que aciona o segmento g mostrado na [Figura 7.13](#), obedecendo o seguinte critério:  $g = 1$  significa segmento g aceso, e  $g = 0$ , segmento g apagado.



**FIGURA 7.13** Decodificador BCD para sete segmentos: (a) diagrama de blocos, (b) circuito lógico que implementa o acionamento do segmento g.

A tabela-verdade a seguir relaciona a variável de saída g às variáveis de entrada N3, N2, N1 e N0 de um número codificado em BCD.

<b>BCD</b>	<b>N3</b>	<b>N2</b>	<b>N1</b>	<b>N0</b>	<b>g</b>
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
-	1	0	1	0	X
-	1	0	1	1	X
-	1	1	0	0	X
-	1	1	0	1	X
-	1	1	1	0	X
-	1	1	1	1	X

Note que na [Figura 7.12.c](#) o segmento g deve ser acionado para que os números 2, 3, 4, 5, 6, 8 e 9 possam ser escritos. Portanto, estas posições na tabela-verdade devem receber 1, e as demais (0, 1 e 7), 0. Entretanto, o código BCD não é definido nas últimas seis linhas da tabela-verdade. Nestas posições,

marcamos um X, que significa “tanto faz” – ou seja, como são combinações numericamente impossíveis de ocorrer, tanto faz assinalar 1 ou 0.

A existência de posições não definidas em tabelas-verdade são típicas de aplicações envolvendo código BCD. Estas posições são conhecidas como *problemas parcialmente definidos*, que resultam em condições sem importância marcadas como X (em inglês, *don't care conditions*).

O próximo passo agora é fazer o mapa K da variável g, como mostrado a seguir.

	$\overline{N1} \overline{N0}$	$\overline{N1} N0$	$N1 \overline{N0}$	$N1 \overline{N0}$
$\overline{N3} \overline{N2}$	0	0	1	1
$\overline{N3} N2$	1	1	0	1
$N3 \overline{N2}$	X	X	X	X
$N3 \overline{N2}$	1	1	X	X

As condições X, exatamente porque podem assumir qualquer valor lógico, são muito úteis nos emparelhamentos do mapa K, tal como mostrado no mapa anterior. Por exemplo, podemos emparelhar todos os X com os dois 1's no canto inferior direito, que passam de um par para um octeto, eliminando assim três variáveis. O emparelhamento dos demais 1's do mapa também se beneficia da proximidade dos X, e dessa forma, nesse particular exemplo, temos X = 1 para todas as seis ocorrências. No mapa K anterior, podemos identificar três quadras e um octeto, resultando na seguinte equação simplificada:

$$g = (\overline{N2} \cdot \overline{N1}) + (N1 \cdot \overline{N0}) + (\overline{N2} \cdot N1) + N3$$

O circuito lógico que aciona o segmento g de um mostrador de sete segmentos é mostrado na [Figura 7.13.b](#). Os demais segmentos são projetados de maneira semelhante. O conjunto dos sete circuitos lógicos compõe o decodificador BCD para sete segmentos mostrado na [Figura 7.13.a](#).

## Codificadores

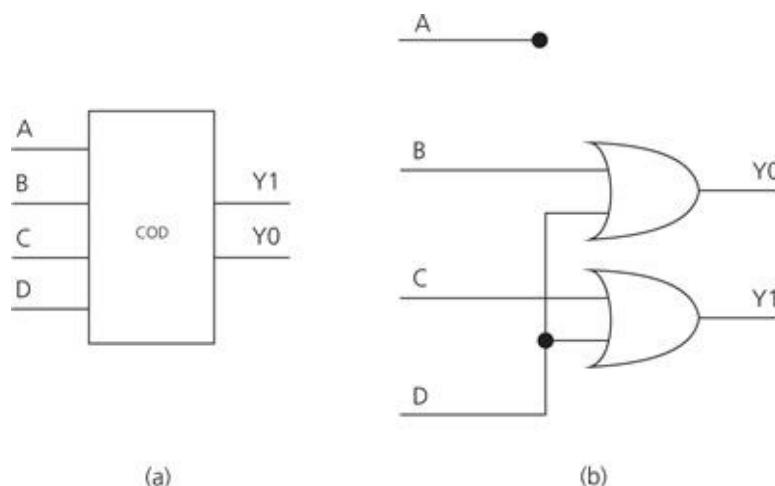
Um codificador é um circuito que realiza a função inversa de um decodificador.

Ele possui várias entradas e várias saídas, e quando apenas uma das entradas é acionada, uma única combinação lógica nas saídas é obtida. A Figura 7.14.a apresenta o símbolo lógico de um codificador 4 para 2, cuja funcionalidade é descrita na tabela-verdade a seguir.

**Tabela 7.7**

**Tabela-verdade de um codificador 4 para 2**

A	B	C	D	Y1	Y0	
1	0	0	0	0	0	Apenas A é acionada
0	1	0	0	0	1	Apenas B é acionada
0	0	1	0	1	0	Apenas C é acionada
0	0	0	1	1	1	Apenas D é acionada



**FIGURA 7.14** Circuito codificador 4 para 2 (COD 4:2): (a) símbolo lógico e (b) circuito lógico interno.

O funcionamento correto de um codificador é garantido sempre que uma das entradas estiver acionada. Nestas condições, não existe liberdade de combinação das entradas além das que estão mostradas na tabela-verdade, ou seja, as demais combinações nunca ocorrem. Desta forma, o projeto lógico fica bastante

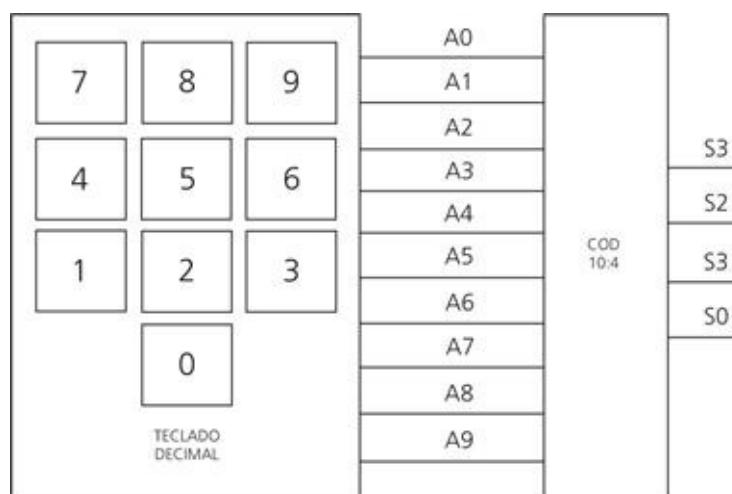
simplificado. Observe na tabela-verdade anterior que  $Y_1$  vale 1 somente se C ou D estiverem acionadas. Da mesma forma,  $Y_0$  vale 1 somente se B ou D estiverem acionadas (estranhamente, A não participa do circuito). Assim, as equações lógicas podem ser simplesmente escritas como:

$$Y_0 = (B + D) \quad Y_1 = (C + D)$$

A [Figura 7.14.b](#) mostra o circuito lógico do codificador 4 para 2 (COD 4:2). Note que esse circuito possui duas importantes limitações:

- a) Ele pode gerar uma combinação errada nas saídas se mais de uma entrada for acionada ao mesmo tempo.
- b) Ele não considera a possibilidade de que nenhuma das entradas esteja acionada em um determinado momento.

Vamos agora aplicar o conceito de circuito codificador para resolver o problema prático de capturar um número digitado em um teclado numérico de 0 a 9. A [Figura 7.15](#) ilustra o diagrama de blocos desta aplicação. Cada número em formato decimal está desenhado sobre uma tecla que aciona um interruptor de pressão. Quando uma tecla ( $i$ ) é acionada, a variável correspondente  $A(i)$  assume valor lógico 1 ( $i$  é um índice na faixa de 0 a 9 correspondente ao número decimal da tecla). Teclas não acionadas resultam em nível lógico 0. Considere também que apenas uma tecla é apertada a cada vez, tornando as variáveis  $A(i)$  mutuamente exclusivas.



**FIGURA 7.15** Circuito codificador 10 para 4 (COD 10:4) para aplicação na captura numérica de um teclado decimal.

Nessa aplicação, a associação das saídas  $\{S_3, S_2, S_1, S_0\}$  se refere ao código BCD correspondente à tecla acionada. Portanto, “0000” corresponde ao número decimal 0; “0001” corresponde ao número 1; e assim sucessivamente. Nestas condições, temos de projetar um codificador de 10 entradas para quatro saídas (COD 10:4). Veja a seguir a tabela-verdade deste problema.

**Tabela 7.8**

**Tabela-verdade do COD 10:4**

Nº	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	S3	S2	S1	S0
-	0	0	0	0	0	0	0	0	0	0	1	1	1	1
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	1
2	0	0	1	0	0	0	0	0	0	0	0	0	1	0
3	0	0	0	1	0	0	0	0	0	0	0	0	1	1
4	0	0	0	0	1	0	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	0	0	0	0	1	0	1
6	0	0	0	0	0	0	1	0	0	0	0	1	1	0
7	0	0	0	0	0	0	0	1	0	0	0	1	1	1
8	0	0	0	0	0	0	0	0	1	0	1	0	0	0
9	0	0	0	0	0	0	0	0	0	1	1	0	0	1

Na aplicação de teclado decimal pode ocorrer a situação em que nenhuma tecla seja apertada (teclado em repouso). Esta situação é resolvida na tabela-verdade anterior logo na primeira linha, quando para todos  $A(i) = 0$  forçamos uma combinação não BCD para as saídas  $\{S_3, S_2, S_1, S_0\}$  igual a “1111”.

Essa situação especial é facilmente detectada pela ocorrência simultânea de todas as entradas  $A(i)$  iguais a 0.

$$Q = \overline{A_0 \cdot A_1 \cdot A_2 \cdot A_3 \cdot A_4 \cdot A_5 \cdot A_6 \cdot A_7 \cdot A_8 \cdot A_9}$$

$$= \overline{(A_0 + A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8 + A_9)}$$

Da mesma forma como no projeto do circuito codificador da [Figura 7.14](#), podemos considerar as entradas A(i) mutuamente exclusivas (nunca há duas ou mais teclas acionadas ao mesmo tempo), e assim simplificar o equacionamento das variáveis de saída:

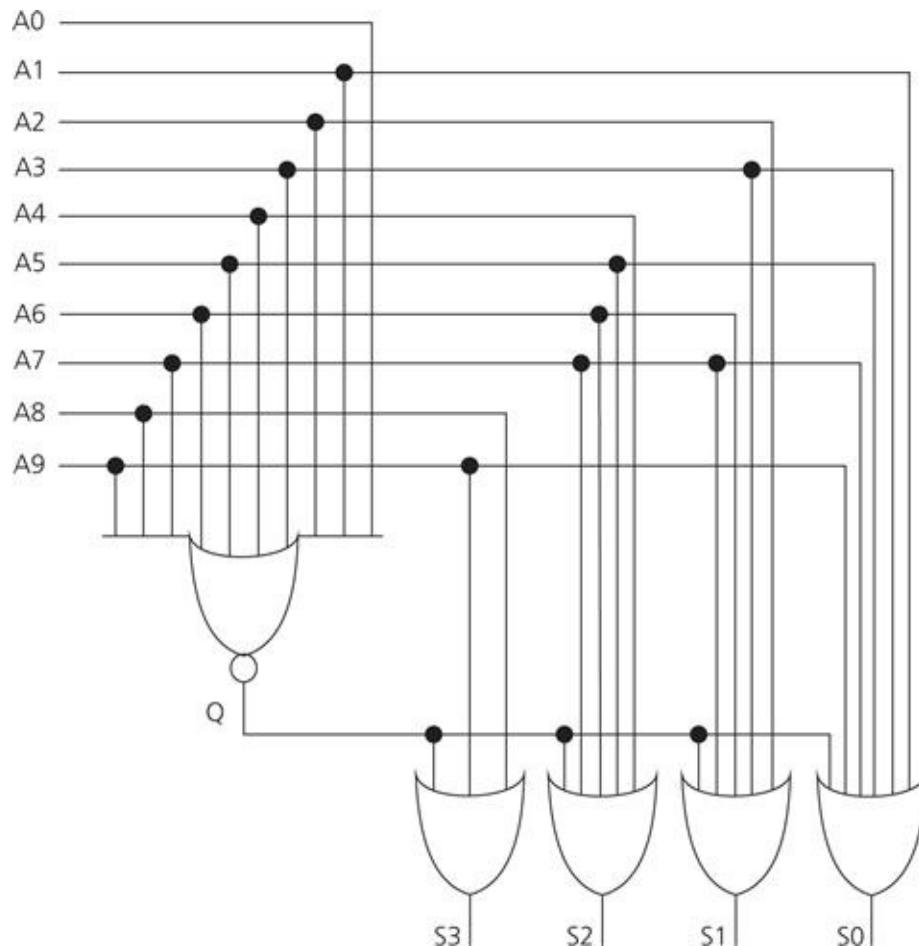
$$S_3 = A_9 + A_8 + Q$$

$$S_2 = A_7 + A_6 + A_5 + A_4 + Q$$

$$S_1 = A_7 + A_6 + A_3 + A_2 + Q$$

$$S_0 = A_9 + A_7 + A_5 + A_3 + A_1 + Q$$

A [Figura 7.16](#) ilustra o circuito lógico construído com as equações anteriores.



**FIGURA 7.16** Circuito lógico interno de um codificador 10 para 4 (COD 10:4) para aplicação na captura numérica.

A variável Q é importante para criar o código não BCD quando se considera o teclado em repouso. Ela também indica com alto nível lógico que não há nenhuma tecla acionada.



## O Que Vem Depois

Números são representados por códigos, e portanto requerem circuitos conversores de código. Embora circuitos lógicos sejam extremamente robustos, alguns erros podem ocorrer (durante processos de transmissão digital, por exemplo), e iremos aprender como detectá-los. Os circuitos aritméticos são uma classe especial de circuitos combinacionais. Iremos analisá-los em detalhes no próximo capítulo, no qual veremos como construir um circuito somador binário com base no algoritmo humano da soma, e, em seguida, como transformá-lo em um circuito somador-subtrator, que faz uso do conceito de complemento de 2 visto no [Capítulo 2](#).

---

## CAPÍTULO 8

---

# Circuitos lógicos combinacionais (segunda parte)

---

### Objetivos Do Capítulo

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Compreender o conceito de código para representar números.
- Compreender o princípio de operação e saber projetar circuitos conversores de código, geradores e verificadores de paridade.
- Entender o algoritmo da soma binária.
- Compreender o funcionamento do circuito somador de 1 bit e estender este conceito para a construção de somadores de N bits.
- Compreender como transformar um circuito somador em um somador-subtrator, cujas operações de subtração são conduzidas utilizando complemento de 2.



### Apresentação

Veremos neste capítulo diversos tipos de códigos binários usados para representar números, bem como os circuitos lógicos utilizados nas transformações entre códigos.



O problema de detecção e correção de erros será discutido juntamente com os circuitos geradores e verificadores de paridade. Por fim, iremos estudar como é possível construir um circuito lógico para realizar operações aritméticas com números binários.

## Fundamentos

### **Conversores de código**

Os circuitos conversores de código transformam uma determinada combinação lógica nas entradas (em um determinado código) em uma combinação lógica diferente nas saídas (segundo um outro código). Normalmente estes circuitos estão associados a transformações de formato numérico. Veja na [Tabela 8.1](#) diversos códigos empregados para a representação de números.

---

#### **Tabela 8.1**

#### **Códigos empregados para representação numérica**

---

Decimal	BCD	Excesso de 3	Binário	Gray	Johnson
0	0000	0011	0000	0000	00000
1	0001	0100	0001	0001	00001
2	0010	0101	0010	0011	00011
3	0011	0110	0011	0010	00111
4	0100	0111	0100	0110	01111
5	0101	1000	0101	0111	11111
6	0110	1001	0110	0101	11110
7	0111	1010	0111	0100	11100
8	1000	1011	1000	1100	11000
9	1001	1100	1001	1101	10000
10	—	—	1010	1111	—
11	—	—	1011	1110	—
12	—	—	1100	1010	—
13	—	—	1101	1011	—
14	—	—	1110	1001	—
15	—	—	1111	1000	—

Na segunda coluna, podemos ver o nosso já conhecido código BCD. O código excesso de 3, como o nome diz, adiciona a quantidade 3 a toda posição do código BCD. Já o código binário corresponde à representação da quantidade decimal na base 2, e portanto não se limita aos números de 0 a 9. Reveja no [Capítulo 2](#) como converter um número de base de decimal para base binária. O código Gray tem a propriedade de modificar apenas 1 bit quando deslocamos o número uma unidade para mais ou para menos, e é muito empregado em processos de contagem numérica. Finalmente, temos o código Johnson, que tem a mesma propriedade do código Gray mas utiliza 5 bits.

Como exemplo, vejamos a conversão de código binário para o código Gray, como mostrado na tabela-verdade a seguir.

<b>B3</b>	<b>B2</b>	<b>B1</b>	<b>B0</b>	<b>G3</b>	<b>G2</b>	<b>G1</b>	<b>G0</b>
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0

1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

As variáveis B(i) são as entradas do código em binário, e as variáveis G(i) são as saídas codificadas em Gray. Cada coluna G(i) é uma função lógica das quatro entradas B(i), e independente das demais saídas. A seguir temos os quatro mapas K para cada bit de saída.

1. Saída G3:

$\overline{B1} \overline{B0}$	$\overline{B1} B0$	$B1 \overline{B0}$	$B1 B\overline{0}$
$\overline{B3} \overline{B2}$	0	0	0
$\overline{B3} B2$	0	0	0
$B3 B2$	1	1	1
$B3 \overline{B2}$	1	1	1

$$G3 = B3$$

2. Saída G2:

	$\overline{B1} \overline{B0}$	$\overline{B1} B0$	B1 B0	B1 $\overline{B0}$
$\overline{B3} \overline{B2}$	0	0	0	0
$\overline{B3} B2$	1	1	1	1
B3 B2	0	0	0	0
B3 $\overline{B2}$	1	1	1	1

$$G2 = (\overline{B3} \cdot B2) + (B3 \cdot \overline{B2}) = B3 \oplus B2$$

3. Saída G1:

	$\overline{B1} \overline{B0}$	$\overline{B1} B0$	B1 B0	B1 $\overline{B0}$
$\overline{B3} \overline{B2}$	0	0	1	1
$\overline{B3} B2$	1	1	0	0
B3 B2	1	1	0	0
B3 $\overline{B2}$	0	0	1	1

$$G1 = (B2 \cdot \overline{B1}) + (\overline{B2} \cdot B1) = B2 \oplus B1$$

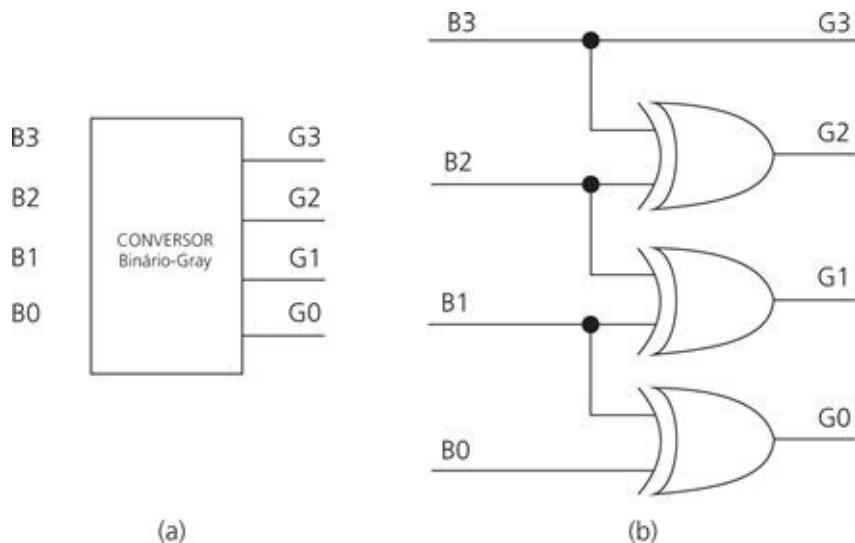
4. Saída G0:

	$\overline{B1} \overline{B0}$	$\overline{B1} B0$	$B1 B0$	$B1 \overline{B0}$
$\overline{B3} \overline{B2}$	0	1	0	1
$\overline{B3} B2$	0	1	0	1
$B3 B2$	0	1	0	1
$B3 \overline{B2}$	0	1	0	1

$$G0 = (\overline{B1} \cdot B0) + (B1 \cdot \overline{B0}) = B1 \oplus B0$$

Note que em G2, G1 e G0 as equações foram identificadas como as funções XOR, conforme vimos no Exemplo 11 no item “Casos especiais de Mapas de Karnaugh” do [Capítulo 6](#).

A [Figura 8.1](#) ilustra o símbolo lógico e o circuito lógico interno do conversor de código binário-Gray de 4 bits. Analisando-se a forma de ligação das portas XOR, percebe-se facilmente a lei de formação deste conversor, e portanto essa solução pode ser estendida a conversores binário-Gray de qualquer quantidade de bits.



**FIGURA 8.1** Circuito conversor de código binário-Gray de 4 bits: (a) símbolo lógico e (b) circuito lógico interno.

## Circuitos Geradores E Verificadores De Paridade

A função OU-exclusivo de duas variáveis descrita no item “Funções lógicas derivadas” do [Capítulo 4](#) detecta a exclusividade, ou seja, a ocorrência do valor lógico 1 em apenas uma das entradas. Agora, iremos construir a tabela-verdade de um XOR de quatro entradas, implementando-o de forma associativa conforme a equação a seguir:

$$Y = A \oplus B \oplus C \oplus D = (A \oplus B) \oplus (C \oplus D)$$

A	B	C	D	$A \oplus B$	$C \oplus D$	Y
0	0	0	0	0	0	0
0	0	0	1	0	1	1
0	0	1	0	0	1	1
0	0	1	1	0	0	0
0	1	0	0	1	0	1
0	1	0	1	1	1	0
0	1	1	0	1	1	0
0	1	1	1	1	0	1
1	0	0	0	1	0	1
1	0	0	1	1	1	0
1	0	1	0	1	1	0
1	0	1	1	1	0	1
1	1	0	0	0	0	0
1	1	0	1	0	1	1
1	1	1	0	0	1	1
1	1	1	1	0	0	0

Nas entradas temos:

Qte. ímpar de 1's

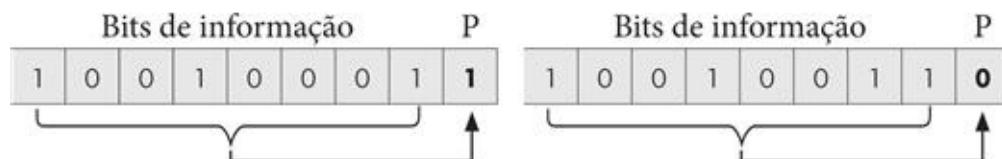
Analisando-se a saída Y em relação às entradas, vemos que a propriedade de

exclusividade não mais se aplica. Entretanto, outra propriedade pode ser observada: Y será igual a 1 sempre que houver uma quantidade ímpar de 1's nas entradas.

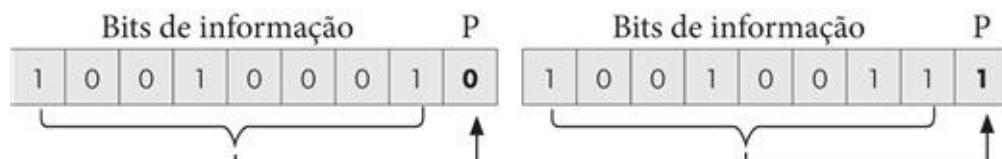
Essa nova propriedade da função XOR, mais genérica, pode ser muito útil para criar uma informação complementar agregada ao conjunto de bits presentes nas entradas, o qual indica a quantidade de 1's em um dado instante ou lugar. Em um segundo momento, ou em um lugar remoto, essa informação pode ser recalculada e comparada com a informação original para que se verifique a consistência desse conjunto de bits.

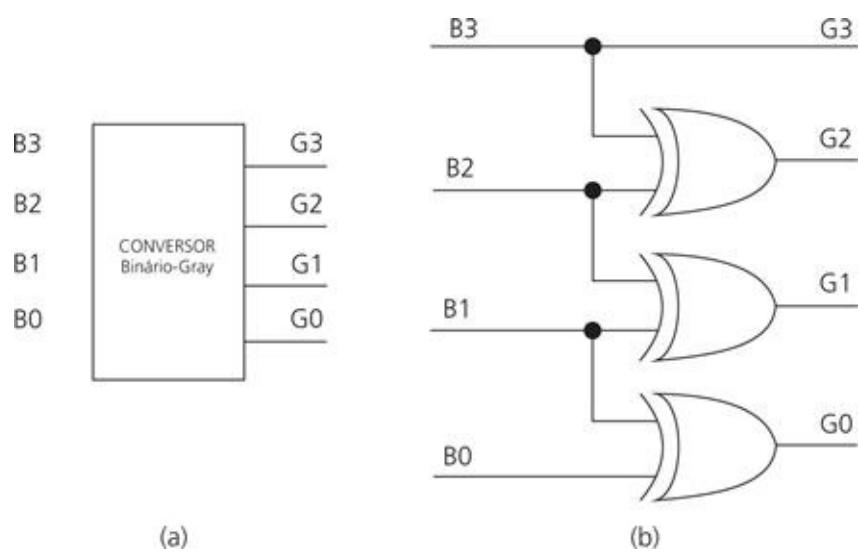
Define-se bit de paridade P como um bit extra agregado a um grupo de bits de informação (ou dados) com a finalidade de indicar a existência de um número total de 1's par ou ímpar dentro do conjunto final (grupo de bits mais bit de paridade). A paridade é “par” se o valor lógico de P for escolhido para totalizar uma quantidade par de 1's no conjunto; caso contrário, a paridade será “ímpar”. Veja a seguir alguns exemplos de cálculo do bit de paridade para grupos de 8 bits de informação:

a) Paridade par

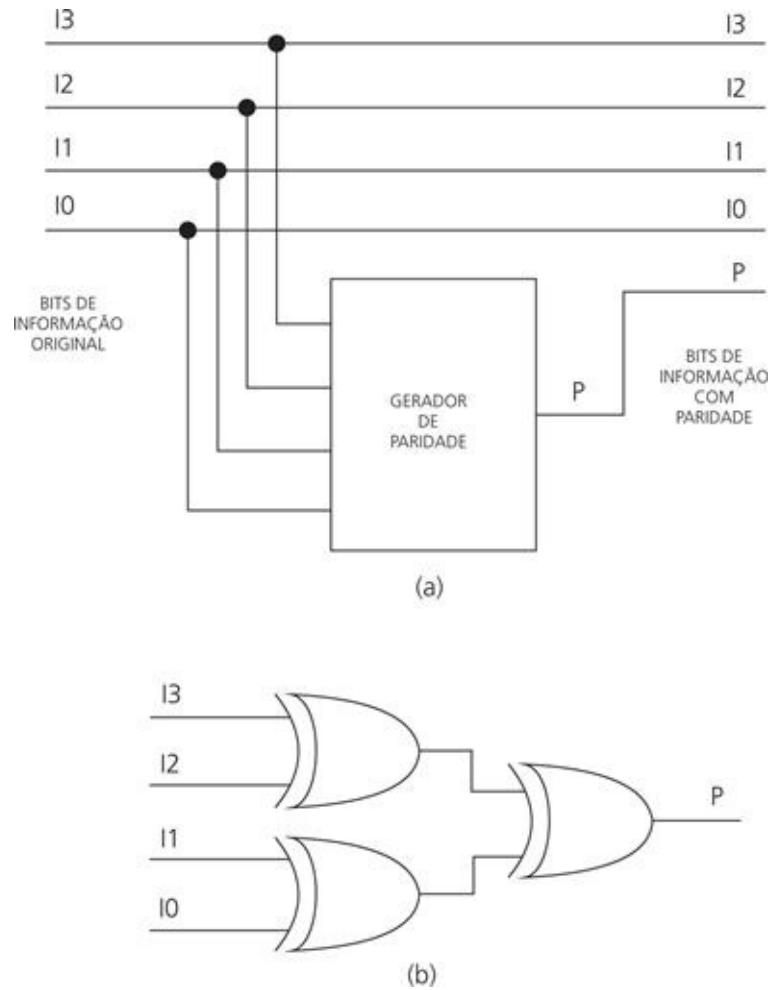


b) Paridade ímpar





Uma função OU-exclusivo de N entradas implementada associativamente pode ser empregada como um gerador de paridade, tal como mostra a [Figura 8.2](#).



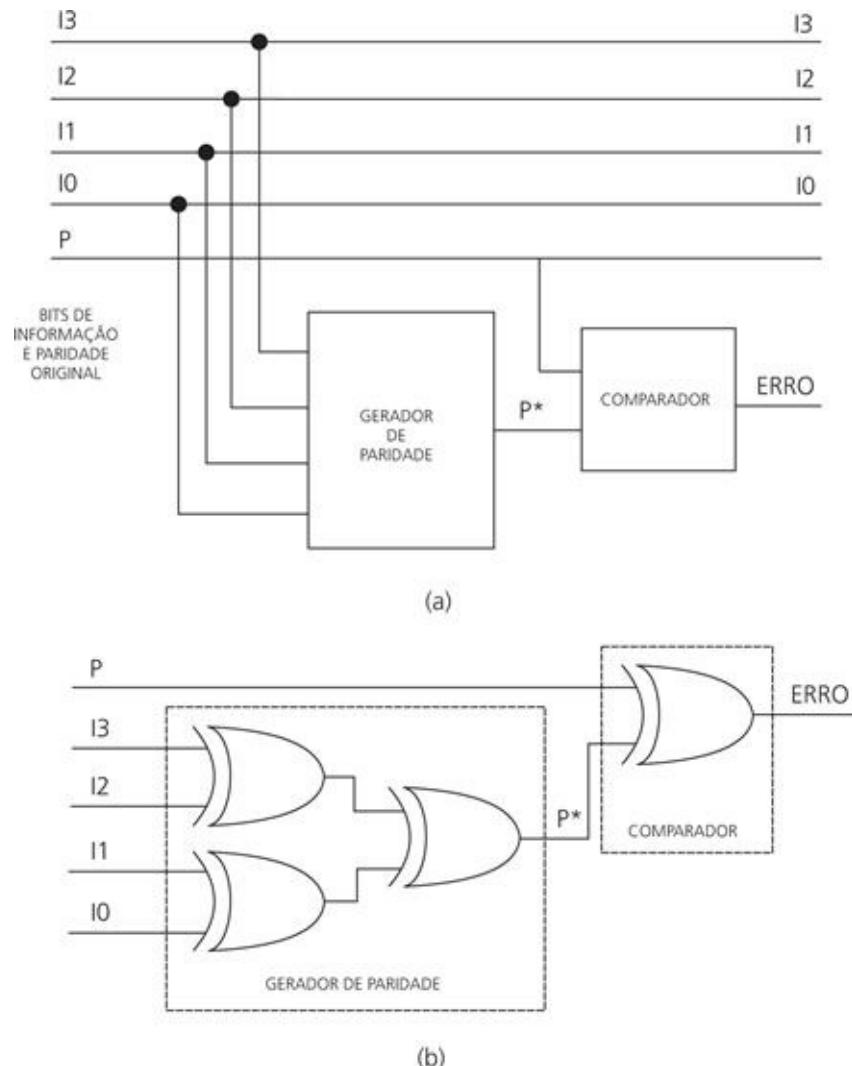
**FIGURA 8.2** Circuito gerador de paridade de 4 bits: (a) diagrama de blocos e (b) circuito lógico interno.

Note que o gerador de paridade não modifica os bits de informação da entrada; ele deve atuar em paralelo e apenas agregar o bit P ao conjunto final.

A inclusão de um bit de paridade é útil em aplicações de transmissão de dados, nas quais pode ocorrer um erro em um dos bits devido às interferências externas durante o percurso de transmissão. Erros em dados digitais correspondem à troca de valor lógico de um dos bits aleatoriamente. Portanto, um erro no conjunto de bits transmitido (bits de informação mais bit de paridade) fará com que a paridade calculada na recepção seja diferente da paridade original. Desta forma, podemos afirmar que a inclusão do bit de paridade permite a detecção de um único erro. Na verdade, o bit de paridade indicaria uma quantidade ímpar de erros (1, 3, 5 etc.), ao passo que uma quantidade par de erros (2, 4, 6 etc.) passaria despercebida. Entretanto, estudos

estatísticos de transmissão digital indicam que a probabilidade de que dois ou mais erros ocorram em um mesmo conjunto de 8 bits é muito menor do que a probabilidade de que um único erro ocorra. Portanto, a inclusão de apenas um bit de paridade é útil em diversas aplicações. Métodos mais complexos de detecção e correção de erros baseados em paridade existem (como o código de Hamming), mas não serão tratados neste texto.

A [Figura 8.3](#) traz um circuito verificador de paridade, que recalcula a paridade do lado da recepção (denominada  $P_*$ ) e a compara com a paridade  $P$  original.



**FIGURA 8.3** Circuito verificador de paridade de 4 bits: (a) diagrama de blocos e (b) circuito lógico interno.

O circuito verificador de paridade é composto por um gerador de paridade

(idêntico ao da [Figura 8.2](#)) e um comparador de bits. Note que neste circuito desejamos acionar a variável lógica ERRO caso as paridades  $P$  e  $P^*$  sejam diferentes. Veja a tabela-verdade a seguir.

<b>P</b>	<b>P*</b>	<b>ERRO</b>
0	0	0
0	1	1
1	0	1
1	1	0

Reconhece-se facilmente a função OU-exclusivo como aquela que detecta a desigualdade entre dois bits (veja que esta é uma outra forma de perceber a propriedade da exclusividade):

$$ERRO = P \oplus P^*$$

Os exemplos de circuitos gerador e verificador de paridade apresentados nas [Figura 8.2](#) e [8.3](#) aplicam-se a quatro bits de informação. Este resultado pode ser estendido a qualquer quantidade de bits, seguindo-se a mesma lei de formação observada para o gerador de paridade: OU-exclusivo de todas as variáveis de entrada associadas duas a duas.

## Círcuito Somador

Vamos imaginar um circuito lógico combinacional para a soma de dois números A e B, cada um com 8 bits, resultando em um número C de 8 bits. Neste caso, teremos 16 entradas (cuja tabela-verdade possui 65.536 linhas...) e 8 saídas. Evidentemente, o projeto lógico deste circuito é uma tarefa muito difícil pelos métodos descritos nos capítulos anteriores. Portanto, devemos buscar uma forma alternativa de projeto.

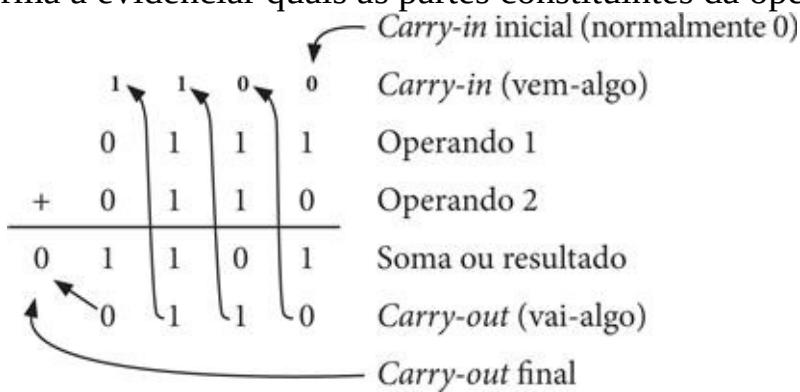
Vejamos inicialmente como uma operação de soma com números binários é realizada manualmente, da mesma forma como somamos dois números decimais.

$$\begin{array}{r}
 7 \\
 + 6 \\
 \hline
 13
 \end{array}
 \qquad
 \begin{array}{r}
 1 \quad 1 \\
 0 \quad 1 \quad 1 \quad 1 \\
 + 0 \quad 1 \quad 1 \quad 0 \\
 \hline
 1 \quad 1 \quad 0 \quad 1
 \end{array}$$

A soma de dois bits pode resultar nos seguintes casos:

$0 + 0 =$	0	
$0 + 1 =$	1	
$1 + 0 =$	1	
$1 + 1 =$	10	Resulta 2, coloca-se 0 e vai 1
$1 + 1 + 1 =$	11	Vem 1, resulta 3, coloca-se 1 e vai 1

Uma forma de resolver o problema da soma binária é analisar a forma humana de calcular a soma. Veja o mesmo exemplo com um pouco mais de detalhe, de forma a evidenciar quais as partes constituintes da operação.



## Módulo somador binário de 1 bit

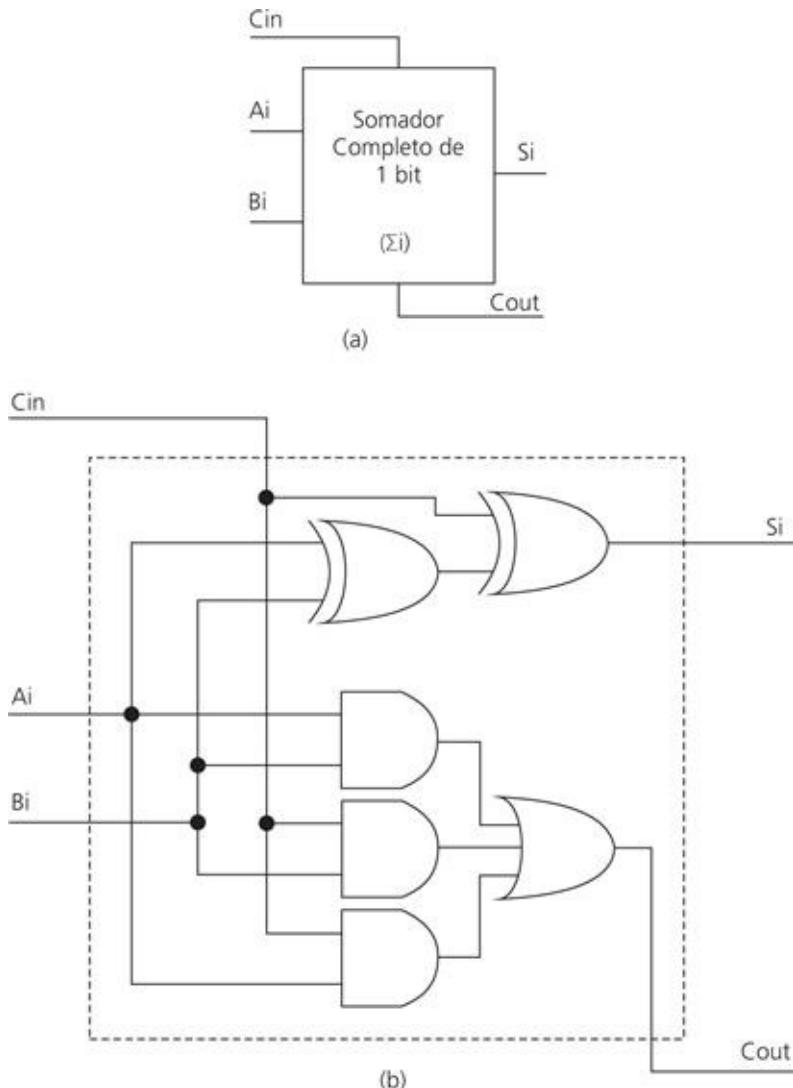
Antes de mais nada, vamos analisar o algoritmo humano de adição, que quebra o problema da soma por partes:

- Inicia-se pela coluna mais à direita.
  - Somam-se os dígitos da primeira casa, anotando-se o resultado e decidindo-se por transportar o vai-um para a coluna seguinte.
  - Somam-se os dígitos da segunda casa mais o vem-um gerado na coluna anterior, anotando-se o resultado e decidindo-se por transportar o vai-um para a coluna seguinte.
  - Continua-se aplicando o passo (c) a cada coluna até que se atinja a última coluna.
  - Se houver um último vai-um, ele é anotado como casa extra à esquerda.
- Perceba que foi exatamente esse o algoritmo empregado no exemplo binário

anterior. Observemos agora que, para cada coluna, a operação de soma é de fato realizada sobre três operandos e resulta no cálculo de dois resultados:

- Operação:  $(\text{carry-in}) + (\text{bit do operando 1}) + (\text{bit do operando 2})$
- Resultados: (bit da Soma) e (carry-out)

Essa constatação nos permite propor um circuito lógico com três entradas e duas saídas, conforme ilustrado na [Figura 8.4.a](#) e seguindo a tabela-verdade mostrada a seguir.



**FIGURA 8.4** Somador binário de 1 bit: (a) símbolo lógico e (b) circuito lógico interno.

<b>Cin</b>	<b>Ai</b>	<b>Bi</b>	<b>Cout</b>	<b>Si</b>
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Note que as colunas das saídas Cout e Si são preenchidas raciocinando-se matematicamente em relação às entradas. Por exemplo, na última linha, temos um caso de A, B e vem-um valendo 1, cuja soma vale 3 (ou “11” em binário), e que deve ter como resultado Si e Cout valendo 1.

Observe que esse circuito é puramente combinacional, e pode ser projetado pelas técnicas que foram descritas no [Capítulo 7](#). Da tabela-verdade, pode-se construir os mapas de Karnaugh para as saídas Cout e Si, como mostrado a seguir.

1. Para Cout:

	$\bar{B}_i$	$B_i$
$\bar{C}_{in}$ $\bar{A}_i$	0	0
$\bar{C}_{in}$ $A_i$	0	1
$C_{in}$ $\bar{A}_i$	1	1
$C_{in}$ $A_i$	0	1

$$Cout = (C_{in} \cdot A_i) + (C_{in} \cdot B_i) + (A_i \cdot B_i)$$

2. Para Si:

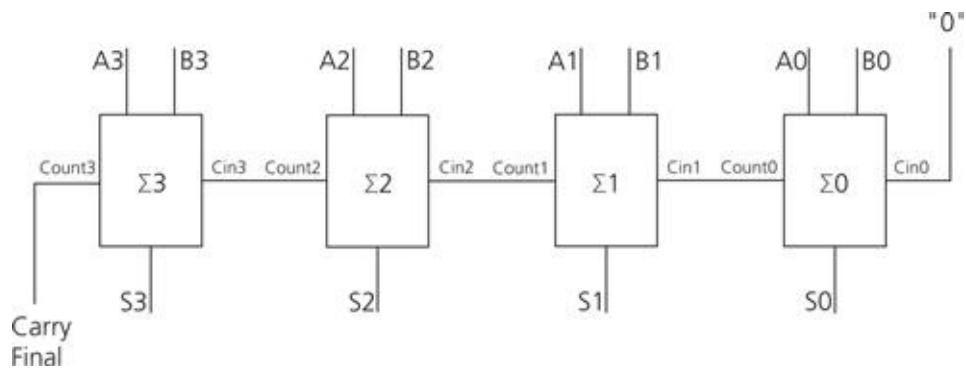
	$\overline{Bi}$	Bi
$\overline{Cin}$ Ai	0	1
$\overline{Cin}$ Ai	1	0
Cin Ai	0	1
Cin $\overline{Ai}$	1	0

$$Si = Cin \oplus (Ai \oplus Bi)$$

Veja no mapa K de Si que a existência de 1's na diagonal indica a função OU-exclusivo, conforme discutido no item “Casos especiais de Mapas de Karnaugh”, Capítulo 6. A Figura 8.4.b mostra o circuito lógico do módulo somador binário de 1 bit construído com base nas equações de Si e Cout.

## Somador de N bits

Para construir um somador de vários bits, necessitamos apenas justapor N módulos somadores de 1 bit, garantindo na interligação dos mesmos que o carry-out (i) seja conectado ao carry-in ( $i + 1$ ). A Figura 8.5 apresenta um exemplo de um somador de 4 bits.



**FIGURA 8.5** Somador binário de 4 bits.

Note que o sinal Cin0 não tem utilidade no circuito e deve ser ligado a 0 para não influenciar o resultado S0. Na outra extremidade, o Cout3 é denominado carry final e serve para indicar que houve um “vai-um” além da posição MSB.

Essa estrutura pode ser expandida para N bits, bastando justapor a quantidade necessária de módulos à esquerda.

## Círculo somador-subtrator

Poderíamos realizar agora o projeto de um módulo subtrator de 1 bit e conectá-lo assim como fizemos na [Figura 8.5](#). Entretanto, é possível utilizar o circuito somador para realizar subtrações se os bits B(i) forem convertidos para complemento de 2, de tal forma que:

$$A - B = A + (-B)_{c_2}$$

Reveja no item “Adição e subtração de números binários sinalizados” do [Capítulo 3](#) como realizar operações de subtração em complemento de 2. Sabemos que, para converter um número binário positivo para complemento de 2, precisamos dos seguintes passos:

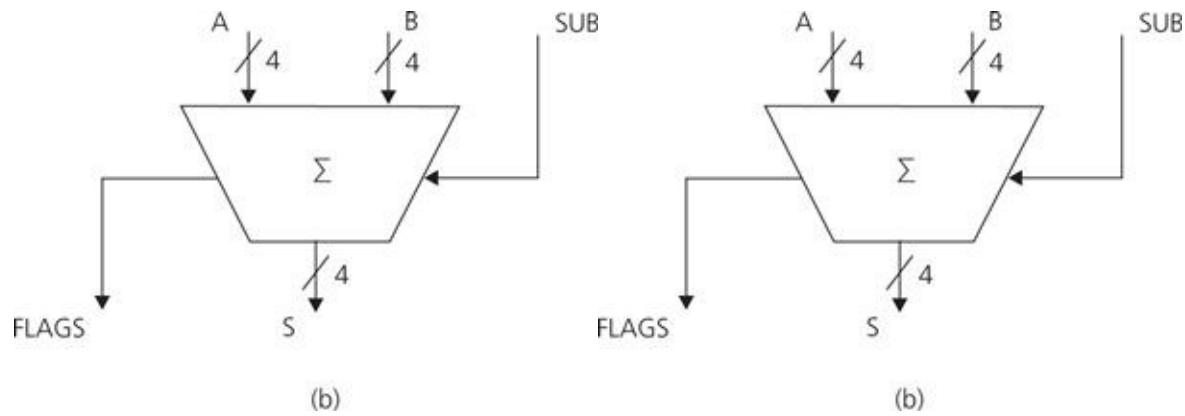
- Inverter todos os bits.
- Somar 1 ao resultado.

A inversão de bits é possível agregando-se uma função NOT aplicada bit a bit. Entretanto, melhor seria se este NOT fosse controlado, pois, se fosse uma SUBTRAÇÃO, os bits B(i) seriam invertidos, e se fosse uma SOMA, os bits B(i) permaneceriam inalterados. A tabela-verdade ao lado conceitua esta operação.

SUB	B(i)	B(i)*
0	0	0
0	1	1
1	0	1
1	1	0

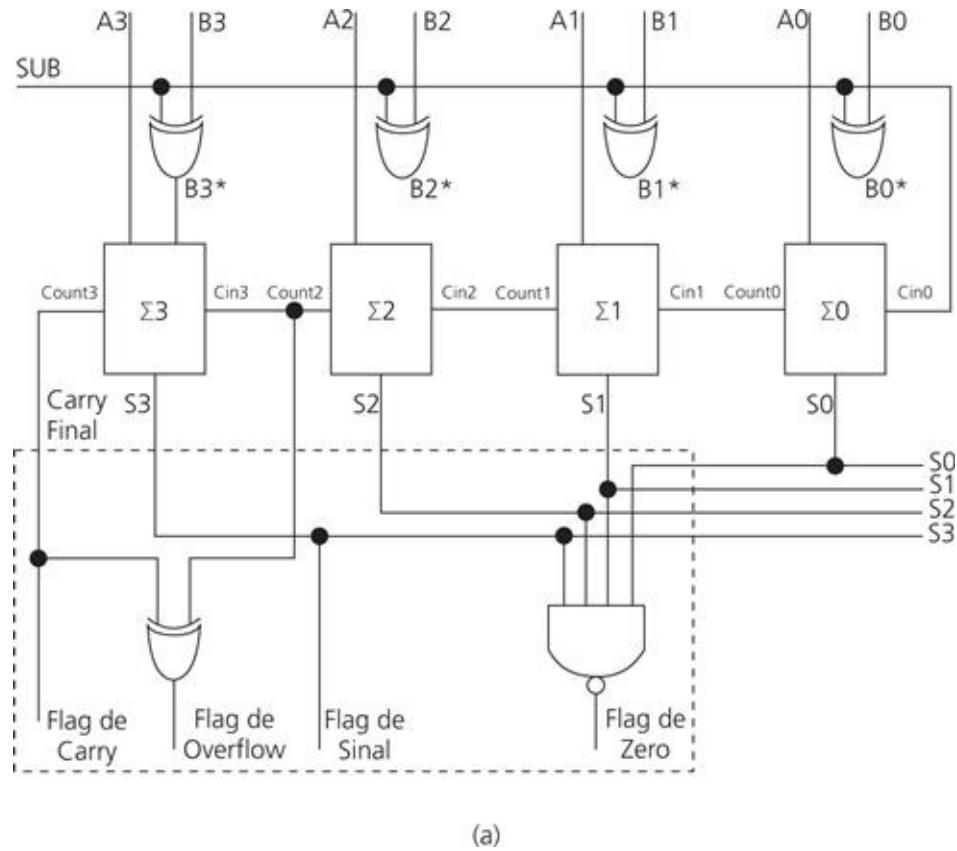
Nota-se, portanto, que  $B(i)_* = B(i)$  somente se  $SUB = 1$ . Essa tabela-verdade indica claramente que uma função OU-exclusivo pode operar como um NOT controlado, conforme for necessário.

A [Figura 8.6.a](#) traz o circuito somador-subtrator. Note que, se o sinal de controle SUB for acionado ( $SUB = 1$ ), as portas XOR fazem a inversão do operando B para o complemento de 1. Observe ainda que a operação de “somar 1” necessária para o complemento de 2 é realizada considerando-se  $Cin_0 = 1$ , que fica conectado ao sinal de controle SUB. Portanto, este circuito executa duas operações sob controle da entrada SUB:



**FIGURA 8.6** Somador-subtrator binário de 4 bits: (a) circuito lógico e (b) símbolo lógico.

SUB	Operação	Cin0
0	Soma	0
1	Subtração com operando B em complemento de 2	1



Pela tabela anterior, percebe-se que Cin0 deve ser ligado à variável de controle SUB para ser setado quando SUB é acionado (SUB = 1, operação de subtração), ou permanecer neutro quando SUB não é acionado (SUB = 0, operação de soma).

## Sinalizadores Ou Flags

Observe o circuito contido no retângulo tracejado na [Figura 8.6.a](#) Ele apresenta algumas saídas especiais denominadas *sinalizadores* (em inglês, *flags*), utilizadas para indicar algumas propriedades do resultado S do circuito somador/subtrator.

O *flag de sinal* sinaliza que a operação resultou em um número negativo. Ele nada mais é do que o bit MSB do número binário S, pois, no contexto de números sinalizados, o sinal de um número encontra-se representado no MSB (reveja este conceito no [Capítulo 2](#)).

O *flag de carry* corresponde ao *carry final* da soma, e serve para sinalizar a ocorrência de transbordamento além da posição MSB. Lembre-se que nem sempre isso é um problema.

O *flag de zero* detecta, a partir de uma função NAND, que os bits S são todos simultaneamente 0, sinalizando que o resultado da operação é nulo.

O *flag de overflow* serve para sinalizar que o resultado da operação não é válido (ou seja, a operação de soma ou subtração gerou um número não representável nos N bits da saída S). Ele é obtido a partir da comparação dos Cout do último e penúltimo módulos  $\Sigma$  de acordo com a seguinte regra:

Cout(MSB)	Cout(MSB-1)	Flag de overflow	Descrição
0	0	0	Não ocorre overflow
0	1	1	Ocorre overflow
1	0	1	Ocorre overflow
1	1	0	Não ocorre overflow

Na tabela anterior, percebe-se que a detecção de overflow pode ser facilmente realizada por uma função XOR.

A [Tabela 8.2](#) resume o que esses flags indicam. Em arquiteturas de computadores digitais, tais flags são denominados flags de estado.

**Tabela 8.2**  
**Significado dos sinalizadores de estado (flags)**

Flag	Valor lógico	Significado
Sinal	SF = 0	Resultado S positivo
	SF = 1	Resultado S negativo
Carry	CF = 0	Não ocorreu transbordamento
	CF = 1	Ocorreu transbordamento
Zero	ZF = 0	Resultado S não nulo
	ZF = 1	Resultado S nulo
Overflow	OF = 0	Resultado válido
	OF = 1	Resultado inválido

O desenho do circuito lógico interno do somador/subtrator é muito complexo, por isso utilizamos um símbolo (veja a [Figura 8.6.b](#)) para representá-lo em diagramas de bloco de sistemas digitais. Nos próximos capítulos, vamos ter a oportunidade de utilizá-lo.



## O Que Vem

Nos capítulos que se seguem iremos definir os circuitos lógicos sequenciais, que se baseiam em uma porta lógica muito importante: o flip-flop, ou elemento armazenador. Aplicações inusitadas serão descritas em detalhes, tais como os armazenadores de diversos bits, denominados registradores, e os circuitos contadores binários. Vamos lá!

---

## CAÍTULO 9

# Elementos armazenadores

---

### Objetivos do capítulo

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Reconhecer a estrutura interna, a tabela-verdade e o símbolo lógico dos elementos armazenadores (latches e flip-flops).
- Compreender o funcionamento de latches e flip-flops.
- Entender o funcionamento de flip-flops sensíveis à borda.
- Entender a inicialização do estado lógico de flip-flops por meio das entradas assíncronas de PRESET e CLEAR.



### Apresentação

Alguns tipos de circuitos lógicos possuem a característica de manter indefinidamente o valor da variável lógica de saída enquanto certas condições das variáveis lógicas de entrada ou do próprio circuito forem mantidas.



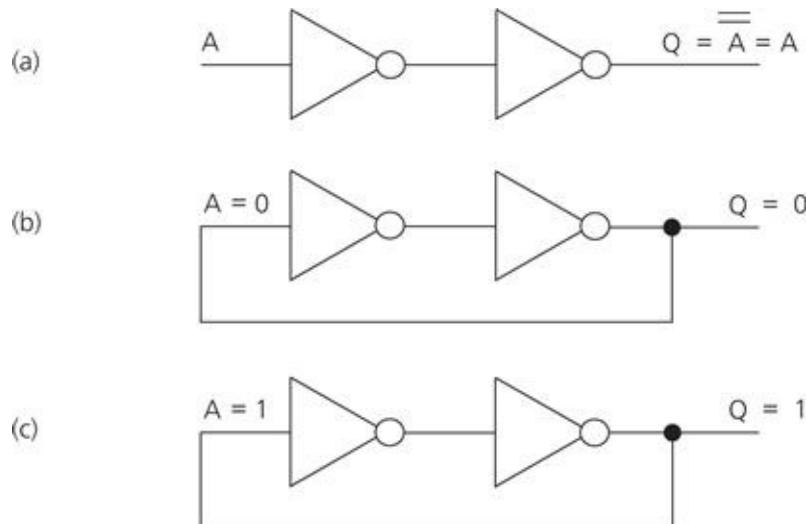
Essa é uma característica totalmente nova em relação aos circuitos combinacionais estudados no [Capítulo 7](#), conhecida como efeito memória.

Iremos agora aprender a estrutura de diversos tipos de elementos armazenadores, cujas aplicações práticas serão posteriormente estudadas no [Capítulo 10](#).

## Fundamentos

### O elemento armazenador básico

Iniciaremos nosso estudo com o exemplo da [Figura 9.1](#). No circuito da [Figura 9.1.a](#) temos a associação em série de duas funções NOT. Pela propriedade da dupla inversão (veja a propriedade 9 do item “Propriedades das funções lógicas básicas” do [Capítulo 5](#)), no exemplo considerado a saída deve ser igual à entrada.



**FIGURA 9.1** Elemento armazenador composto por NOTs.

Mas se a saída  $Q$  for ligada à entrada  $A$ , nota-se que  $Q$  pode adquirir, de forma estável, o valor lógico 0 e também o valor lógico 1.

Na [Figura 9.1.b](#), se  $A$  “nascer” com valor 0,  $Q$  apresentará 0 indefinidamente. Por outro lado, na [Figura 9.1.c](#), se  $A$  “nascer” com valor 1,  $Q$  apresentará 1 indefinidamente.

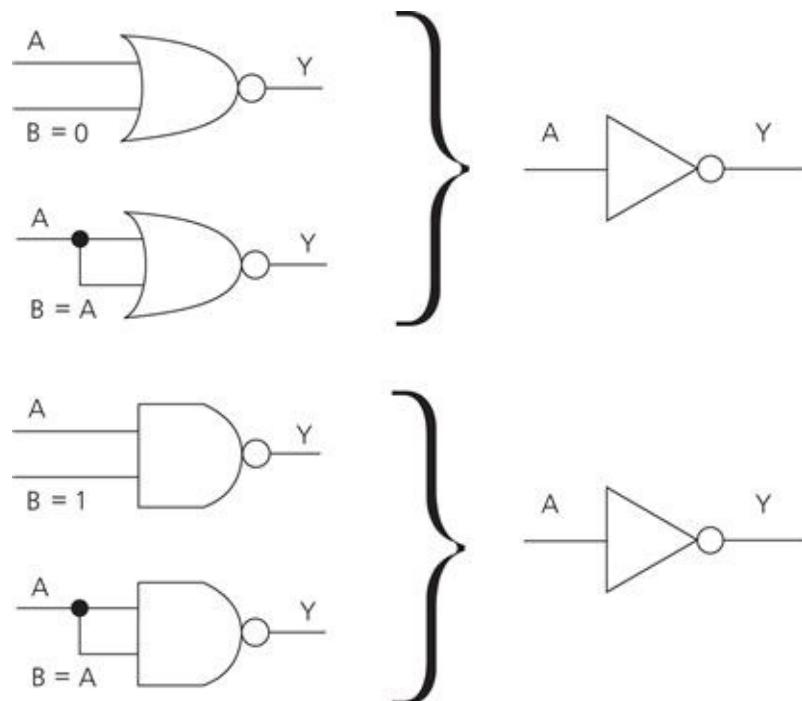
Dizemos que esse circuito armazena um bit 0 ou 1, ou seja, apresenta a característica de armazenamento, pois o valor de sua saída pode ser “lembrado” em qualquer momento futuro. Este circuito é conhecido como elemento armazenador ou célula de memória. Uma característica importante deste circuito é a ligação da saída para a entrada, chamada de realimentação. O caminho de

realimentação é responsável pelo efeito memória e deve sempre apresentar um número par de negações para ser estável.

Esse circuito tem um problema prático: ele deve “nascer” com um certo valor lógico na saída Q, e não permite a alteração desse valor em momento algum. Falta ao menos uma entrada lógica neste circuito que permita definir ou redefinir o valor lógico na sua saída.

Entretanto, seu grande mérito é o armazenamento por tempo indefinido, o que se define como *efeito de memória*.

A [Figura 9.2](#) mostra como funções NOR e NAND podem se comportar como NOT sob certas condições.



**FIGURA 9.2** Funções NOR e NAND operando como NOT.

Ao analisarmos a tabela-verdade de uma função NOR cuja entrada B é considerada fixa em 0 (veja o item “Função NOR” do [Capítulo 4](#)), podemos observar que restam apenas a 2<sup>a</sup> e a 4<sup>a</sup> linhas da tabela, para as quais Y é sempre o oposto de A. Logo, este comportamento equivale ao da função NOT aplicada à variável A apenas (a variável B é subutilizada). O mesmo raciocínio vale para a função NAND quando a entrada B é fixa em 1.

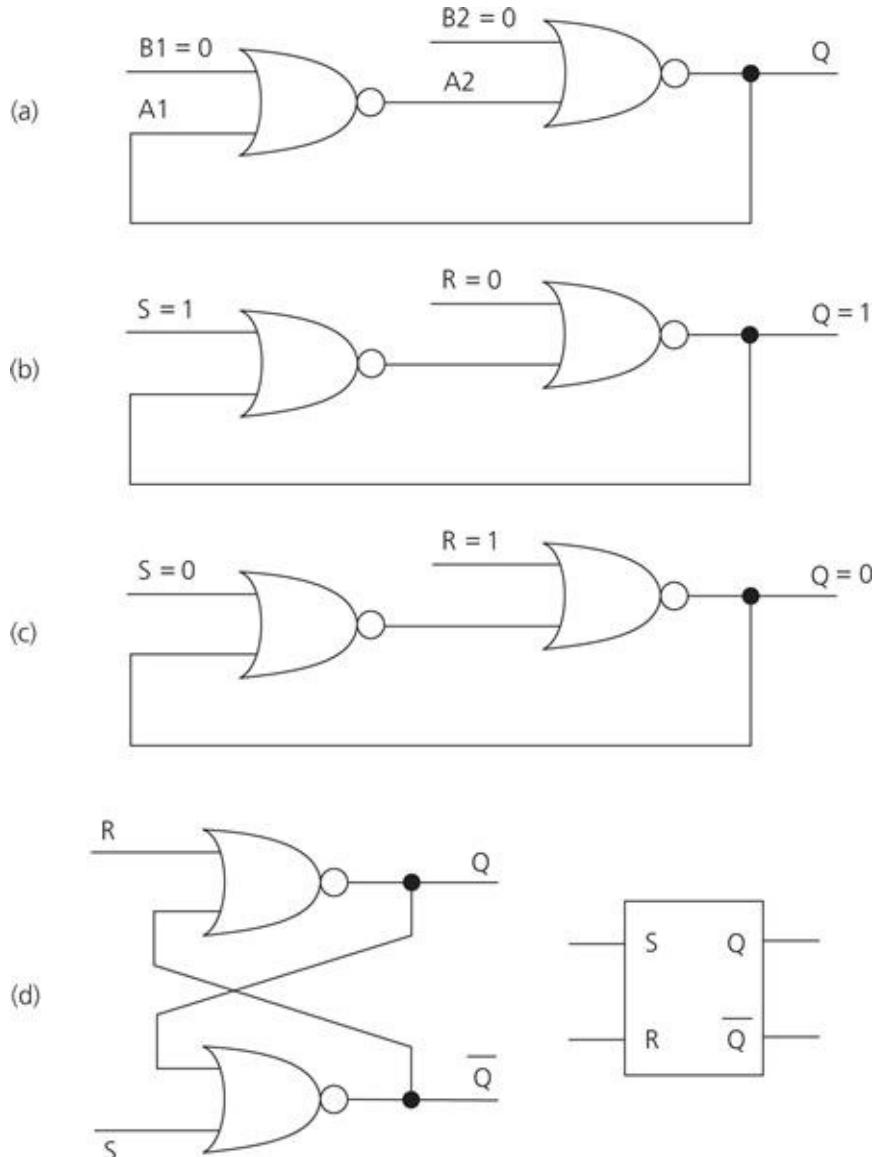
Note também que em ambas tabelas-verdade, Y é o oposto de A se B for

considerada igual a A. Estes resultados podem ser aplicados para substituir os NOTs do circuito da [Figura 9.1](#).

As células de memória com algumas modificações em sua estrutura dão origem aos latches e flip-flops, que são elementos lógicos muito usados em eletrônica digital para o armazenamento de bits. A principal diferença entre latches e flip-flops está no método usado para alterar o bit armazenado.

## Latch SR

A [Figura 9.3](#) ilustra uma célula de memória construída com dois NOR de duas entradas, com a entrada B fixa em 0.



**FIGURA 9.3** Célula de memória (latch SR) construída com NOR.

Podemos renomear as entradas B1 e B2 como S e R, respectivamente. No circuito da [Figura 9.3.b](#), se considerarmos apenas  $S = 1$ , a saída do NOR à esquerda se definirá como 0 (pela tabela-verdade da função NOR, quando uma das entradas for 1 a saída se define como 0 – confira!) e será negada pelo NOR à direita, resultando  $Q = 1$ . Alternativamente, se fizermos apenas  $R = 1$ , Q assumirá valor 0 pelo mesmo motivo explicado anteriormente.

Notamos, portanto, que a célula de memória assim construída agora possui duas entradas que conseguem facilmente definir e alterar o valor da saída Q, obedecendo às seguintes regras:

- Quando as entradas S e R estão simultaneamente desativadas em 0, ambos os NORs funcionam como NOTs, e a saída Q mantém seu valor lógico indefinidamente ([Figura 9.3.a](#)).
- A entrada S é denominada SET, e quando for ativada em 1 faz a saída Q = 1 ([Figura 9.3.b](#)).
- A entrada R é denominada RESET, e quando for ativada em 1 faz Q = 0 ([Figura 9.3.c](#)).
- Não se pode fazer SET e RESET simultaneamente, pois dessa forma cria-se uma ambiguidade no valor a ser armazenado. Portanto, a combinação S = R = 1 é proibida.

A célula de memória NOR, denominada latch SR, pode ser redesenhada da forma mostrada na [Figura 9.3.](#), e possui um símbolo específico mostrado à direita nesta mesma figura. Note que a realimentação da [Figura 9.1](#) aparece no circuito final como uma ligação cruzada das saídas para as entradas das portas NOR. Note também a existência de uma segunda saída, a Q-barra, que é exatamente a negação de Q e existe naturalmente entre os dois NORs do circuito. Embora Q-barra seja irrelevante para o entendimento da funcionalidade da célula de memória, é comum encontrá-la disponível no símbolo lógico.

Em resumo, a funcionalidade do latch SR construído com NOR é descrita na tabela-verdade seguinte.

---

### Tabela 9.1

#### Tabela-verdade do latch SR (construído com NOR)

---

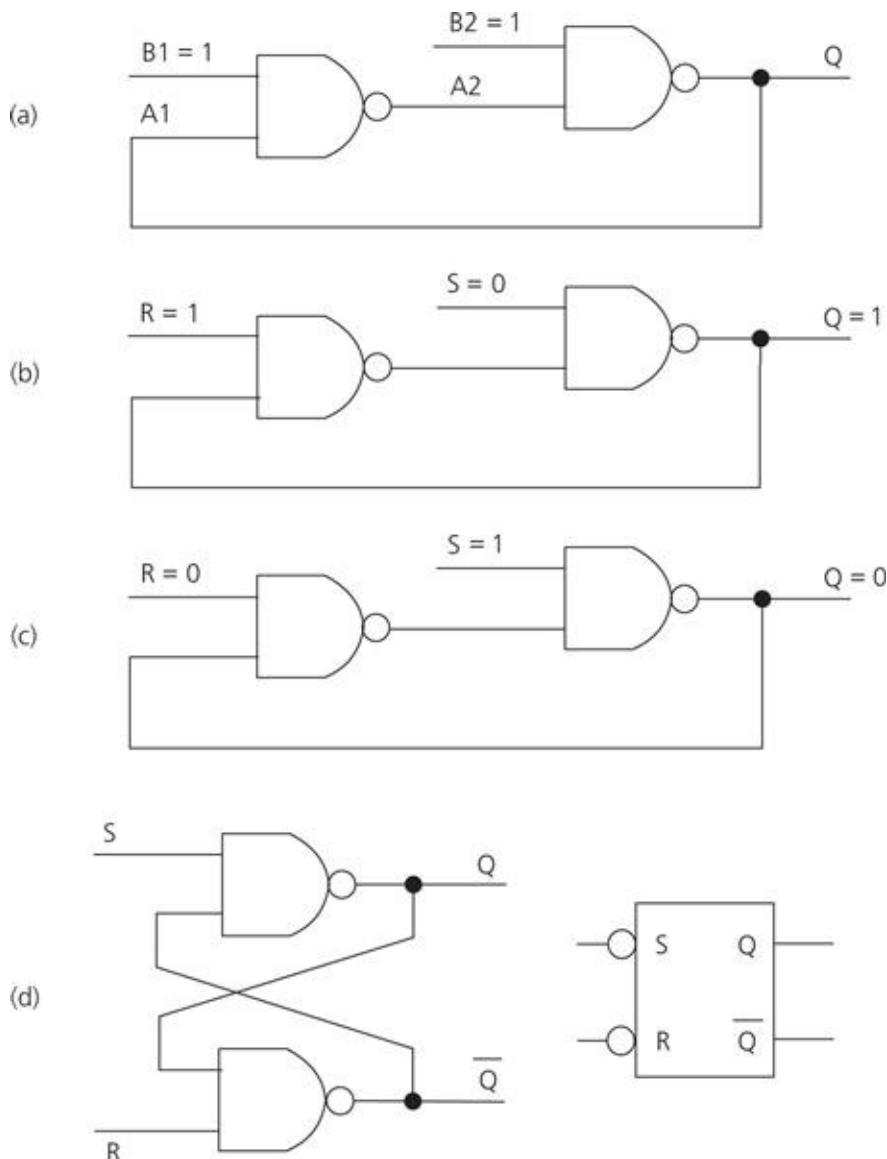
S	R	Q	Q-barra	Funcionalidade
0	0	Mantém	Mantém	Efeito memória
0	1	0	1	RESET
1	0	1	0	SET
1	1	?	?	Combinação proibida

Um latch SR pode também ser construído com portas lógicas NAND, tal como mostra a [Figura 9.4](#). Seu funcionamento obedece às seguintes regras:

- Quando as entradas S e R estão simultaneamente desativadas em 1, ambos os NANDs funcionam como NOTs, e a saída Q mantém seu valor lógico

indefinidamente ([Figura 9.4.a](#)).

- A entrada S é denominada SET, e quando ativada em 0 faz a saída Q = 1 ([Figura 9.4.b](#)).
- A entrada R é denominada RESET, e quando ativada em 0 faz Q = 0 ([Figura 9.4.c](#)).
- Não se pode fazer SET e RESET simultaneamente, pois dessa forma cria-se uma ambiguidade no valor a ser armazenado tal como na versão baseada em NOR. Portanto, a combinação S = R = 0 é proibida.



**FIGURA 9.4** Célula de memória (latch SR) construída com NAND.

Nota-se que os circuitos construídos com NOR e NAND são semelhantes e funcionalmente equivalentes. Na versão NOR, S e R são ativados pelo valor lógico 1, enquanto na versão NAND, S e R são ativados por 0. Esta diferença é realçada no símbolo lógico do latch SR (mostrado na [Figura 9.4.d](#)) com a existência de “bolinhas” nas entradas S e R.

A funcionalidade do latch SR construído com NAND é descrita na tabelaverdade a seguir.

## Tabela 9.2

### Tabela-verdade do latch SR (construído com NAND)

S	R	Q	Q-barra	Funcionalidade
0	0	?	?	Combinação proibida
0	1	1	0	SET
1	0	0	1	RESET
1	1	Mantém	Mantém	Efeito memória

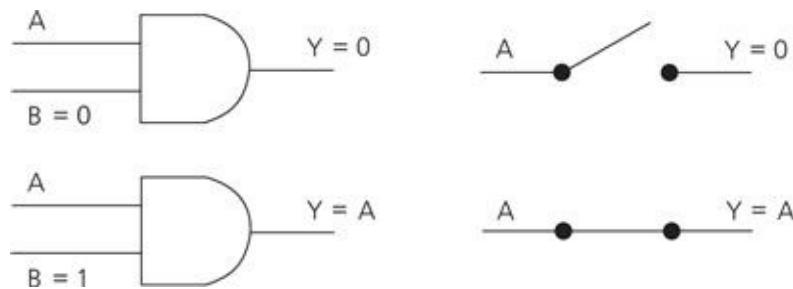
Portanto, as tabelas-verdade de latches SR construídos com NOR e NAND são invertidas. Podemos também dizer que aqui se manifesta o princípio da dualidade descrito no item “O Princípio da Dualidade” do [Capítulo 5](#).

### Latch SR síncrono

Um latch SR aceita alterações no valor armazenado a qualquer momento, bastando ativar uma das entradas (S ou R). Em sistemas digitais, é comum definir momentos específicos nos quais as alterações de valor dos elementos armazenadores são permitidas; neste sentido, define-se um sinal lógico denominado CLK (do inglês *clock*, ou seja, relógio), que alterna seu valor lógico periodicamente entre 1 e 0. O latch SR pode ser modificado para incluir a entrada CLK e permitir a propagação da ação SET ou RESET apenas quando CLK for 1.

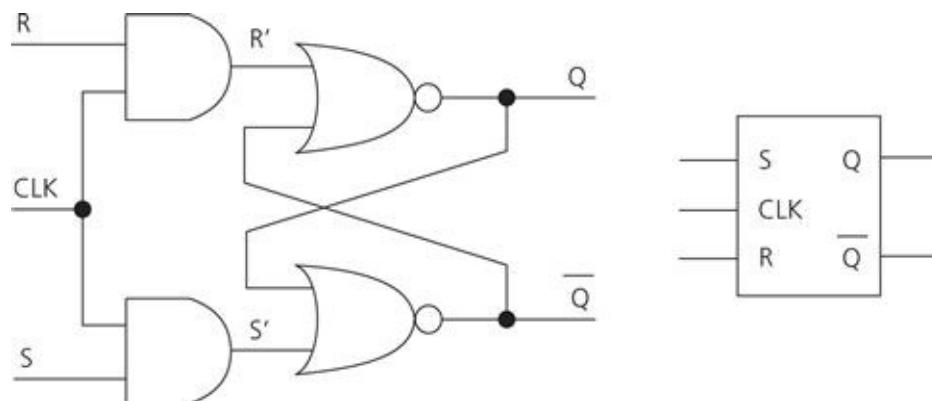
Observe na [Figura 9.5](#) o efeito causado por uma porta AND quando uma das entradas é usada como controle. Quando B = 0, temos a saída Y = 0 qualquer que seja o valor da entrada A. Porém, quando B = 1, Y = A. Logo, uma porta

AND nessas condições opera como porta de passagem/bloqueio, permitindo que a variável A se propague até Y somente quando a outra entrada for 1. Caso contrário, o bloqueio ocorre e a saída assume o valor lógico 0.



**FIGURA 9.5** AND como porta de passagem/bloqueio.

Aplicando-se o conceito de porta de passagem/bloqueio ao latch SR obtemos o latch SR síncrono mostrado na [Figura 9.6](#). Quando  $CLK = 1$ , as variáveis internas  $S'$  e  $R'$  ficam definidas pelas entradas  $S$  e  $R$  respectivamente. Quando  $CLK = 0$ ,  $R' = S' = 0$ , o latch mantém inalterado o valor lógico armazenado.



**FIGURA 9.6** Latch SR síncrono.

A tabela-verdade seguinte descreve a funcionalidade do latch SR síncrono baseado em NOR. Observe na 1<sup>a</sup> linha o uso do valor X para significar “qualquer que seja” ou “não importa”: quando  $CLK = 0$ , para qualquer  $S$  e para qualquer  $R$  a saída  $Q$  mantém o valor lógico armazenado. O uso do X ajuda a resumir a funcionalidade descrita em uma tabela-verdade.

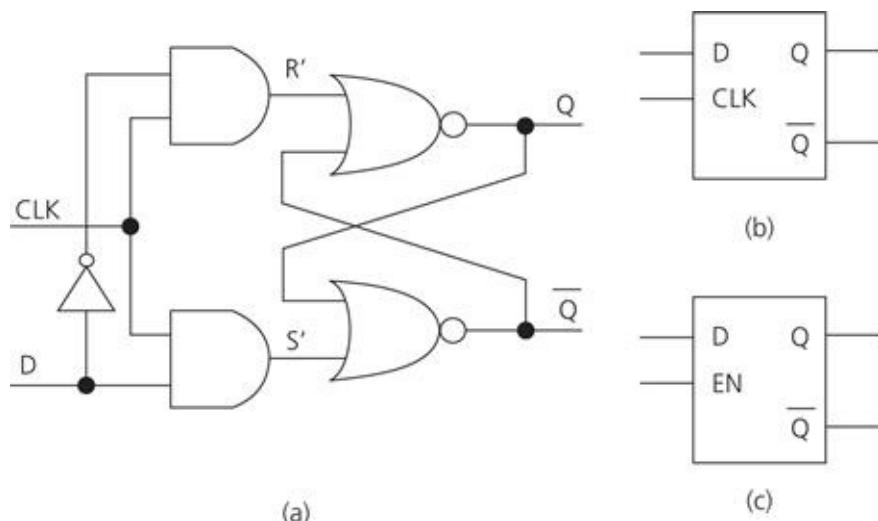
CLK	S	R	Q	Q-barra	Funcionalidade
0	X	X	Mantém	Mantém	Efeito memória
1	0	0	Mantém	Mantém	Efeito memória
1	0	1	0	1	RESET
1	1	0	1	0	SET
1	1	1	?	?	Combinação proibida

De forma semelhante, é possível construir um latch SR síncrono baseado em NAND. Para isso, basta unir o latch SR básico construído com a NAND da Figura 9.4 às duas portas de passagem/bloqueio da Figura 9.5.

### Latch D síncrono

Na tabela-verdade do latch SR síncrono observa-se que a combinação proibida  $S = R = 1$  ainda existe e que a combinação  $S = R = 0$  não é necessária, pois a característica de efeito memória é garantida por  $CLK = 0$ .

O que acontece então se considerarmos S e R sempre diferentes (portanto, nunca iguais)? Isso é facilmente obtido renomeando-se a entrada S como D, e gerando a entrada R como a negação (NOT) de D. Veja o circuito mostrado na Figura 9.7.a, denominado latch D síncrono.



**FIGURA 9.7** Latch D síncrono: (a) circuito lógico interno, (b) símbolo lógico e (c) símbolo lógico da versão latch D com *enable* (termo em inglês que significa

“permissão”).

A tabela-verdade a seguir, bastante simplificada, resume a funcionalidade do latch D síncrono. Note o uso do valor X na 1<sup>a</sup> linha: quando CLK = 0, qualquer que seja D, a saída Q mantém o valor lógico armazenado. A letra D é usada como indicação de que este latch armazena um dado (em inglês, *data latch*).

---

### Tabela 9.3

#### Tabela-verdade do latch D síncrono

---

CLK	D	Q	Q-barra	Funcionalidade
0	X	Mantém	Mantém	Efeito memória
1	0	0	1	Armazena dado = 0
1	1	1	0	Armazena dado = 1

O latch D, cujo símbolo lógico é mostrado na [Figura 9.7.b](#), é muito utilizado na construção de circuitos digitais para processamento de dados devido a sua simplicidade de operação. Ele atua como um elemento armazenador que captura o valor da entrada D durante o intervalo em que a entrada CLK permanece em 1, mantendo o bit armazenado durante todo o tempo em que CLK estiver em 0.

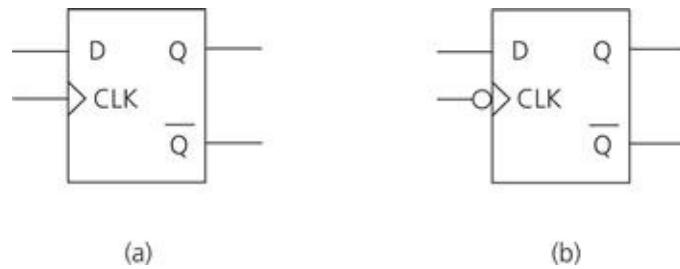
O leitor poderá encontrar o símbolo do latch D ligeiramente modificado, com CLK substituído por EN (do inglês *enable*, termo já visto), porém ele desempenhará a mesma função. O símbolo lógico do latch D com *enable* é mostrado na [Figura 9.7.c](#).

### Flip-flop tipo D sensível à borda

Com a utilização do latch D, a captura do dado não é instantânea, mas se verifica durante todo o intervalo em que CLK esteja em nível 1. Isso parece não ser um grande problema, exceto pelo fato de que o latch D pode responder a uma alteração da entrada D até o último momento antes de CLK mudar para 0; em diversas aplicações de Eletrônica Digital, o que se espera é que a captura do dado seja instantânea, tal qual uma fotografia rápida – um processo normalmente denominado amostragem.

Portanto, uma última melhoria a ser feita no latch D seria transformá-lo em um

flip-flop tipo D (ou DFF), que realiza a captura da entrada D em alguma transição ou borda do sinal de CLK, quer seja na borda de subida ou na borda de descida. Isto requer uma eletrônica especial, não implementável por portas lógicas básicas. A [Figura 9.8](#) mostra o símbolo utilizado para o flip-flop tipo D. Note que a indicação de qual borda ativa a captura está agregada ao símbolo junto à entrada CLK. A sensibilidade à borda é indicada pelo símbolo “>” associado à entrada CLK, dentro do retângulo. A borda de descida é visualmente identificada agregando-se uma “bolinha” ao pino do sinal CLK; a borda de subida não possui esta “bolinha”.



**FIGURA 9.8** Flip-flop tipo D: (a) borda de subida e (b) borda de descida.

O leitor poderá encontrar as seguintes denominações (todas com o mesmo significado) para esse novo tipo de flip-flop:

- Flip-flop tipo D sensível à borda de subida (descida).
- Flip-flop tipo D gatilhado pela transição de subida (descida).
- Flip-flop tipo D disparado pela borda de subida (descida).
- *Rising (falling) edge-triggered D-type flip-flop* (em inglês).

A tabela-verdade desse elemento traz uma novidade: a representação da borda ativa. Normalmente se usa o símbolo do degrau ou de uma fecha para indicar a transição de subida ou de descida da variável CLK. Note que o conceito de borda (subida ou descida) não representa um valor lógico e só se aplica à variável CLK. Veja a seguir a tabela-verdade de um flip-flop tipo D sensível à borda de subida e, na sequência, a tabela-verdade de um flip-flop tipo D sensível à borda de descida.

#### Tabela 9.4

##### Tabela-verdade de um flip-flop tipo D sensível à borda de subida

<b>CLK</b>	<b>D</b>	<b>Q</b>	<b>Q-barra</b>	<b>Funcionalidade</b>
0	X	Mantém	Mantém	Efeito memória
1	X	Mantém	Mantém	Efeito memória
↑	0	0	1	Armazena dado = 0 na borda ↑
↑	1	1	0	Armazena dado = 1 na borda ↑

**Tabela 9.5**

**Tabela-verdade de um flip-flop tipo D sensível à borda de descida**

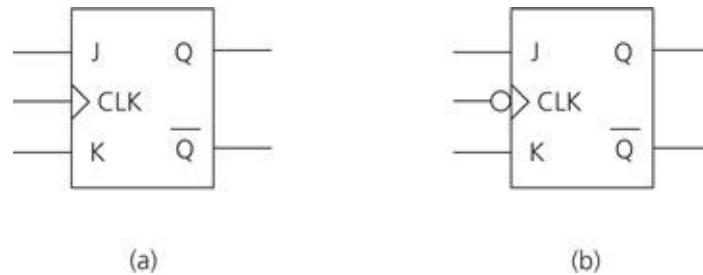
<b>CLK</b>	<b>D</b>	<b>Q</b>	<b>Q-barra</b>	<b>Funcionalidade</b>
0	X	Mantém	Mantém	Efeito memória
1	X	Mantém	Mantém	Efeito memória
↓	0	0	1	Armazena dado = 0 na borda ↓
↓	1	1	0	Armazena dado = 1 na borda ↓

O flip-flop tipo D também é muito empregado na construção de circuitos digitais para processamento de dados, e não apenas pela sua simplicidade de operação, mas principalmente pela sua característica de captura instantânea. A escolha entre um latch D e um flip-flop tipo D é uma opção do projetista de circuitos digitais. Entretanto, as características de algumas aplicações que serão vistas futuramente acabam por exigir uma das opções.

### **Flip-flop tipo JK sensível à borda**

Outra linha de desenvolvimento dos flip-flops, que parte do latch SR síncrono da Figura 9.6, resolve o problema da combinação proibida definindo uma nova função para a mesma. Que tal dar à combinação  $S = R = 1$  a função de trocar o valor lógico armazenado? No contexto lógico, trocar significa negar (trocar pelo outro valor lógico possível) ou comutar. Esta alternativa, agregada ao conceito de sensibilidade à borda de subida ou descida (visando tornar a captura

instantânea), deu origem ao flip-flop tipo JK sensível à borda, cujas duas opções de símbolo são mostradas na [Figura 9.9](#).



**FIGURA 9.9** Flip-flop tipo JK (a) borda de subida e (b) borda de descida.

Assim como flip-flop tipo D, o leitor poderá encontrar as seguintes denominações (todas com o mesmo significado) para ao flip-flop tipo JK sensível à borda:

- Flip-flop tipo JK sensível à borda de subida (descida).
- Flip-flop tipo JK gatilhado pela transição de subida (descida).
- Flip-flop tipo JK disparado pela borda de subida (descida).
- *Rising (falling) edge-triggered JK-type flip-flop* (em inglês).

Veja a seguir as tabelas-verdade para as duas opções de borda:

### Tabela 9.6

#### Tabela-verdade do flip-flop tipo JK sensível à borda de subida

CLK	J	K	Q	Q-barra	Funcionalidade
0	X	X	Mantém	Mantém	Efeito memória
1	X	X	Mantém	Mantém	Efeito memória
↑	0	0	Mantém	Mantém	Efeito memória
↑	0	1	0	1	RESET faz Q = 0 na borda ↑
↑	1	0	1	0	SET faz Q = 1 na borda ↑
↑	1	1	Comuta	Comuta	Comuta o valor de Q na borda ↑

Note que J e K equivalem funcionalmente a S e R respectivamente, exceto pelas ações de J (SET) e K (RESET), que ocorrem apenas na borda de sensibilidade do flip-flop.

**Tabela 9.7**

### Tabela-verdade do flip-flop tipo JK sensível à borda de descida

CLK	J	K	Q	Q-barra	Funcionalidade
0	X	X	Mantém	Mantém	Efeito memória
1	X	X	Mantém	Mantém	Efeito memória
↑	0	0	Mantém	Mantém	Efeito memória
↑	0	1	0	1	RESET faz Q = 0 na borda ↑
↑	1	0	1	0	SET faz Q = 1 na borda ↑
↑	1	1	Comuta	Comuta	Comuta o valor de Q na borda ↑

O flip-flop tipo JK também é muito empregado na construção de circuitos digitais nos quais a característica de comutar o bit armazenado é necessária, como veremos futuramente nos circuitos contadores.

Por fim, perceba que mudamos a designação de latch para flip-flop exatamente ao introduzir o conceito de sensibilidade à borda. Para diferenciar do comportamento por borda, dizemos que os latches são acionados por nível (ou, de outra forma, que os latches são sensíveis ao nível), ou seja, podem mudar o valor lógico armazenado enquanto o nível ativo de CLK for mantido.

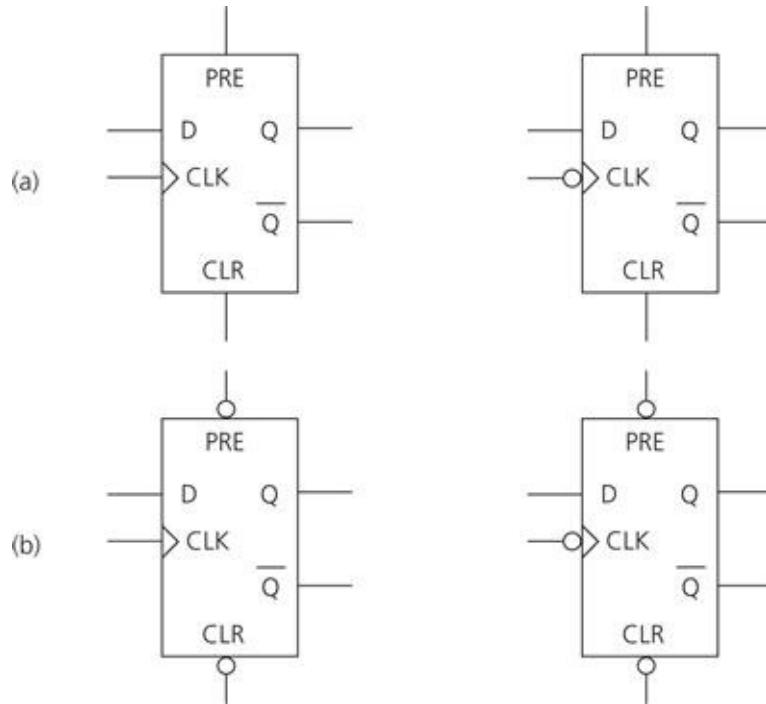
## Entradas Assíncronas

Como vimos, os flip-flops tipo D e JK são controlados fundamentalmente por uma determinada borda na entrada CLK. As demais entradas são ditas síncronas, pois seus valores somente têm efeito sobre a variável de saída Q após a sincronização com o sinal CLK.

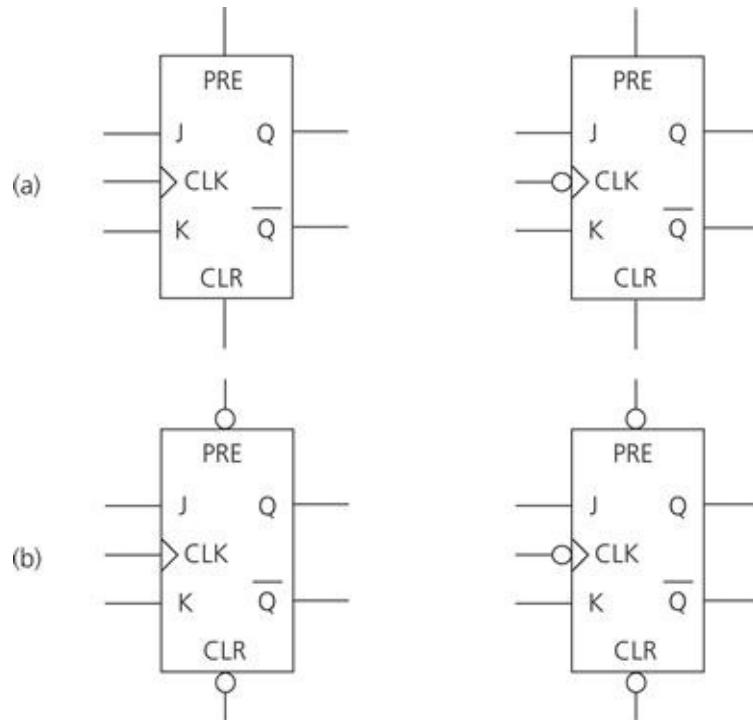
Existem aplicações nas quais se espera certo grau de controle da saída sem o sincronismo de CLK. Nossa objetivo agora é voltar um pouco em direção ao

latch SR básico e criar entradas semelhantes a SET e RESET assíncronas.

As [Figuras 9.10 e 9.11](#) ilustram os símbolos lógicos de flip-flops tipo D e JK que contêm entradas assíncronas de PRESET (PRE = SET prévio) e CLEAR (CLR = RESET prévio, equivalente a limpar ou zerar).



**FIGURA 9.10** Flip-flops tipo D com entradas assíncronas de PRESET e CLEAR (a) ativas em nível alto e (b) ativas em nível baixo.



**FIGURA 9.11** Flip-flops tipo JK com entradas assíncronas de PRESET e CLEAR (a) ativas em nível alto e (b) ativas em nível baixo.

Como podemos ver nos símbolos dessas figuras, existem duas possibilidades:

- Ativar as entradas PRE e CLR aplicando um nível lógico alto ou 1, causando em cada entrada o efeito descrito pelo seu nome.
- Realizar o mesmo procedimento, agora aplicando um nível lógico baixo ou 0.

Se aplicarmos o nível não ativo (0 no caso ‘a’ e 1 no caso ‘b’), os efeitos de PRE e CLR não ocorrem e os flip-flops passam a ser governados pelas demais entradas, conforme explicado nos itens anteriores.

Vejamos alguns exemplos:

1. Considere um flip-flop tipo D sensível à borda de descida que contém entradas assíncronas PRE e CLR ativas em nível alto (para encurtar, dizemos também PRE e CLR ativo-alto). Esta funcionalidade corresponde ao símbolo lógico mostrado na [Figura 9.10.a](#) à direita, cuja tabela-verdade é descrita a seguir.

<b>PRE</b>	<b>CLR</b>	<b>CLK</b>	<b>D</b>	<b>Q</b>	<b>Q-barra</b>	<b>Funcionalidade</b>
1	0	X	X	1	0	PRESET faz $Q = 1$
0	1	X	X	0	1	CLEAR faz $Q = 0$
0	0	0	X	Mantém	Mantém	Efeito memória
0	0	1	X	Mantém	Mantém	Efeito memória
0	0	↓	0	0	1	Armazena $D = 0$ na borda ↓
0	0	↓	1	1	0	Armazena $D = 1$ na borda ↓

Note que na tabela-verdade as entradas PRE e CLR têm prioridade sobre as demais entradas, ou seja, quando elas são ativadas, não importa o valor lógico das demais (veja o “X” nas colunas CLK e D). Não é permitida a ativação das entradas de PRE e CLR simultaneamente, pois esta situação causa ambiguidade na definição das saídas.

2. Considere agora um flip-flop tipo JK sensível à borda de subida que contém entradas assíncronas PRE e CLR ativas em nível baixo (dizemos também PRE e CLR ativo-baixo). O símbolo lógico deste flip-flop é mostrado na [Figura 9.11.b](#) à esquerda, e sua tabela-verdade é descrita a seguir.

<b>PRE</b>	<b>CLR</b>	<b>CLK</b>	<b>J</b>	<b>K</b>	<b>Q</b>	<b>Q-barra</b>	<b>Funcionalidade</b>
0	1	X	X	X	1	0	PRESET faz $Q = 1$
1	0	X	X	X	0	1	CLEAR faz $Q = 0$
1	1	0	X	X	Mantém	Mantém	Efeito memória
1	1	1	X	X	Mantém	Mantém	Efeito memória
1	1	↑	0	0	Mantém	Mantém	Efeito memória
1	1	↑	0	1	0	1	Faz $Q = 0$ na borda ↑
1	1	↑	1	0	1	0	Faz $Q = 1$ na borda ↑
1	1	↑	1	1	Comuta	Comuta	Comuta $Q$ na borda ↑

As entradas PRE e CLR também têm prioridade sobre as demais entradas. Como a ativação das entradas de PRE e CLR se dá pelo nível lógico 0, ambas estarão desativadas se forem deixadas em nível lógico 1. A ativação simultânea de PRE e CLR não é permitida por causar ambiguidade na definição das saídas.

Note finalmente que as entradas assíncronas PRE e CLR não devem ser acionadas a todo momento. Elas servem principalmente para que possamos obter a inicialização de um circuito lógico sequencial (composto também por flip-flops) no qual os valores iniciais armazenados são definidos para que o circuito inicie a partir de um ponto bem determinado. Também servem para que se possa restabelecer essa condição inicial caso o circuito lógico pare de funcionar devidamente.

Circuitos digitais com centenas ou milhares de flip-flops podem conter uma enorme quantidade de combinações de valores armazenados ou estados (bilhões, trilhões), e algumas delas podem causar o travamento do circuito em uma combinação estática, da qual só se pode sair por meio da reinicialização do circuito.



## O que vem depois

Uma vez apresentadas as diversas opções de elementos armazenadores na forma de latches e flip-flops, precisamos aprender a utilizá-los eficientemente. Existem diversas aplicações interessantes que usufruem do efeito memória (armazenamento) presente nos flip-flops. O armazenamento de bits em formato paralelo ou serial e a evolução desse conteúdo na forma de uma contagem em código binário são exemplos destas aplicações. Vamos conhecê-las no próximo capítulo.

---

## CAPÍTULO 10

# Circuitos lógicos sequenciais

---

### Objetivos do capítulo

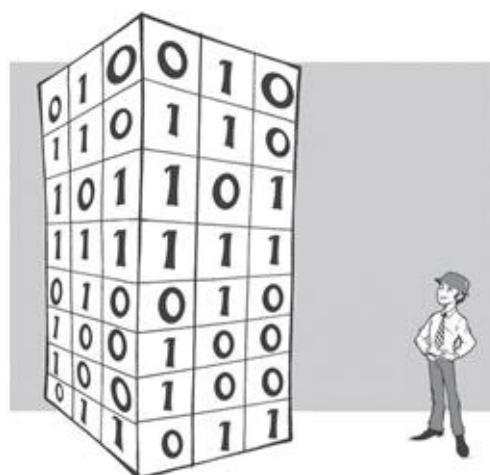
Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Compreender e reconhecer circuitos lógicos sequenciais.
- Compreender o princípio de operação e saber projetar as diversas estruturas de circuitos registradores e contadores.
- Entender o conceito de máquinas de estados finitos e sua aplicação em circuitos controladores.



### Apresentação

Circuitos contendo flip-flops não são combinacionais, mas apresentam um comportamento que depende da sequência de estímulos recebida até um certo momento, ou seja, dependem da história passada, principalmente devido ao efeito memória. Portanto, circuitos contendo flip-flops são denominados circuitos lógicos sequenciais.



Veremos também neste capítulo como projetar circuitos armazenadores de dados (ou registradores) e circuitos contadores. Ao final, estenderemos o conceito de contador para algo mais genérico, que nos auxilie a construir um circuito controlador.

## Fundamentos

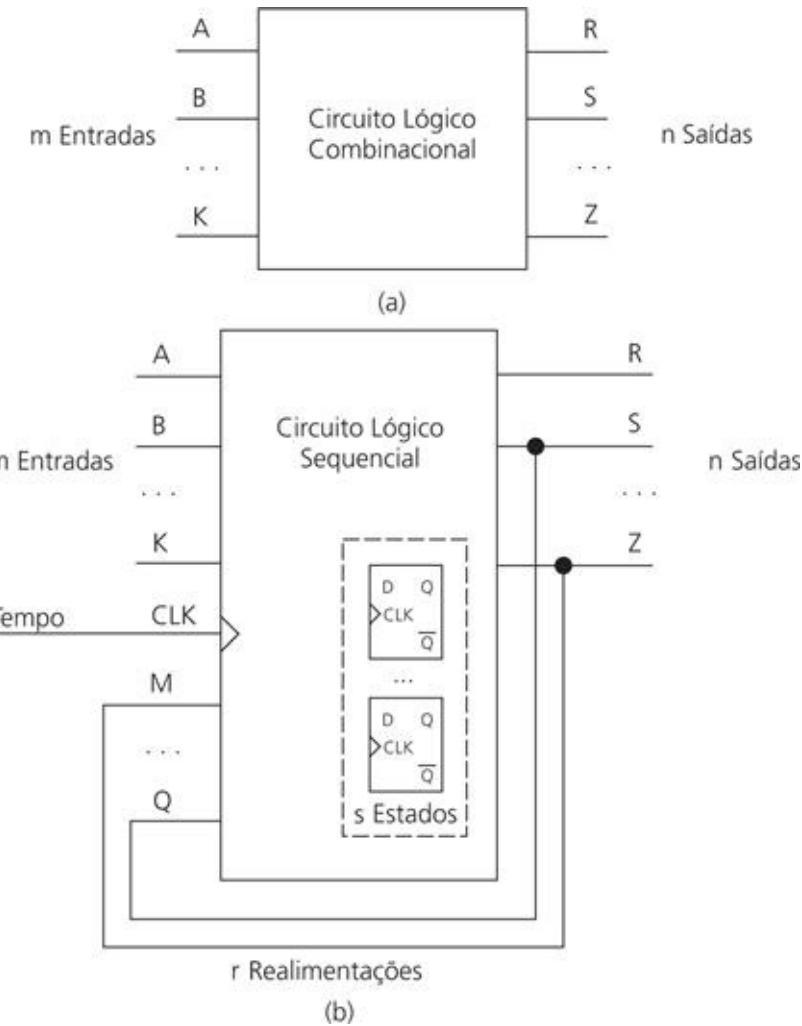
### ***Representação generalizada de um circuito lógico***

Vimos no [Capítulo 7](#) que os circuitos lógicos combinacionais são aqueles em que as variáveis de saída dependem apenas de combinações lógicas das variáveis de entrada, e que eles contêm apenas portas lógicas combinacionais.

Os circuitos lógicos sequenciais são aqueles em que as variáveis de saída podem depender de uma ou mais das seguintes variáveis:

- Variáveis de entrada.
- Variáveis de estado.
- Variáveis de saída realimentadas como entradas.
- Tempo representado pela variável lógica CLK (*clock*).

A [Figura 10.1](#) mostra os diagramas de bloco que conceituam os circuitos combinacionais e sequenciais, para fins de comparação.



**FIGURA 10.1** Diagramas de bloco de (a) circuitos combinacionais e (b) circuitos sequenciais.

Define-se “estado” como o valor lógico do bit armazenado em um flip-flop. Portanto, variáveis de estado são coleções de bits armazenados nos flip-flops de um dado circuito lógico sequencial.

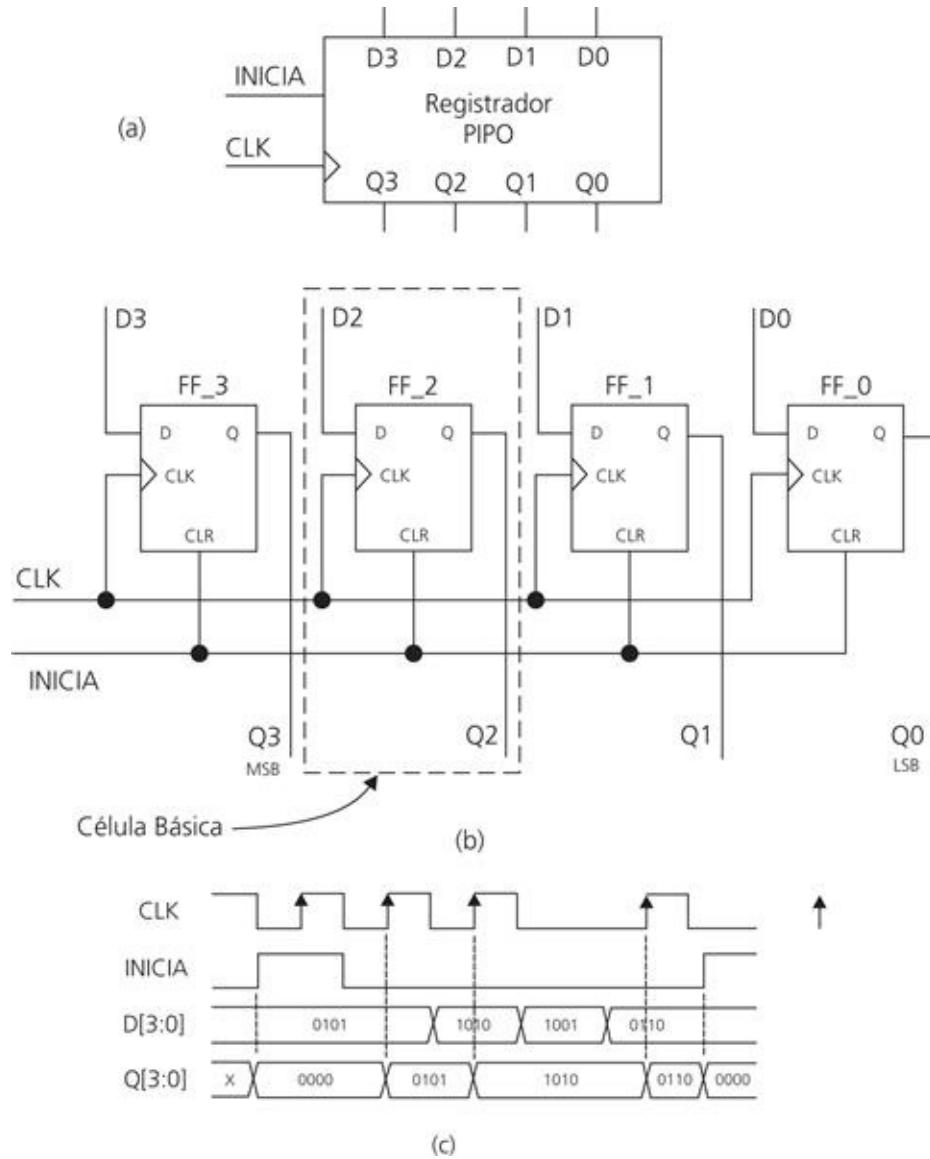
A seguir, iremos analisar diversos circuitos sequenciais compostos fundamentalmente por flip-flops.

## Registradores

A função básica de um flip-flop é armazenar um único bit. Portanto, um conjunto de N bits é armazenado em N flip-flops conectados segundo configurações específicas para que alguma propriedade seja evidenciada.

A Figura 10.2 mostra a configuração conhecida como registrador de entradas e

saídas paralelas ou PIPO (*Parallel In/Parallel Out*).



**FIGURA 10.2** Registrador de 4 bits com entradas e saídas paralelas (PIPO): (a) símbolo lógico, (b) circuito lógico interno e (c) formas de onda.

O circuito lógico da Figura 10.2.b tem como característica fundamental a disponibilidade das entradas e das saídas em formato paralelo, e isso é justamente o que o define como um registrador PIPO. Além disso, observamos algumas outras características:

- a) Um registrador paralelo deve armazenar todos os bits  $D(i)$  instantaneamente. Isto requer o uso de flip-flops tipo D sensíveis à borda. A escolha do tipo de

borda (subida ou descida) é opcional: na falta de algum motivo ou requisito de projeto, é praxe escolher a borda de subida.

- b) Um registrador paralelo deve também armazenar todos os bits D(i) simultaneamente. Isto requer que todas as entradas CLK dos flip-flops tipo D sejam ligadas ao sinal CLK externo. Quando CLK for ativado, todos os bits serão armazenados exatamente no mesmo instante.
- c) É uma boa prática de projeto dotar o circuito de uma forma de inicialização, que defina um valor inicial para todas as saídas Q(i). Por esta razão, foram escolhidos os flip-flops com entradas CLR assíncronas, todas ligadas ao sinal externo INICIA. Ao se acionar INICIA em nível alto, todos os CLR são acionados resultando no valor inicial {D3, D2, D1, D0} = 0000. Inicializações com valores diferentes de “0000” são possíveis com o uso das entradas PRE nas posições em que se deseja um valor 1, que devem também ser conectadas ao sinal externo INICIA.
- d) Note que nos flip-flops não estão desenhadas as saídas Q-barra, pois num registrador não faz sentido negar o valor armazenado. Registradores capturam os bits de entrada e os mantêm armazenados sem alterações.
- e) O armazenamento de bits é feito individualmente, utilizando-se um flip-flop exclusivo para cada bit. Portanto, nunca pense em poder armazenar diversos bits em um mesmo flip-flop ao mesmo tempo.
- f) Costuma-se numerar as entradas, os estágios e saídas de um registrador, iniciando-se com o índice 0, para estabelecer uma ordem crescente relativa à posição de cada bit no conjunto e, sempre que possível, desenhar o estágio 0 mais à direita. Os bits de índice 0 são chamados de LSB (do inglês *Least Significant Bit*, “bit menos significativo”). Os bits de maior índice são chamados de MSB (do inglês *Most Significant Bit*, ou bit mais significativo). A razão desta ordenação tem relação com a notação posicional de um número binário, conforme visto no [Capítulo 2](#).

Observe na [Figura 10.2.b](#) a célula básica de um registrador PIPO. Utilizando esta célula, podemos construir registradores com qualquer quantidade de bits, bastando replicá-la, adicionando-se ou removendo-se estágios a partir da posição MSB.

Para a compreensão da função desempenhada por qualquer circuito sequencial, é necessário recorrer a um diagrama de formas de onda, tal como mostrado na [Figura 10.2.c](#).

Primeiramente, observe que tanto as entradas Di como as saídas Qi foram agrupadas em uma forma de onda única, D[3:0] (lê-se D de 3 a 0) e Q[3:0], para

facilitar a visualização. O valor dos bits de cada variável é indicado por uma sequência de bits que se relaciona à ordem especificada no grupo. Por exemplo,  $D[3:0] = 0101$  corresponde a  $Q3 = 0$ ,  $Q2 = 1$ ,  $Q1 = 0$  e  $Q0 = 1$  respectivamente. Esta é uma notação muito utilizada na representação de variáveis ordenadas ou indexadas.

Observe que o sinal INICIA é ativado no início de funcionamento do circuito, fazendo com que as entradas CLR dos flip-flops sejam ativadas, resultando em  $Q[3:0] = 0000$ .

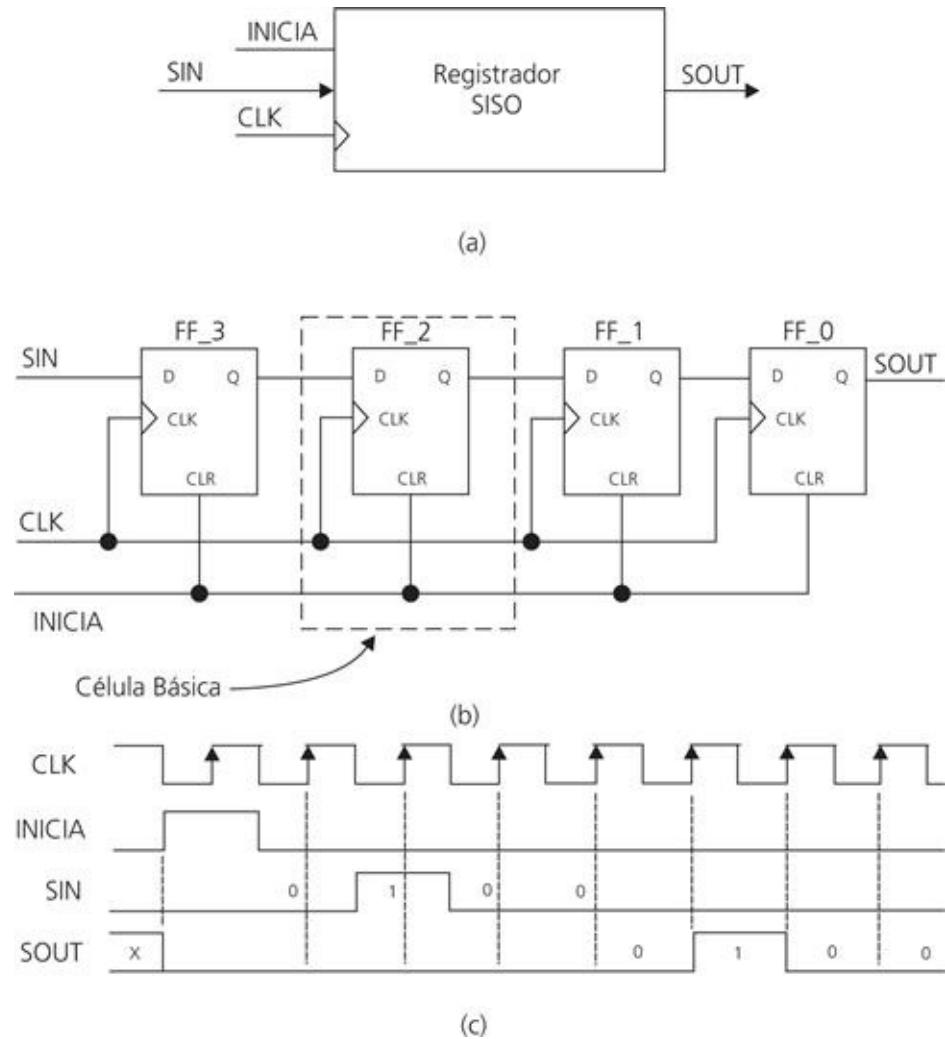
A partir do momento em que INICIA é desativado, o circuito passa a depender exclusivamente da entrada CLK. Note que as entradas D são capturadas nas bordas de subida de CLK, e as saídas Q apenas se atualizam após a borda de subida.

Note que na [Figura 10.2.c](#) há um momento em que as entradas D mudam de valor; as saídas Q não o capturaram pois justamente nesse momento foi suprimida uma borda de subida de CLK. O valor anterior de Q (1010) é mantido a despeito da mudança das entradas caso não haja a “ordem para captura” definida pela borda de subida de CLK.

Finalmente, perceba que as poucas combinações exercitadas pelo diagrama de formas de onda da [Figura 10.2.c](#) permitem confirmar a funcionalidade esperada em um registrador PIPO. Não é necessário ser exaustivo e verificar o funcionamento do registrador para todas as combinações de entradas.

## Registradores De Deslocamento

A [Figura 10.3](#) ilustra um registrador de 4 bits com uma configuração de ligações internas diferente, na qual se pode obter um efeito de deslocamento dos bits armazenados a cada borda ativa de CLK. Neste circuito, temos apenas uma entrada à esquerda e uma saída à direita da estrutura de flipflops encadeados. Esta configuração é denominada registrador de entrada serial e saída serial ou SISO (*Serial In/Serial Out*).



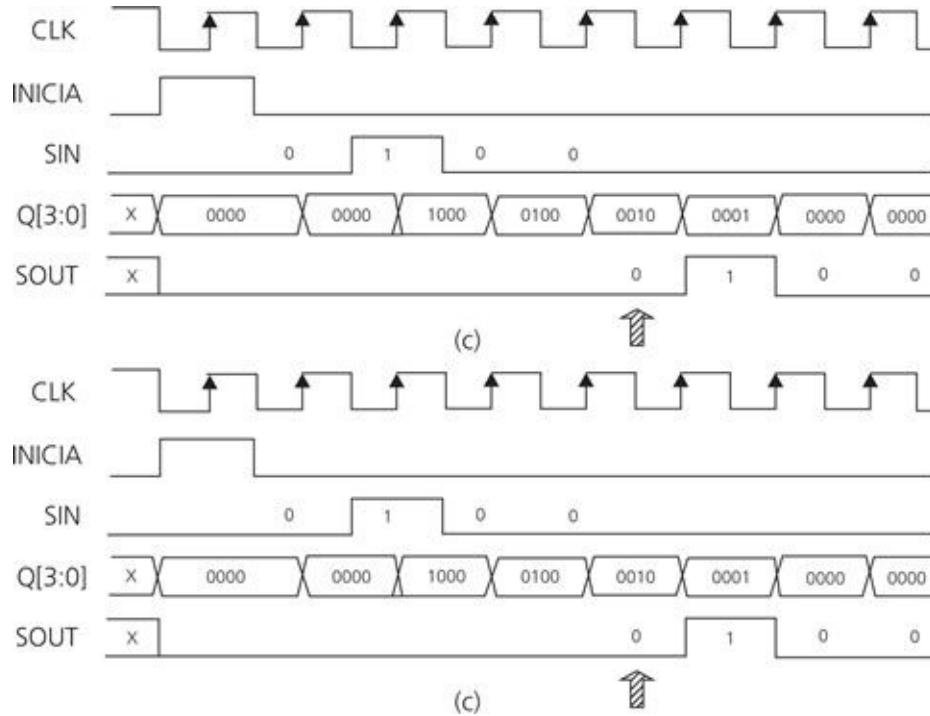
**FIGURA 10.3** Registrador de deslocamento 4 bits com entrada e saída serial (SISO):  
(a) símbolo lógico e (b) circuito lógico interno.

Observe no circuito da [Figura 10.3.b](#) que a saída Q de um flip-flop FF<sub>i</sub> (i) está diretamente ligada à entrada D do flip-flop FF<sub>(i-1)</sub> imediatamente à direita. Nesta forma de encadeamento, a cada borda de subida, as saídas Q(i) são capturadas pelas entradas D(i-1) dos flip-flops FF<sub>(i-1)</sub> e aparecem logo em seguida nas saídas Q(i-1). Este comportamento equivale a um deslocamento de bits da esquerda para a direita.

O diagrama de formas de onda da [Figura 10.3.c](#) ilustra uma demonstração de como o valor serial  $SIN = 0100$  é capturado pelo registrador SISO e aparece na saída SOUT quatro períodos de CLK depois (de uma forma geral, após um número de períodos de CLK igual ao número de estágios do registrador). Note que a entrada SIN é continuamente capturada no flip-flop (3), que passa seu bit

para o flip-flop (2) que o passa para o flip-flop (1) que o passa para o flip-flop (0), cuja saída alimenta a saída SOUT. Este mecanismo de movimentação de bits deu origem ao nome deste circuito: registrador de deslocamento (em inglês, *shift register*).

Uma variação óbvia da estrutura SISO é o registrador de entrada serial e saídas paralelas ou SIPO (*Serial In/Parallel Out*): para obtê-la, basta exteriorizar as variáveis  $Q(i)$  já existentes, tal como mostra a [Figura 10.4](#).



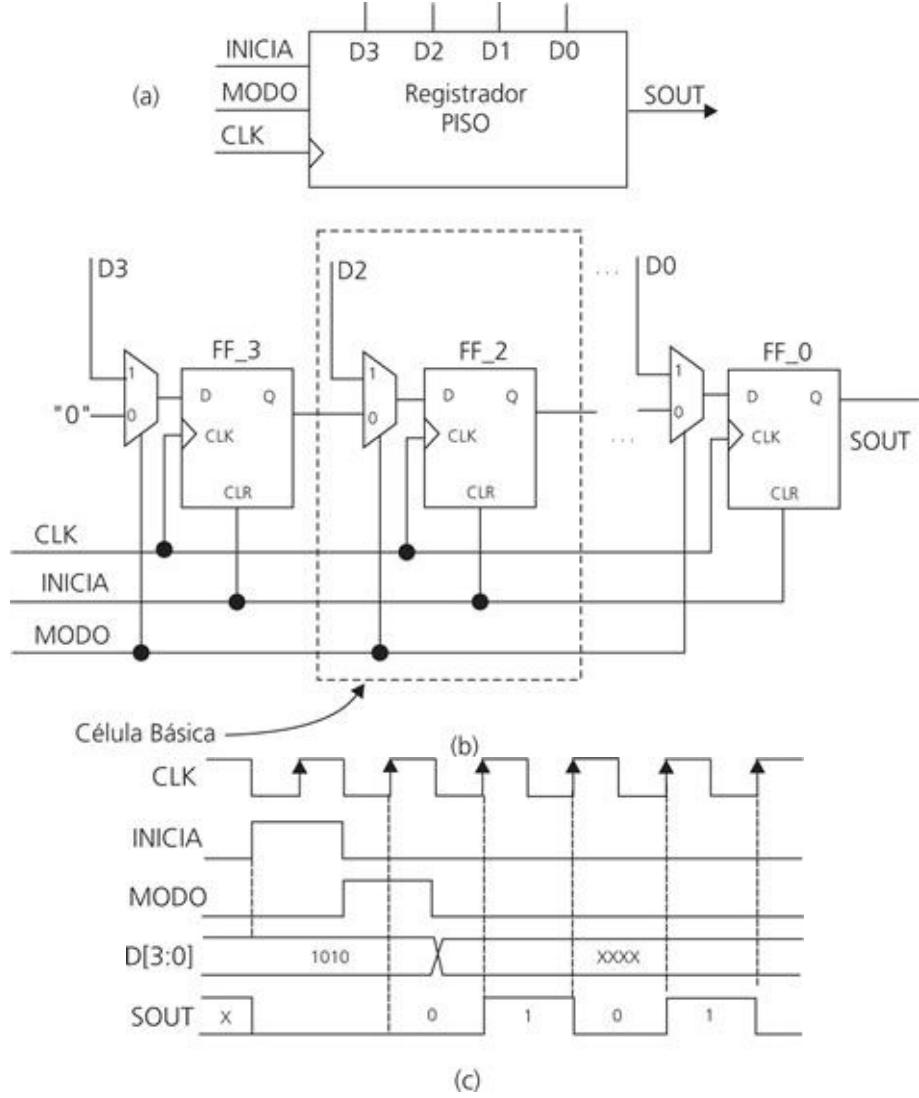
**FIGURA 10.4** Registrador de deslocamento 4 bits com entrada serial e saídas paralelas (SIPO): (a) símbolo lógico, (b) circuito lógico interno e (c) formas de onda.

Observe que o diagrama de formas de onda da [Figura 10.4.c](#) é praticamente idêntico ao da [Figura 10.3.c](#), exceto pelo fato de todas as saídas  $Q$  estarem disponíveis e permitirem a visualização do deslocamento dos bits. Note que após um número de períodos de CLK igual ao número de estágios do registrador SIPO, podemos obter a informação deslocada em formato paralelo nas saídas  $Q[3:0] = 0100$ .

Finalmente, temos a estrutura de um registrador com entradas paralelas e saída serial PISO (*Parallel In/Serial Out*), que também deriva da estrutura SISO da [Figura 10.3](#), agora com a inclusão de um mecanismo para carregamento dos

dados no formato paralelo antes do deslocamento dos bits em direção à saída serial.

Para visualizar melhor, observe a [Figura 10.5](#). Note a existência dos MUX 2:1 conectados às entradas D de todos os flip-flops, cuja função é selecionar uma dentre duas possibilidades: as entradas paralelas D(i) no momento do carregamento dos dados ou as saídas Q(i + 1) no momento do deslocamento dos bits. O carregamento dos dados é obtido pela variável MODO em valor lógico 1, cuja duração deve ser suficiente para haver uma borda de subida de CLK, a qual efetivamente realiza o armazenamento de todos os bits. Em seguida, a variável MODO é trocada para 0 a fim de que sucessivas bordas de subida de CLK causem o deslocamento progressivo dos bits em direção à saída SOUT.



**FIGURA 10.5** Registrador de deslocamento 4 bits com entradas paralelas e saída serial (PISO): (a) símbolo lógico, (b) circuito lógico interno e (c) formas de onda.  
Registrador de deslocamento 4 bits com entradas paralelas e saída serial (PISO): (a) símbolo lógico, (b) circuito lógico interno e (c) formas de onda. (Continuação)

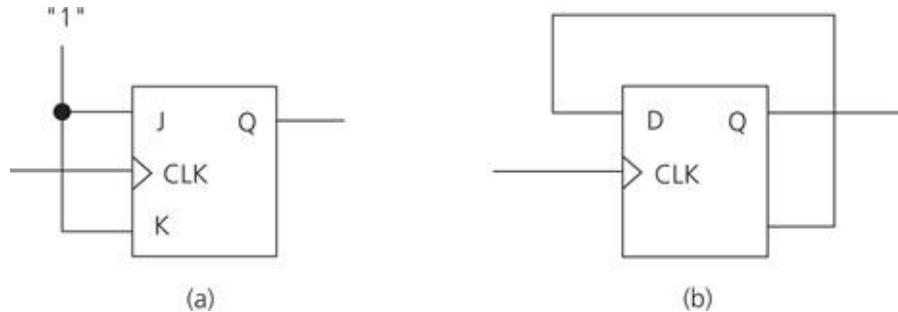
A Figura 10.5.c ilustra as formas de onda que demonstram o funcionamento de um registrador PISO. Note que as entradas D são amostradas em apenas uma borda de subida de CLK, e por isso concluímos que a variável MODO é 1. Logo após, o primeiro bit dos dados capturados, correspondente ao LSB, já está disponível na saída SOUT. Depois demais N-1 períodos de CLK (N é o número de estágios do registrador e o tamanho dos dados capturados nas entradas D), SOUT receberá os demais bits, e o MSB será o último deles.

Note que os circuitos das Figuras 10.3, 10.4 e 10.5 também possuem o sinal de controle INICIA para definir seus valores iniciais (chamados de estado inicial) e são construídos com flip-flops tipo D que utilizam apenas as saídas Q para não alterar a informação armazenada. Além disso, estes circuitos são baseados na borda de subida por mera escolha (poderiam ser baseados na borda de descida, caso alguma aplicação em especial assim exigisse).

Portanto, observamos que parece natural a escolha de flip-flops tipo D em todos os tipos de registradores. O uso de flip-flops tipo JK também é possível mediante algumas adaptações, cujo circuito final pode ser desnecessariamente mais complexo.

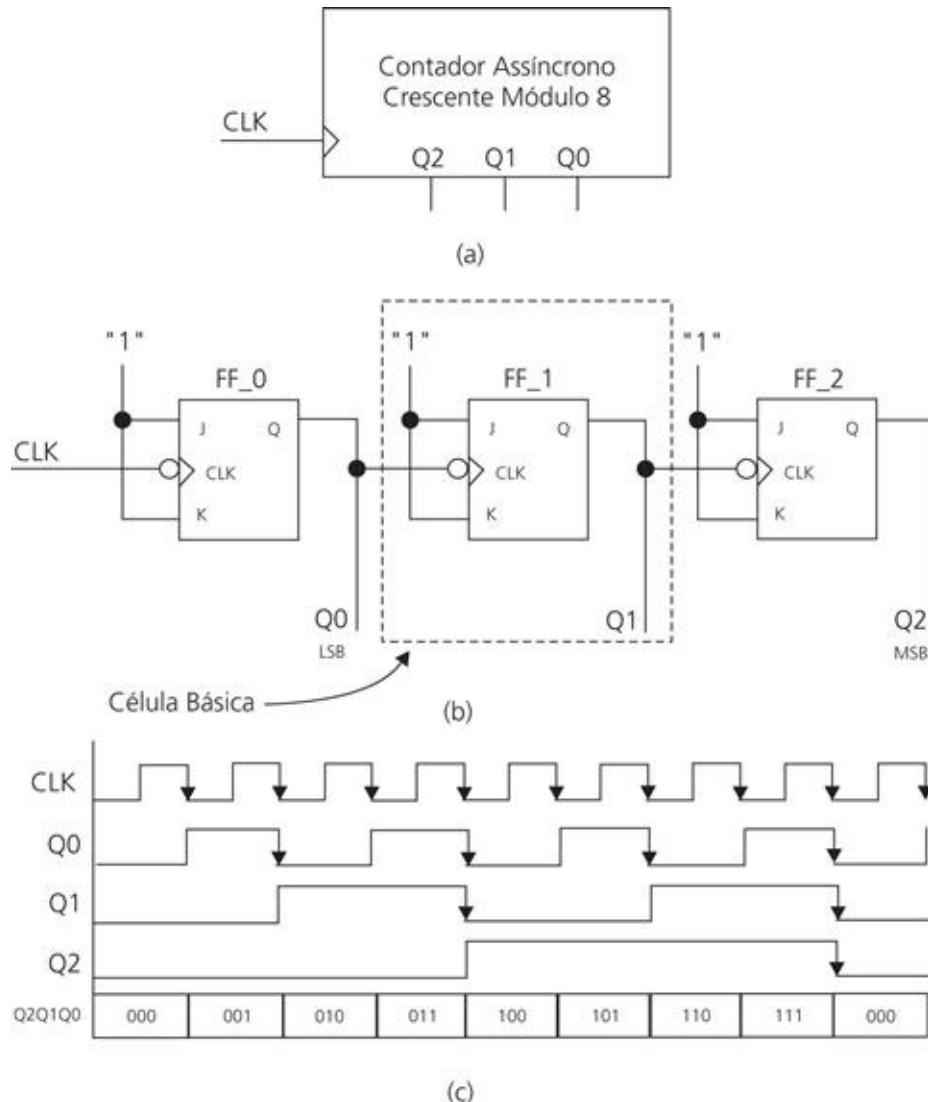
## Contadores assíncronos

A Figura 10.6 mostra duas construções de flip-flops funcionalmente equivalentes, denominadas células de comutação (ou *toggles*). Nestes circuitos, a cada borda ativa de CLK a saída Q comuta de valor lógico. Este comportamento é evidente no flip-flops tipo JK (Figura 10.6.a), uma vez que suas entradas J e K estão fixas em 1, configurando o flip-flops para o modo de comutação. O mesmo comportamento é obtido no flip-flops tipo D (Figura 10.6.b), com a realimentação da saída Q-barra para a entrada D.



**FIGURA 10.6** Células de comutação construídas com (a) flip-flops tipo JK e (b) flip-flops tipo D operando em modo *toggle*.

As células de comutação podem ser encadeadas conforme mostrado na [Figura 10.7.b](#) (ou seja, um exemplo com flip-flop tipo JK sensível à borda de descida). O efeito produzido deve ser analisado com base no diagrama de formas de onda da [Figura 10.7.c](#).



**FIGURA 10.7** Contador assíncrono crescente módulo 8: (a) símbolo lógico, (b) circuito lógico interno e (c) formas de onda.

Esse circuito é dito assíncrono, pois apenas o flip-flop à esquerda está conectado a CLK (ou, de outra forma, sincronizado com CLK). Os demais flip-flops obtêm seu *clock* da saída Q do flip-flop anterior. Observa-se nas formas de onda da Figura 10.7.c que Q0 comuta a cada borda de descida de CLK, assim como Q1 comuta a cada borda de descida de Q0 e Q2 comuta a cada borda de descida de Q1. O efeito obtido é percebido visualmente comparando-se as formas de onda: Q0 tem a metade da frequência de CLK, e Q1 tem a metade da frequência de Q0; portanto, um quarto da frequência de CLK. Continuando a análise, verificamos que Q2 tem um oitavo da frequência de CLK. Consequentemente, células de comutação conectadas como na Figura 10.7.b dão

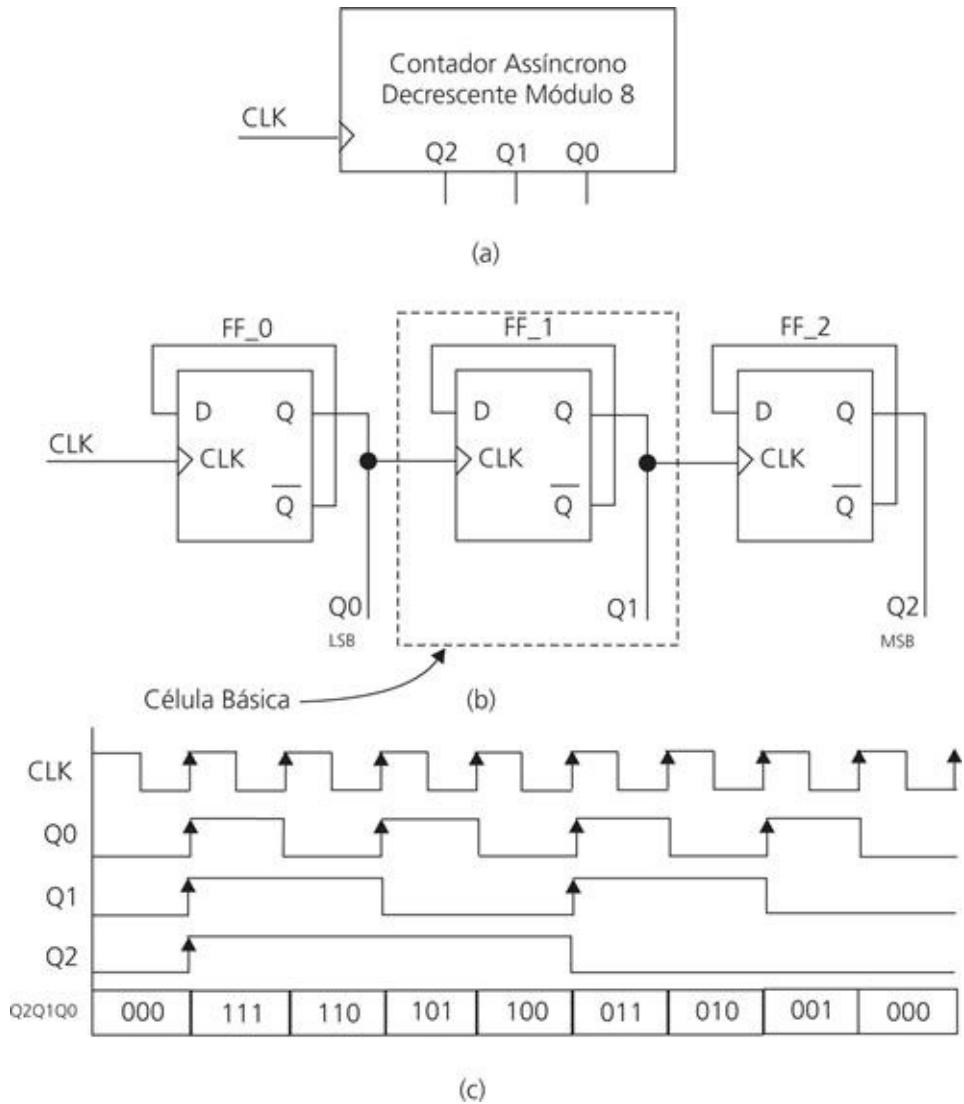
origem a um circuito conhecido como divisor de frequência.

Entretanto, um efeito interessante pode ser percebido se anotarmos o valor lógico das saídas  $Q(i)$ , ordenadas na sequência  $Q_2Q_1Q_0$ , tal como mostrado na última linha das formas de onda da [Figura 10.7.c](#). Observa-se que a sequência de bits evolui de forma semelhante ao código binário limitado a 3 bits, visto na seção “Circuitos combinacionais - segunda parte”. A tabela a seguir resume os resultados.

Decimal	$Q_2Q_1Q_0$
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111
0	000

O circuito divisor de frequência executa também uma contagem binária crescente de 0 a 7, ciclicamente retornando a 0 ao atingir o valor máximo 7, e repetindo indefinidamente a sequência de contagem. Esta configuração é denominada circuito contador assíncrono crescente.

Como segundo exemplo, observe o circuito da [Figura 10.8.b](#), construído com células de comutação baseadas em flip-flops tipo D. Note a sutil mudança na borda ativa dos flip-flops, que agora são todos sensíveis à borda de subida. A análise de funcionamento é semelhante ao exemplo anterior. As formas de onda deste circuito são mostradas na [Figura 10.8.c](#), e neste caso correspondem a um circuito contador assíncrono decrescente.



**FIGURA 10.8** Contador assíncrono decrescente módulo 8: (a) símbolo lógico, (b) circuito lógico interno e (c) formas de onda.

O circuito da [Figura 10.8.b](#) executa um processo de divisão de frequência totalmente equivalente ao circuito da [Figura 10.7.b](#). Além disso, realiza uma contagem binária decrescente de 7 a 0, ciclicamente retornando a 7 ao atingir o valor mínimo 0 e repetindo indefinidamente a sequência de contagem.

Em resumo, os contadores assíncronos podem igualmente ser construídos com células de comutação baseadas em D-FF ou JK-FF. Entretanto, o sentido de contagem (crescente ou decrescente) depende exclusivamente do uso de flip-flops sensíveis à borda de descida para contadores crescentes, ou sensíveis à borda de subida para contadores decrescentes.

A quantidade de estágios pode ser facilmente estendida para alcançar a

quantidade de bits desejada, e para isso basta replicar a célula básica das estruturas.

Os contadores possuem alguns parâmetros que os caracterizam completamente:

- a) O número de estágios  $N$  corresponde ao número de flip-flops utilizados, que equivale ao número de bits usados para obter o número binário utilizado na contagem.
- b) As contagens normalmente possuem valor mínimo zero e valor máximo igual a  $2^{N-1}$ , onde  $N$  é o número de estágios.
- c) Cada combinação lógica das saídas  $Q(i)$  que o contador utiliza durante o processo de contagem é denominada “estado”.
- d) Define-se módulo  $M$  como a quantidade de estados (valores diferentes de contagem) que um contador possui, calculado como  $M = 2^N$ .

Assim, um contador de três estágios é um contador módulo 8, com oito estados, e conta de 0 a 7 se for crescente ou de 7 a 0 se for decrescente.

Nos flip-flops dos circuitos contadores das [Figuras 10.7.b](#) e [10.8.b](#), poderiam ainda ser incluídas as entradas de CLR assíncrono ligadas externamente a um sinal INICIA, assim como foi feito nos registradores, para que se garanta meios de definir a condição inicial dos circuitos em  $\{Q_2, Q_1, Q_0\} = 000$ . Note também que a ordem das variáveis de saída estão desenhadas com o LSB à esquerda, ao contrário dos registradores. Isto decorre apenas pelo fato de a variável CLK entrar no circuito pela esquerda, e de o estágio LSB ser aquele que comuta com base em CLK, exigindo que o desenho seja invertido. A mudança de orientação do LSB/MSB é meramente estética e não altera o funcionamento dos contadores. Porém, lembre-se que, nos contadores assíncronos, o estágio LSB deve ser sempre conectado diretamente a CLK.

## **Contadores síncronos**

Os contadores assíncronos têm como característica principal o fato de todos os seus flip-flops estarem ligados ao sinal CLK externo.

Seu projeto lógico não deriva diretamente do efeito de divisão de frequência como nos contadores assíncronos, mas baseia-se em uma análise atenta da tabela-verdade dos valores das saídas  $Q(i)$  que se espera de um circuito contador, ou seja, da contagem numérica expressa em código binário.

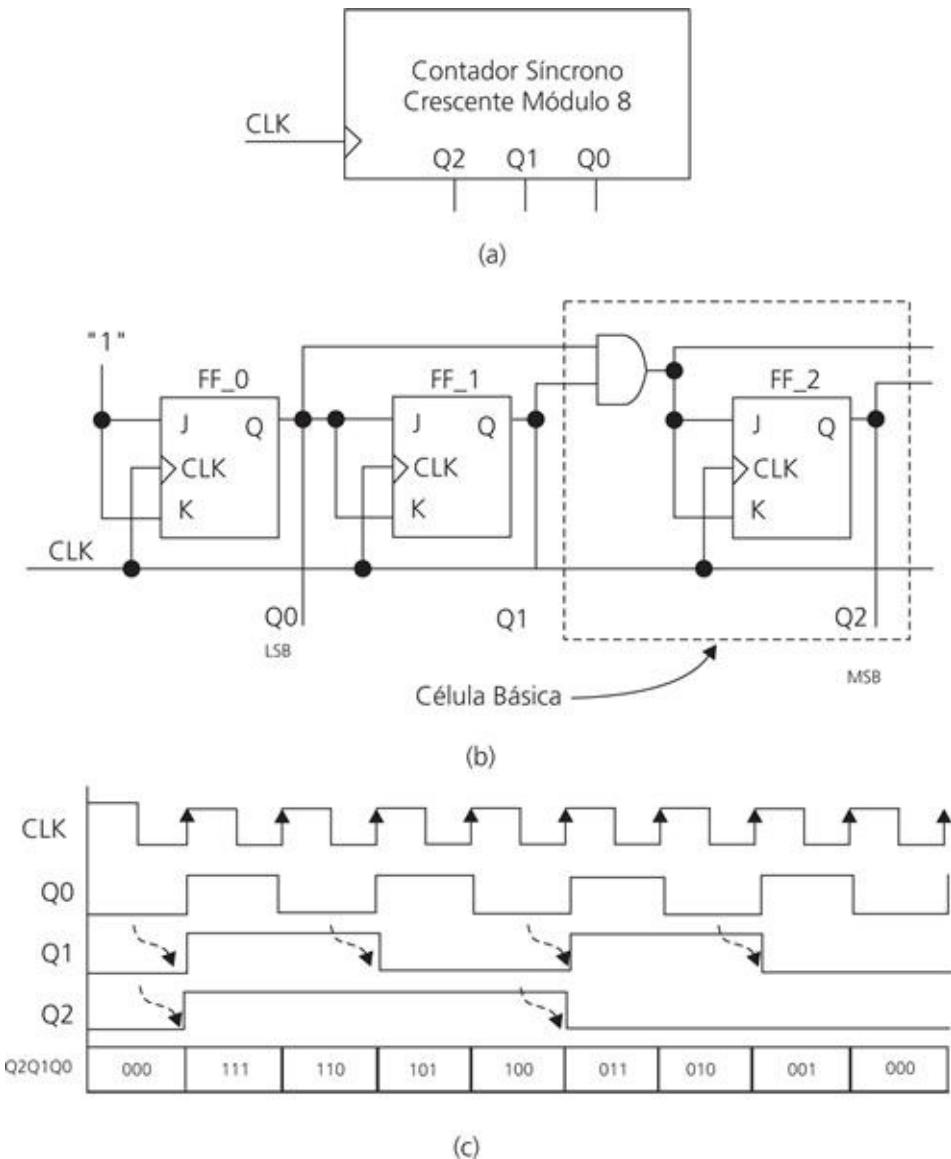
Vejamos a tabela seguinte, que lista os valores de uma contagem binária crescente cíclica de 0 a 15. As setas indicam o momento em que o bit de cada coluna comuta de valor (passa de 0 para 1 ou vice-versa).

Em um processo de contagem crescente, observamos que o bit Q0 comuta de valor a cada linha de forma totalmente independente, e que o bit Q1 prepara-se para comutar de valor (portanto, comuta na próxima linha) quando Q0 for 1. Q2 prepara-se para comutar quando Q1 e Q0 forem 1, e Q3 prepara-se para comutar quando Q2, Q1 e Q0 forem simultaneamente 1 em uma determinada linha.

Decimal	Q3	Q2	Q1	Q0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1
0	0	0	0	0

Generalizando a regra para a contagem síncrona crescente: em um contador síncrono crescente, cada estágio comuta de valor quando os estágios anteriores (de menor índice) forem todos simultaneamente de valor 1. A exceção é o estágio 0, que comuta a todo momento.

A percepção dessa regra de funcionamento nos dá uma ideia de como projetar um contador síncrono crescente. Veja na [Figura 10.9](#) um exemplo de um contador módulo 8 de três estágios.



**FIGURA 10.9** Contador síncrono crescente módulo 8: (a) símbolo lógico, (b) circuito interno e (c) formas de onda.

Como o controle do momento de comutação de cada estágio é a chave para a implementação de um contador síncrono, utilizamos flip-flops tipo JK com as entradas J e K conectadas juntas, de modo que apenas as combinações “00” (mantém) e “11” (comuta) sejam possíveis. Desta forma, pode-se controlar a

comutação de cada estágio mediante uma lógica que calcula a simultaneidade de 1's (implementada por meio das portas AND) nas saídas  $Q(i)$  anteriores.

Na [Figura 10.9.b](#), verificamos que o estágio 0 é implementado como uma célula de comutação simples, pois ele deve comutar a cada borda de subida de CLK. O estágio 1 depende apenas da variável  $Q_0$ , portanto, deixamos  $J_1$  e  $K_1$  ligados a  $Q_0$  para que  $Q_1$  comute na próxima borda de subida sempre que  $Q_0$  estiver em nível 1. Já no estágio 2, verificamos que  $Q_2$  depende de  $Q_1$  e  $Q_0$  simultaneamente em 1. Esta condição é detectada pela AND, cuja saída controla  $J_2$  e  $K_2$ , fazendo com que  $Q_2$  comute na próxima borda de subida em que  $(Q_1 \cdot Q_0)$  estiver em nível lógico 1 – e assim por diante. Note a célula básica do contador, que pode ser replicada à direita do estágio 2 para expandir a quantidade de bits do contador síncrono crescente. Os estágios 0 e 1 têm configuração especial, diferente desta célula básica.

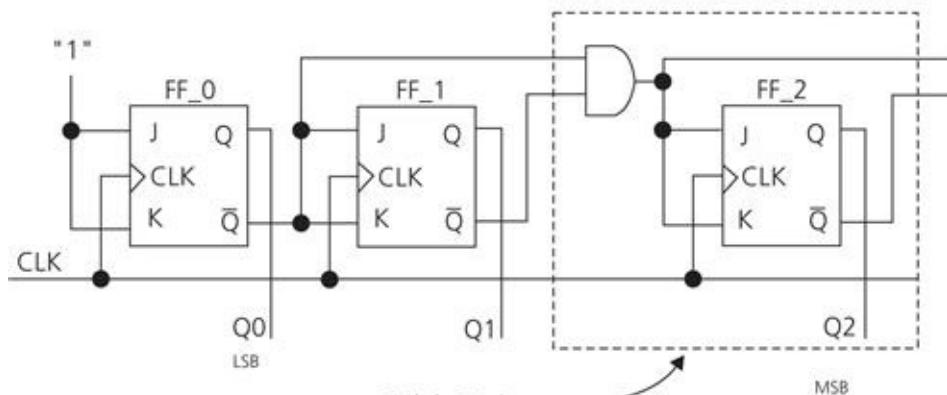
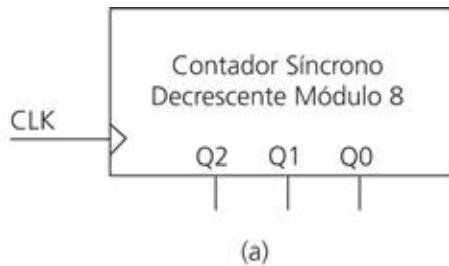
A [Figura 10.9.c](#) mostra as formas de onda desse circuito. As setas tracejadas indicam quando uma das saídas fica pronta para comutar em função das saídas anteriores, respeitando a regra de operação síncrona. Observe também que a contagem crescente ainda é mantida se os flip-flops forem trocados para borda de descida (verifique!).

Vejamos agora a tabela para uma contagem binária decrescente cíclica de 15 a 0. Em um processo de contagem decrescente, o bit  $Q_0$  também comuta de valor a cada linha de forma totalmente independente, e o bit  $Q_1$  prepara-se para comutar de valor (portanto, comuta na próxima linha) quando  $Q_0$  for 0.  $Q_2$  prepara-se para comutar quando  $Q_1$  e  $Q_0$  forem 0, e  $Q_3$  prepara-se para comutar quando  $Q_2$ ,  $Q_1$  e  $Q_0$  forem simultaneamente 0 em uma determinada linha.

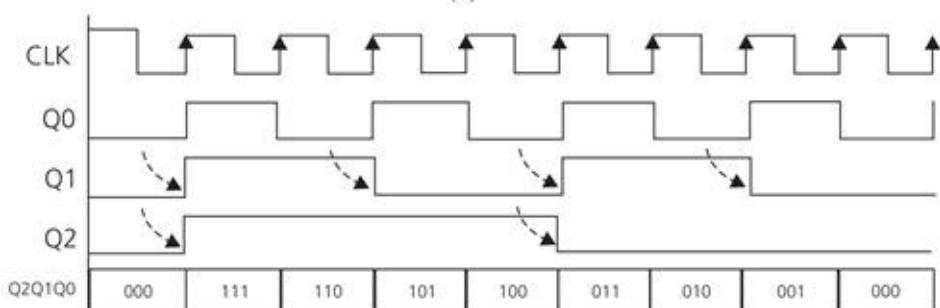
Decimal	Q3	Q2	Q1	Q0
15	1	1	1	1
14	1	1	1	0
13	1	1	0	1
12	1	1	0	0
11	1	0	1	1
10	1	0	1	0
9	1	0	0	1
8	1	0	0	0
7	0	1	1	1
6	0	1	1	0
5	0	1	0	1
4	0	1	0	0
3	0	0	1	1
2	0	0	1	0
1	0	0	0	1
0	0	0	0	0
15	1	1	1	1

Generalizando a regra para a contagem síncrona decrescente: cada estágio do contador comuta de valor quando os estágios anteriores (de menor índice) forem todos de valor 0 simultaneamente. A exceção é o estágio 0, que comuta a todo momento.

Na [Figura 10.10](#) podemos verificar que o circuito lógico de um contador síncrono decrescente é muito semelhante ao da versão crescente. A única mudança ocorre na deteção de 0's simultâneos nas saídas dos estágios anteriores, que é realizada com uma AND das Q-barra anteriores (veja que esta é a primeira vez que temos um motivo para utilizar a saída Q-barra de um flip-flops).



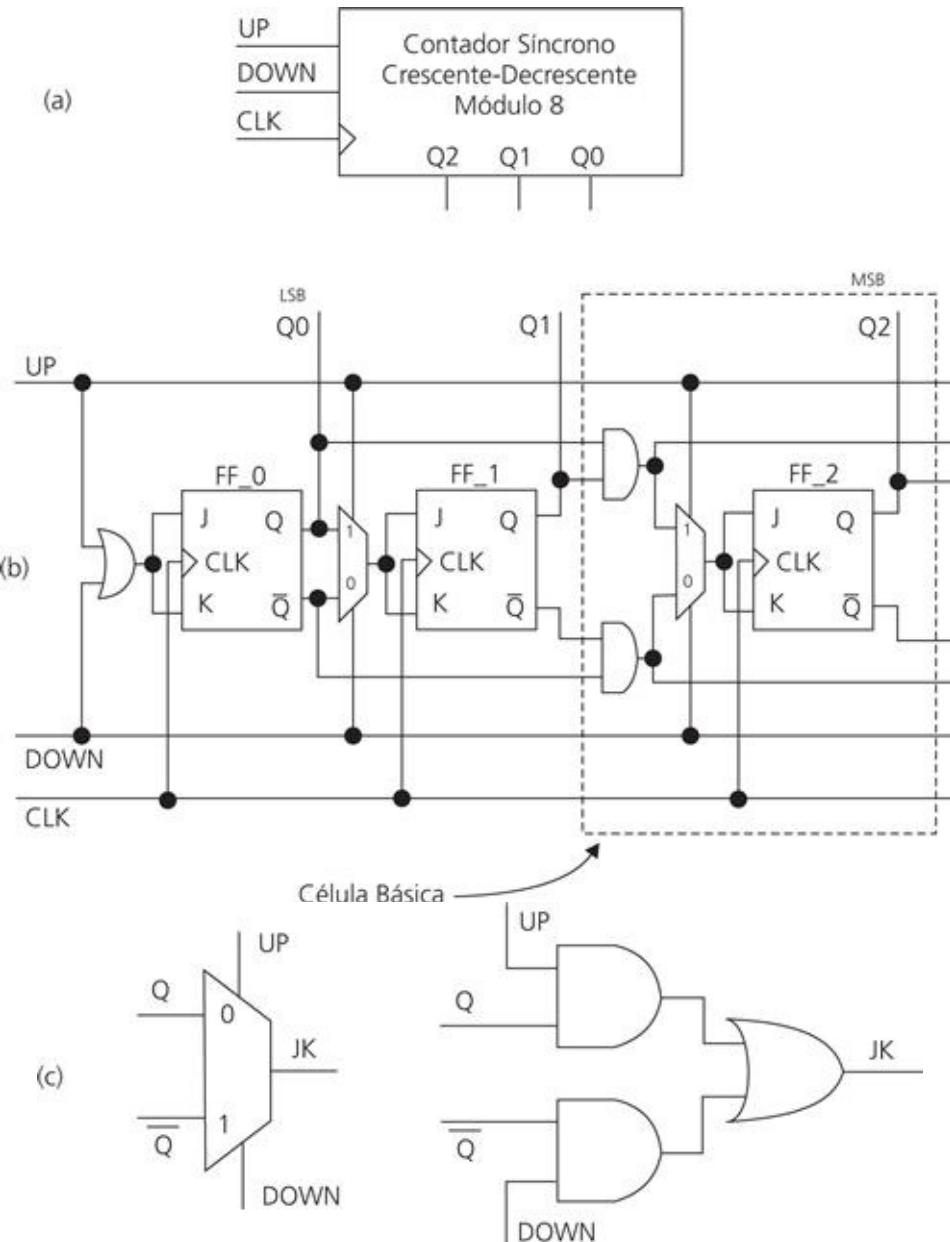
(b)



(c)

**FIGURA 10.10** Contador síncrono decrescente módulo 8: (a) símbolo lógico, (b) circuito interno e (c) formas de onda.

Os circuitos das Figuras 10.9 e 10.10 podem ser combinados para formar um contador síncrono crescente-decrescente (em inglês, *up-down counter*), que permite aplicações bastante interessantes na contagem de eventos, como no controle da quantidade de automóveis dentro de um estacionamento ou da quantidade de pessoas dentro de um ônibus. Veja o circuito correspondente na Figura 10.11.



**FIGURA 10.11** Contador síncrono crescente-decrescente módulo 8: (a) símbolo lógico, (b) circuito interno.

Contador síncrono crescente-decrescente módulo 8: (c) MUX 2:1 modificado.  
(Continuação).

A tabela-verdade a seguir identifica as variáveis de controle UP e DOWN e os respectivos modos de operação do contador.

UP	DOWN	Operação
0	0	Contagem congelada

0	1	Contagem decrescente
1	0	Contagem crescente
1	1	Combinação inválida

O controle do estágio 0 é feito por uma porta OR, pois Q0 deve comutar somente se UP = 1 ou DOWN = 1 e deve congelar se UP = DOWN = 0. Os estágios subsequentes necessitam de MUX 2:1 modificado, cujo circuito é mostrado na [Figura 10.11c](#). Considere agora a tabela-verdade a seguir. O MUX 2:1 modificado faz uso de ANDs como portas de passagem/bloqueio e resultam em uma combinação na qual a saída JK = 0 quando UP e DOWN recebem 0. Ao ser aplicada à célula de comutação do contador, esta combinação faz com que o respectivo flip-flops não comute e mantenha o bit armazenado. Esta função é a chave para obter a contagem congelada.

UP	DOWN	JK
0	0	0
0	1	Q-barra
1	0	Q
1	1	Inválida

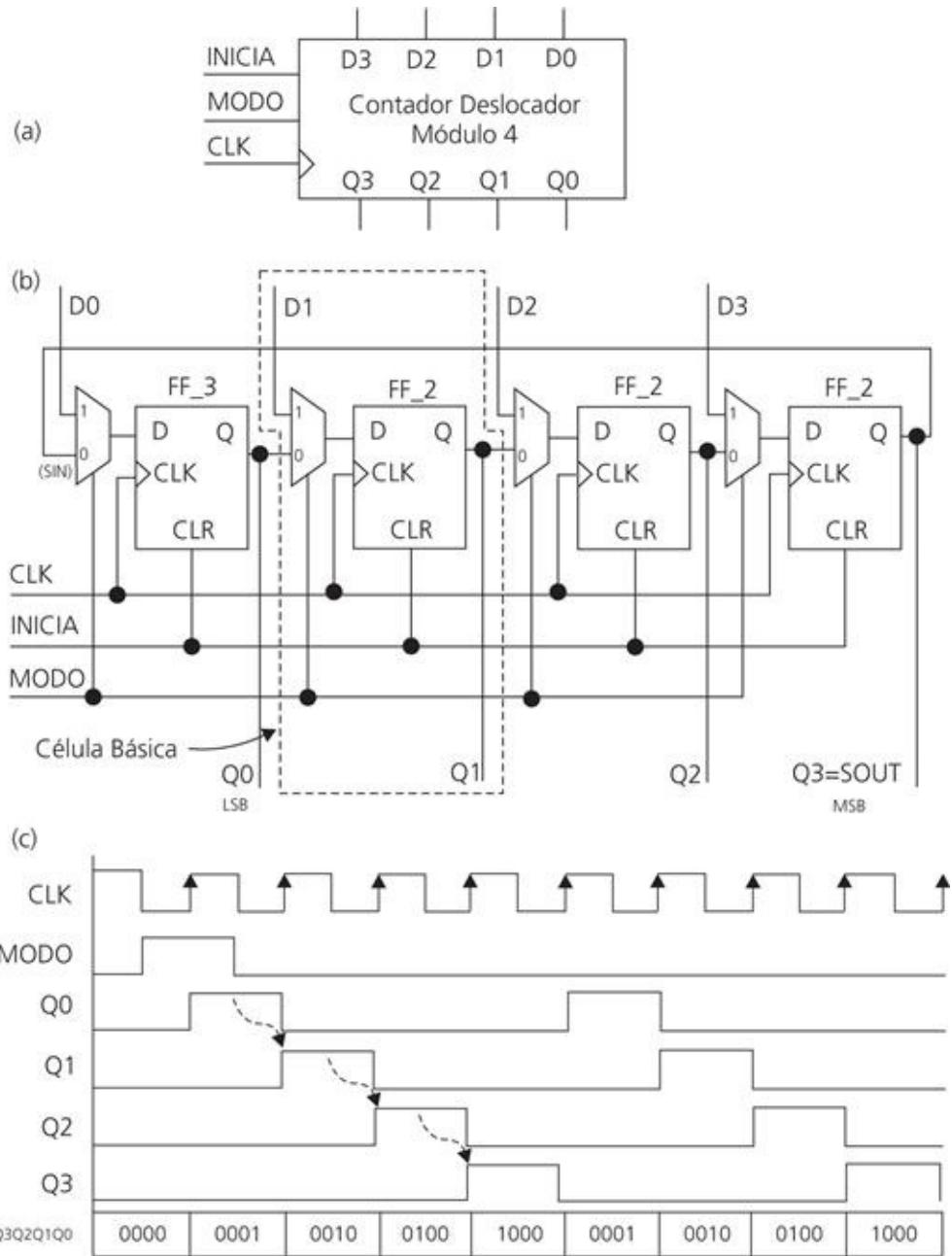
Do estágio 3 em diante, a célula básica torna-se a mesma, tal como mostrada na [Figura 10.11.b](#). A diferença é que as entradas Q e Q-barra do MUX 2:1 modificado devem receber as operações AND das respectivas entradas anteriores.

## Contadores Deslocadores

Um registrador SIPO modificado para realimentar a saída SOUT na entrada SIN dá origem a um circuito que repete ciclicamente um padrão lógico à medida que desloca os bits em seu interior. Este circuito, embora não conte em um formato numérico crescente ou decrescente, é denominado contador deslocador, pois apresenta uma quantidade bem definida de estados e, portanto, possui um módulo.

A [Figura 10.12](#) ilustra o circuito de um contador deslocador baseado em um registrador PISO, cujo padrão a ser deslocado é introduzido pelas entradas paralelas no momento inicial por meio do acionamento da variável MODO, e posteriormente os bits deslocados são reintroduzidos pela conexão de SOUT para SIN. Note nas formas de onda da [Figura 10.12.c](#) que este circuito “conta”

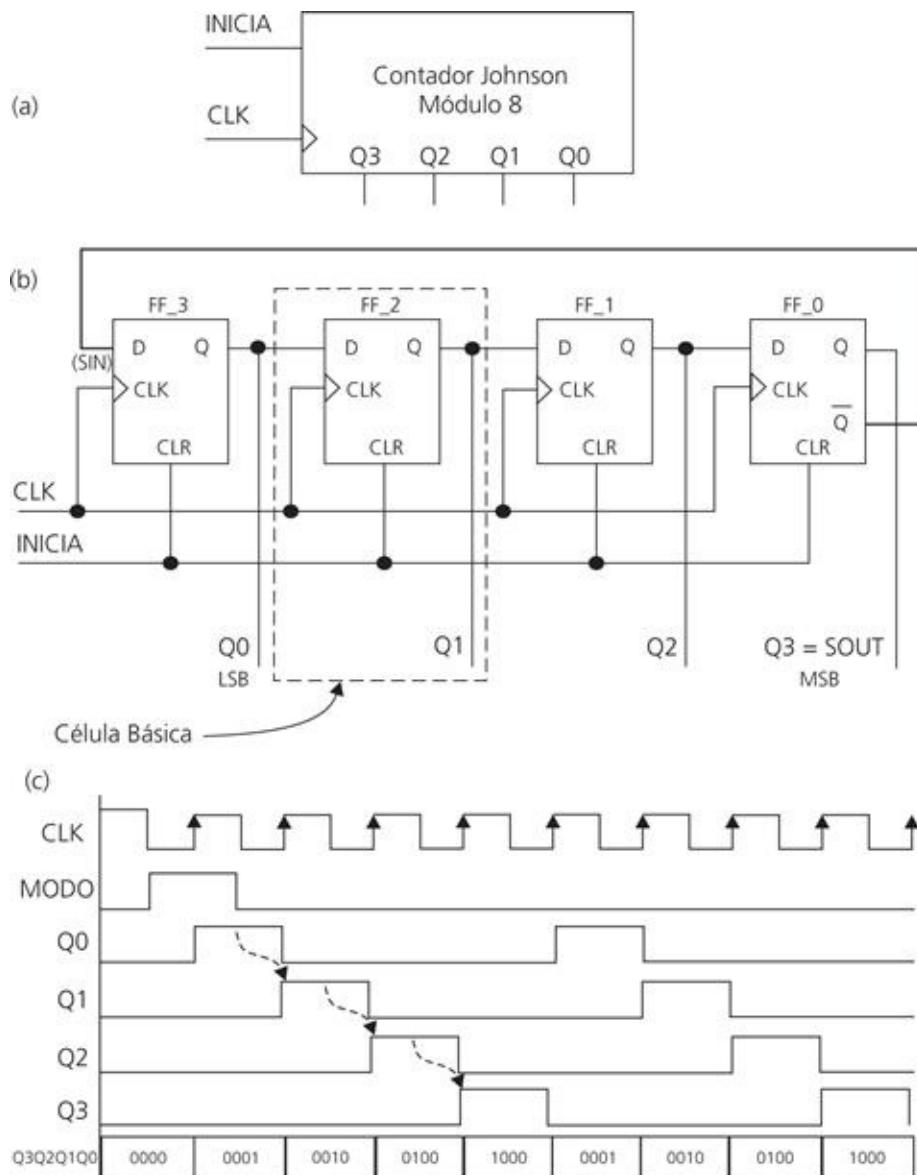
apenas quatro padrões de bits, e que seu módulo M é igual ao número de estágios N ( $M = N$ ).



**FIGURA 10.12** Contador deslocador módulo 4: (a) símbolo lógico, (b) circuito interno e (c) formas de onda.

A [Figura 10.13](#) mostra uma variação do contador deslocador derivado do registrador SIPO, que é de estrutura mais simples por não possuir entradas

paralelas. Após iniciar com valor 0000, o contador reintroduz os bits da saída Q3-barra para a entrada D0, produzindo assim uma inversão no padrão deslocado, cujo efeito é mostrado na Figura 10.13.c. Este circuito é conhecido como contador Johnson, por produzir uma contagem em código Johnson (veja a tabela de códigos da seção “Circuitos combinacionais – segunda parte). Observe na Figura 10.13.c que este circuito “conta” oito padrões de bits, e que seu módulo M é igual ao dobro do número de estágios N ( $M = 2N$ ).

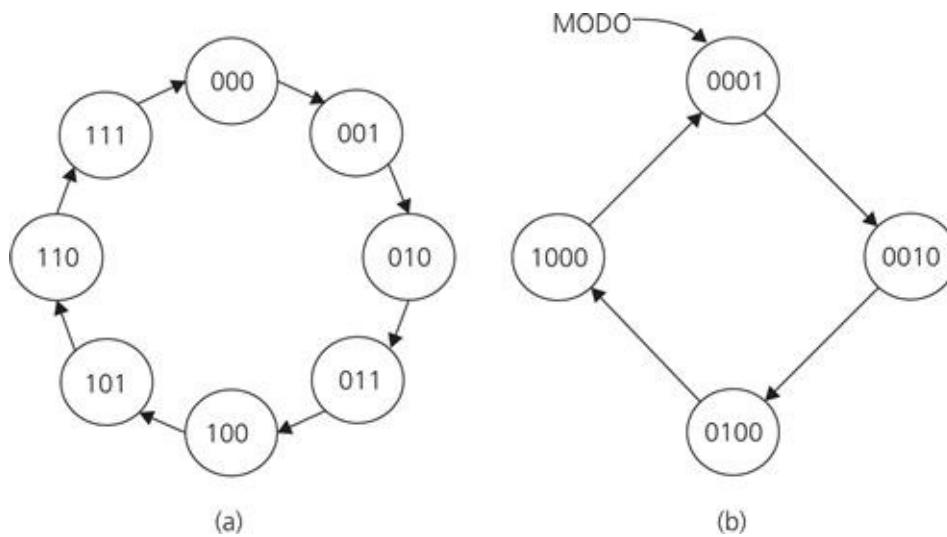


**FIGURA 10.13** Contador Johnson módulo 8: (a) símbolo lógico, (b) circuito interno e (c) formas de onda.

Geralmente não se usa os contadores deslocadores para contar eventos de forma matemática. O seu potencial, entretanto, está relacionado ao modo como gera as formas de onda das saídas. O contador deslocador da Figura 10.12 produz saídas do tipo “uma-de-cada-vez” (em inglês, *one-hot output*), ou seja, apenas uma delas está ativa a cada momento. Esta característica pode ser muito interessante na geração de sinais de controle exclusivos.

## Introdução Aos Circuitos Controladores

Um contador, qualquer que seja sua estrutura, gera um padrão de bits nas saídas a cada borda ativa de CLK. Podemos dizer que ele sequencia estados, que ficam estáveis entre duas bordas consecutivas de CLK. A Figura 10.14 ilustra o que se convenciona chamar de diagrama de estados, desenhados para os contadores das Figuras 10.9 e 10.12.

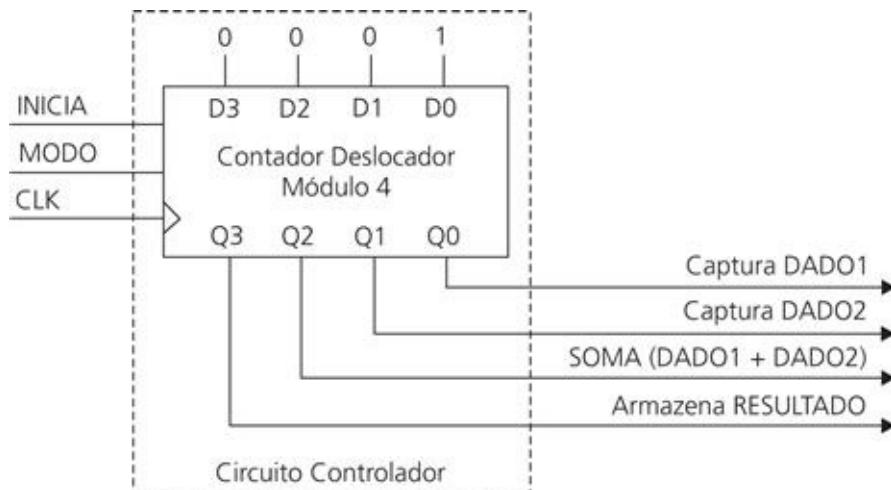


**FIGURA 10.14** Diagramas de estados (a) de um contador crescente módulo 8 e (b) de um contador deslocador módulo 4.

Os círculos indicam os estados, caracterizados pelos valores lógicos das saídas  $Q(i)$ , anotados dentro destes círculos. As setas orientadas indicam a transição de estado que ocorre a cada borda ativa de CLK. Como as transições são incondicionais, não há anotações indicativas ao lado das setas. Note que os diagramas de estados de contadores são cíclicos, pois os contadores repetem a contagem ao atingir seu valor máximo ou mínimo. Observe na Figura 10.14.b que o estado “0001” é também alcançado quando a variável MODO é acionada,

carregando este valor nos flip-flops.

Em particular, o contador deslocador módulo 4 pode ser empregado para gerar sinais mutuamente exclusivos, que servem para habilitar operações lógicas em outros circuitos. Com este enfoque, tal contador pode ser considerado um controlador, como exemplifica a [Figura 10.15](#). Observe que os controles indicados são acionados em nível lógico 1 sequencialmente, de modo que uma certa operação sob controle (no exemplo, a soma de dois números) seja executada passo a passo de forma coerente.



**FIGURA 10.15** Exemplo de circuito controlador.

Portanto, temos uma primeira solução para o problema de gerar controles sequenciados a partir de um contador deslocador semelhante ao mostrado na [Figura 10.12](#). Se uma dada aplicação requerer, por exemplo, oito sinais de controle gerados em sequência, basta ampliarmos esse circuito para que ele tenha oito estágios.



## O Que Vem Depois

Neste capítulo finalizamos a Parte 2 do livro, e vimos com detalhes como os circuitos lógicos podem ser criados para realizar funções operando variáveis lógicas. Na sequência, iremos estudar a organização básica de um computador digital e como as instruções básicas na sua forma mais simples (em linguagem de máquina) geram as variáveis de controle necessárias para pôr em

funcionamento, de forma articulada e coerente, os diversos módulos lógicos que compõem sua arquitetura interna.

---

## CAPÍTULO 11

# Organização de um computador digital

---

### Objetivos do capítulo

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Entender como um processador digital é composto e quais as funções de suas unidades.
- Entender a organização de qualquer processador com o qual venha a se deparar.



### Apresentação

Estamos tão acostumados a utilizar um computador que não paramos para analisar como ele funciona. Considerando isso, iremos apresentar a seguir as unidades que compõem os computadores, o funcionamento de cada uma delas e a maneira como interagem entre si.



Em um computador, a unidade central de processamento (UCP, ou CPU, do

inglês *Central Processing Unit*) age como se fosse nosso cérebro, processando e armazenando as informações; já as unidades de entrada e saída funcionam como nossos sentidos, e a unidade de controle pode ser vista como o sistema nervoso central do computador.

Vamos agora começar a estudar as unidades e seu funcionamento.

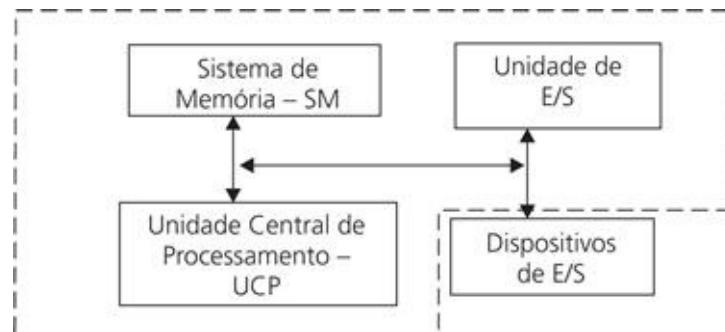
## Fundamentos

### **Organização básica de um computador digital**

Um computador digital é composto basicamente por três unidades:

- Sistema de memória
- Unidade central de processamento (UCP)
- Unidade de entrada e saída (E/S)

A [Figura 11.1](#) nos mostra esta organização.



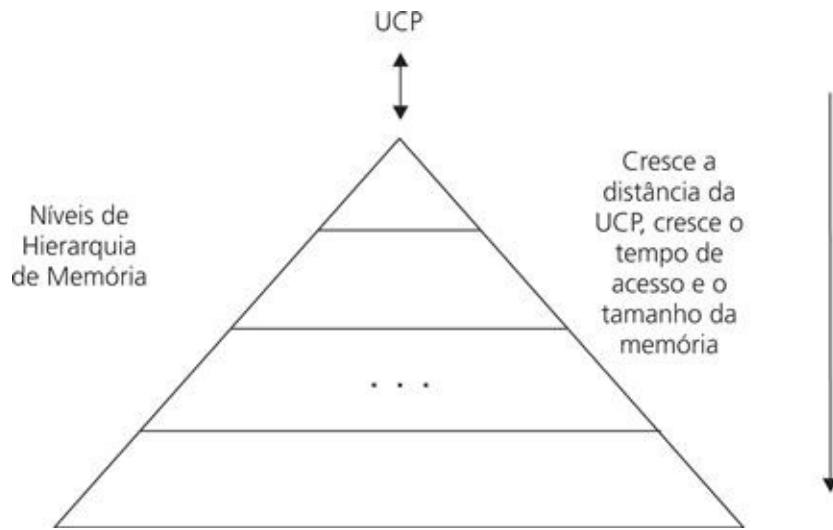
**FIGURA 11.1** Organização básica de um computador digital.

O sistema de memória (SM) é responsável pelo armazenamento das informações que serão processadas pela unidade central de processamento – UCP. A unidade de E/S é responsável pelo fluxo das informações entre o SM ou o UCP e os dispositivos de entrada e saída. Os dispositivos de E/S são responsáveis por possibilitar a interação entre o usuário e o computador.

## Sistema De Memória

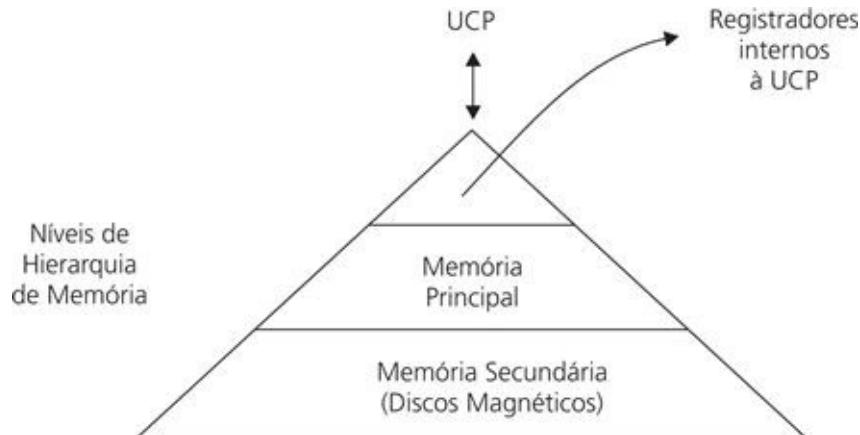
A memória de um computador é o local onde as informações são armazenadas. Na realidade, não temos uma memória, e sim um sistema de memória, composto por vários tipos de memórias, cada um com sua função. A [Figura 11.2](#) mostra o

sistema de memória utilizado nos computadores.



**FIGURA 11.2** Sistema hierárquico de memória.

Este sistema é dito hierárquico porque é composto por níveis, cada um deles correspondendo a uma ou mais memórias com características específicas. Em um sistema com três níveis de memória (como visto na [Figura 11.3](#)), por exemplo, o nível mais “próximo” à UCP são os registradores internos a esta unidade. Os registradores são as memórias cujo tempo de acesso é o mais próximo à velocidade de processamento da UCP, com um maior custo de armazenamento por bit e consequentemente uma menor capacidade de armazenamento. O próximo nível corresponde à memória principal, mais lenta que os registradores, porém com custo de armazenamento menor e maior capacidade. O último nível corresponde à memória secundária, ou seja, o disco magnético: ele é mais lento, tem custo de armazenamento menor e maior capacidade em comparação à memória principal.



**FIGURA 11.3** Sistema hierárquico de memória com três níveis.

Nos sistemas de memória, podemos ainda encontrar mais um nível entre os registradores da UCP e a memória principal, chamado de memória *cache*. Pela definição de sistema hierárquico, podemos concluir que as memórias caches são mais rápidas (e mais caras) que as principais e mais lentas (com menor custo) que os registradores da UCP.

## **Memórias semicondutoras**

As memórias semicondutoras são memórias que utilizam materiais semicondutores como o silício em sua construção. Podemos dividir este tipo de memória em três categorias: memórias RAM, ROM e *flash*.

### **Memória De Acesso Aleatório – RAM (*Random Access Memory*)**

As memórias tipo RAM são memórias voláteis; na ausência da tensão de alimentação, os valores armazenados são perdidos. Temos dois tipos de memória RAM:

- **Memórias RAM estáticas – SRAM (*Static Random Access Memory*):** seu conteúdo só é alterado quando sobrescrevemos outro valor ou quando a tensão de alimentação é desligada. São exemplos deste tipo de memória os registradores internos da UCP e a memória *cache*.

- **Memórias RAM dinâmicas DRAM (*Dynamic Random Access Memory*):** seu conteúdo tem de ser periodicamente reescrito para que não se perca. Este processo é chamado de *refresh* de memória. O nível de hierarquia de memória chamado de memória principal é um exemplo de memórias

DRAM.

## Memória Somente De Escrita – ROM (*Read Only Memory*)

As memórias tipo ROM são memórias não voláteis, ou seja, mesmo na ausência da tensão de alimentação, os valores armazenados são mantidos. Temos quatro tipos de memória ROM:

- **Memórias somente de leitura – ROM (*Read Only Memory*):** são memórias cujo conteúdo é gravado pelo fabricante da memória. Não é possível alterar as informações armazenadas após a sua gravação ([Figura 11.4](#)).



**FIGURA 11.4** Circuitos integrados (*chips*) de memória ROM.

- **Memórias somente de leitura, programáveis – PROM (*Programmable Read Only Memory*):** são memórias ROM nas quais a gravação do conteúdo é feita pelo usuário, por meio de um dispositivo especial ([Figura 11.5](#)). Não é possível alterar as informações armazenadas após a sua gravação.



**FIGURA 11.5** Programador de memórias PROM.

- **Memórias somente de leitura, programáveis e apagáveis – EPROM (*Erasable Programmable Read Only Memory*):** são memórias ROM programáveis nas quais a gravação do conteúdo é feita pelo usuário assim como nas memórias PROM. Porém, seu conteúdo pode ser apagado quando um feixe de luz ultravioleta é aplicado sobre ela ([Figura 11.6](#)).

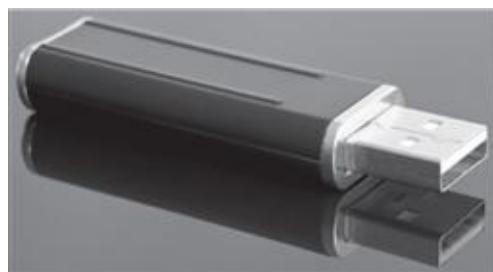


**FIGURA 11.6** Circuito integrado (chip) de memória EPROM.

- **Memórias somente de leitura, programáveis e eletricamente apagáveis – EEPROM (*Electrical Erasable Programmable Read Only Memory*):** são memórias ROM programáveis e apagáveis, assim como a memória EPROM, exceto por serem apagáveis eletricamente quando se aplica um determinado nível de tensão em um dos pinos do circuito integrado.

## Memórias *Flash*

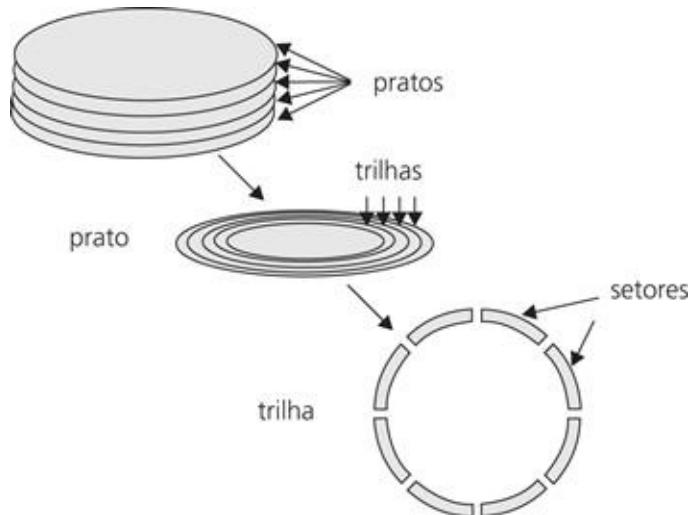
São memórias semicondutoras, não voláteis, cujo conteúdo pode ser lido e apagado sem a necessidade de dispositivos especiais. Atualmente substituem, com vantagens, as memórias da família ROM. Podemos dizer ainda que as memórias *flash* são semelhantes às memórias EEPROM, pois podemos ler e apagar seu conteúdo, mas que elas também se comportam como memórias RAM não voláteis. O processo de gravação e leitura é feito por blocos de múltiplos conteúdos. Um exemplo bem conhecido de utilização de memórias *flash* é o *pen drive* ([Figura 11.7](#)).



**FIGURA 11.7** Memória flash em um pen drive.

## Memórias magnéticas

Os representantes mais importantes deste tipo de memória são os discos rígidos. Os discos rígidos são organizados em pratos, trilhas e setores, como podemos ver na [Figura 11.8](#). Esta organização permite que os dados possam ser gravados e recuperados. Os pratos são revestidos de uma camada de material magnético, onde uma cabeça de gravação-leitura magnetiza uma região para armazenar um bit ou lê o campo magnético existente na região para ler o bit. O conjunto destas regiões é chamado de setor, e o conjunto de setores, por sua vez, equidistantes do centro do prato, forma a trilha. Finalmente, o conjunto de trilhas forma o prato.



**FIGURA 11.8** Organização de um disco magnético. Reproduzido de Patterson, David A.; Hennessy, John L *Organização e projeto de computadores – a interface hardware/software*. 2. ed. São Paulo: LTC, 2000.

Na [Figura 11.9](#) podemos ver um disco rígido comum por dentro.



**FIGURA 11.9** Vista interna de um disco rígido.

Os discos magnéticos são classificados dentro da hierarquia de memória como memória secundária, também chamada de memória de massa.

Outros tipos de memórias magnéticas são as fitas e os cartuchos magnéticos, que ainda são utilizados em cópias de segurança para sistemas de grande porte.

## Memórias Ópticas

Outro meio de armazenamento utilizado são os discos ópticos, os CDs e os DVDs. Diferentemente dos discos rígidos, as trilhas dos discos ópticos têm um

formato de espiral, como mostra a [Figura 11.10](#).



**FIGURA 11.10** Trilhas espirais de um disco óptico.

### ***Organização das memórias em endereços e conteúdos***

Uma memória de acesso aleatório precisa ter um rótulo (ou identificador) para cada posição, para que possamos acessar (ler ou escrever) de maneira rápida e eficiente um conteúdo em uma determinada posição. Este identificador é chamado de endereço, e indica a posição onde está armazenado o conteúdo. A [Figura 11.11](#) mostra esta organização considerando uma memória de 4MB ( $4 \times 2^{20}$  bytes). Note que os endereços, para esta memória, estão no intervalo de 0 a  $(4M - 1)$ .

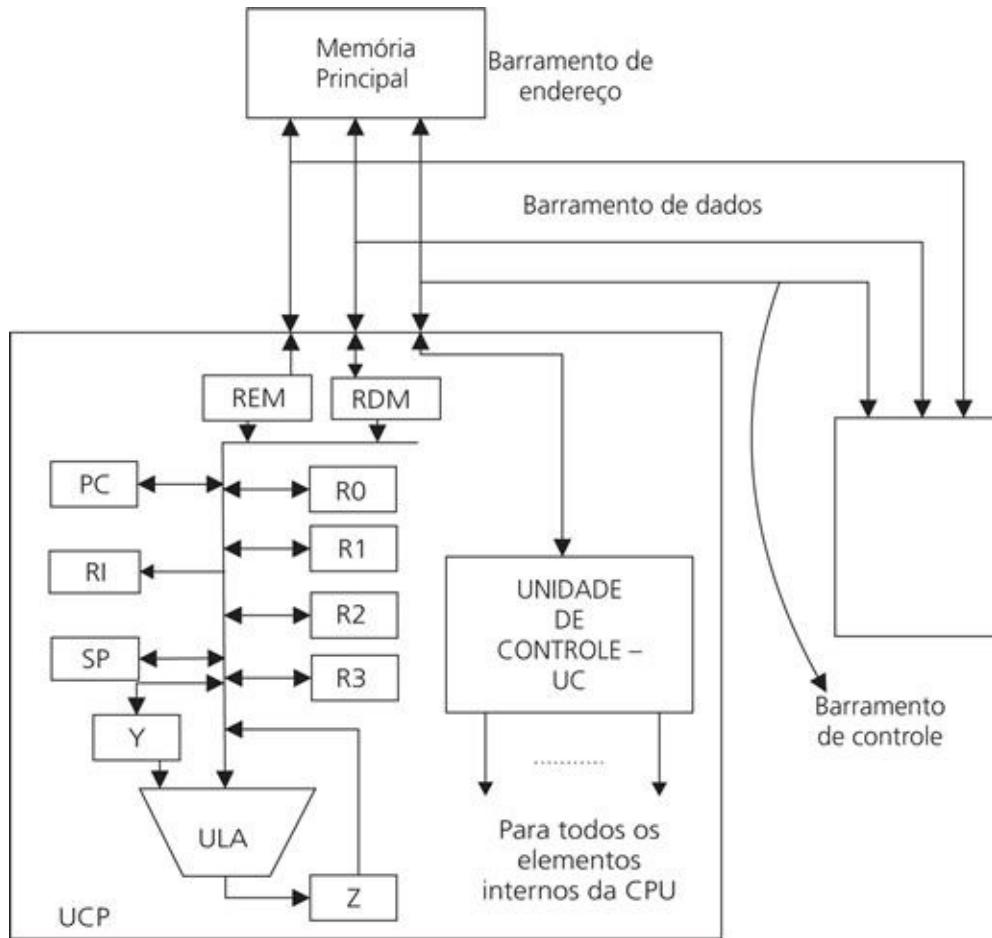
Endereço	Conteúdo
(4MB – 1)	10110101
...	...
1048576	01001010
...	...
1765	01001101
...	...
4	01010000
3	11111111
2	11101001
1	11011010
0	01100100

**FIGURA 11.11** Organização de uma memória em endereço e conteúdo.

Para que possamos ler uma informação da memória temos que fornecer a ela o respectivo endereço da informação, a qual nos será fornecida após esse processo. Quando se trata de escrever uma informação, devemos fornecer o endereço e a informação à memória para que ela armazene esta informação na posição indicada pelo endereço.

## Unidade Central De Processamento

Fazendo uma analogia com o corpo humano, a unidade central de processamento (ou simplesmente processador) seria o cérebro do computador, pois é na UCP que as informações são processadas. Para possibilitar o processamento, a UCP deve conter registradores para o armazenamento das informações e dos resultados decorrentes deste processo, uma unidade para realizar os cálculos necessários e uma unidade que coordene todas as suas operações. A [Figura 11.12](#) mostra a organização geral de uma UCP, com sua interação com o sistema de memória e a unidade de E/S.



**FIGURA 11.12** Organização geral de uma unidade central de processamento (UCP) e sua interação com o sistema de memória e a unidade de E/S.

A seguir, veremos quais elementos compõem a UCP.

### **Registradores de propósito geral (RPG)**

Os registradores de propósito geral têm a função de armazenar os dados que serão processados pela UCP. Se quisermos somar dois valores armazenados na memória, por exemplo, primeiro teremos de ler os dois números da memória e armazená-los em dois dos registradores de propósito geral para depois efetuarmos a soma.

O número de registradores de propósito geral e seus nomes dependem do processador utilizado.

### **Registradores específicos**

Como o nome já diz, esses registradores têm uma função específica, diferente da

função dos registradores de propósito geral. Normalmente os processadores têm os seguintes registradores específicos:

- **Registrador de endereços de memória (REM):** tem a função de guardar um endereço de memória que será acessado por uma instrução que tenha a função de leitura ou escrita.
- **Registrador de dados de memória (RDM):** guarda um dado lido da memória ou um dado a ser escrito na memória.
- **Registrador de instrução (RI):** tem a função de armazenar a instrução lida da memória, até o fim de sua execução.
- **Contador de programa (PC, do inglês *Program Counter*):** mantém sempre armazenado o endereço de uma instrução que será executada. Outro nome utilizado para este registrador é apontador de instrução (*IP, do inglês Instruct Pointer*).
- **Apontador de pilha (SP, do inglês *Stack Pointer*):** o registrador SP guarda sempre o endereço do último dado armazenado na pilha (uma organização lógica de armazenamento, implementada fisicamente na memória principal), ou seja, o endereço do topo da pilha.

## ***Unidade de controle (UC)***

Continuando com a analogia anterior, podemos considerar a unidade de controle como o sistema nervoso central do computador. A UC é responsável por controlar todas as atividades dentro de um computador por meio da geração de sinais elétricos (sinais de controle) para as unidades internas e externas à UCP. Estes sinais são gerados dependendo da instrução que se quer executar.

## ***Unidade lógica e aritmética (ULA)***

Todas as instruções que o processador executa lidam com operações lógicas e/ou aritméticas. A unidade lógica e aritmética tem a função de executar estas operações.

## ***Barramento interno***

O barramento interno é um conjunto de vias (fios) que interligam as unidades internas da UCP.

## ***Barramentos externos***

As ligações entre a UCP, o sistema de memória e as unidades de E/S também são

feitas pelos barramentos. Temos três tipos de barramentos:

- **Barramento de dados:** é o meio físico por onde trafegam as informações que serão manipuladas.
- **Barramento de endereços:** são os caminhos que permitem que os endereços possam ser enviados para a memória ou para a unidade de E/S.
- **Barramento de controle:** transporta os sinais de controle gerados pela unidade de controle até as demais unidades externas à UCP, como a memória ou a unidade de E/S.

Para exemplificar a utilização desses três barramentos, considere a leitura de um determinado dado da memória. Ao receber um sinal de controle de leitura enviado pelo UC através do barramento de controle, a memória reconhece que a operação se trata de uma leitura. Depois disso, o endereço da localização de memória a ser acessada é enviado pelo barramento de endereços, e o dado chega à UCP pelo barramento de dados.

## Unidade De Entrada E Saída (E/S)

As unidades de entrada e saída são responsáveis pela comunicação entre o computador e os periféricos ou dispositivos de entrada e saída. Podemos classificar as unidades de E/S em três tipos: interfaces, canais de E/S e processadores de E/S, os quais serão explicados a seguir.

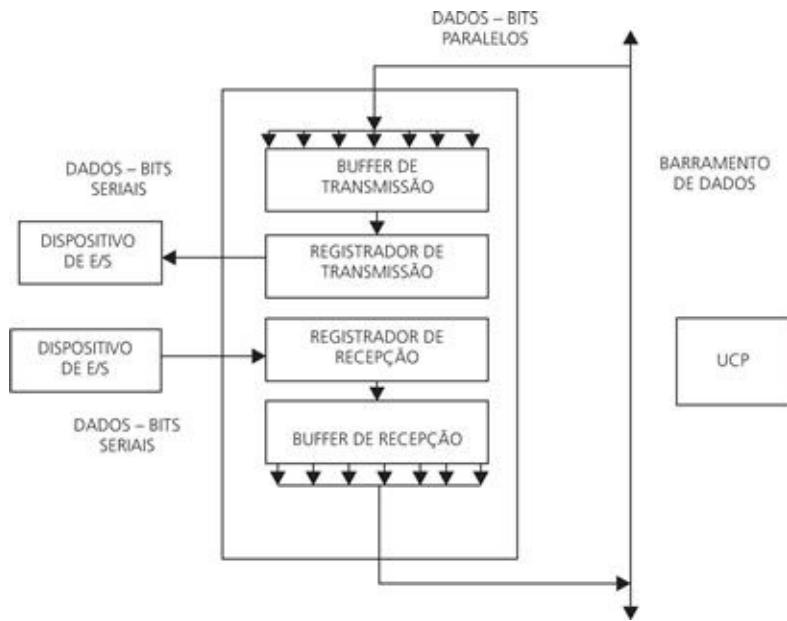
### **Interfaces**

As interfaces de E/S são circuitos que compatibilizam o formato de dados utilizado pelo computador e o formato de dados utilizado pelo dispositivo. Esta compatibilização inclui desde a maneira como os dados são transmitidos (se serial ou paralelamente, como veremos a seguir) até a velocidade de transmissão e níveis elétricos do sinal. Estes parâmetros são chamados de protocolo de comunicação. As interfaces são as unidades de E/S com menor “inteligência”, pois o grande responsável por gerenciar esta transmissão é a UCP.

### **Interface serial**

A Figura 11.13 mostra a organização básica de uma interface serial. O dado é tratado dentro da UCP de forma paralela. Se quisermos transferi-lo para um dispositivo que utiliza o dado de forma serial, a interface tem de serializar os bits de dados, e vice-versa. Assim, para que seja enviado até um dispositivo serial, o dado inicialmente é armazenado em um buffer de transmissão que o transfere ao

registrador de transmissão, onde ele é finalmente serializado e enviado.



**FIGURA 11.13** Interface serial e sua ligação com a unidade central de processamento e dispositivos de E/S.

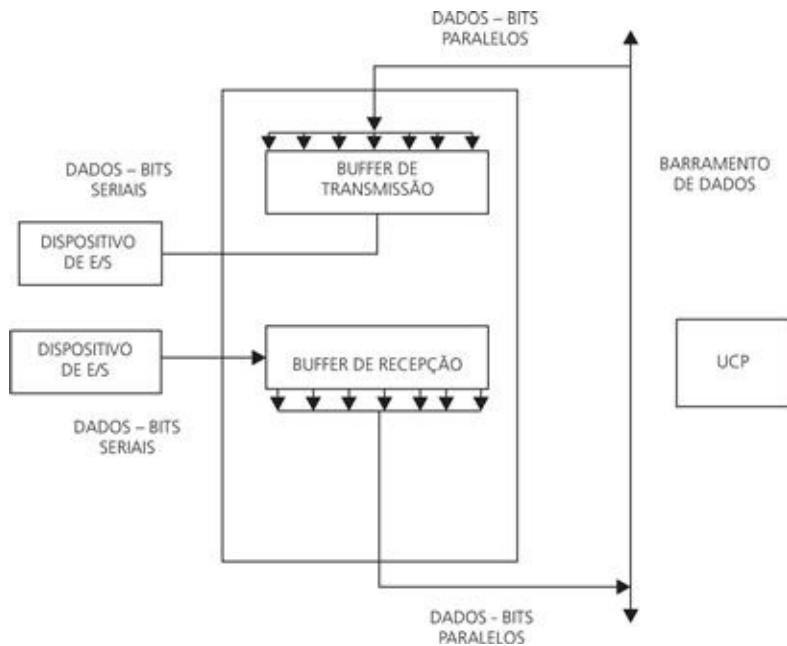
Chamamos de buffer o conjunto de registradores que armazenam temporariamente os dados que estão sendo transferidos, pois as velocidades da UCP e dos dispositivos são diferentes, e não podemos ter nenhuma perda de dados.

Na recepção de um dado de um dispositivo serial, o caminho inverso é feito: primeiro o dado é armazenado no registrador de recepção, e quando a palavra estiver completa ela é transferida para o buffer de recepção, deixando o dado disponível para a UCP.

O exemplo mais conhecido de uma interface serial é a interface padrão RS-232.

### **Interface paralela**

Na [Figura 11.14](#) temos o esquema de uma interface paralela. Neste caso, como ela é ligada a dispositivos paralelos, não há a necessidade de serializar os dados, mas apenas compatibilizar os parâmetros dos protocolos. Portanto, o dado paralelo é armazenado no buffer de transmissão – ou de recepção, se for uma operação de entrada de dados – e enviado ao dispositivo, ou à UCP.



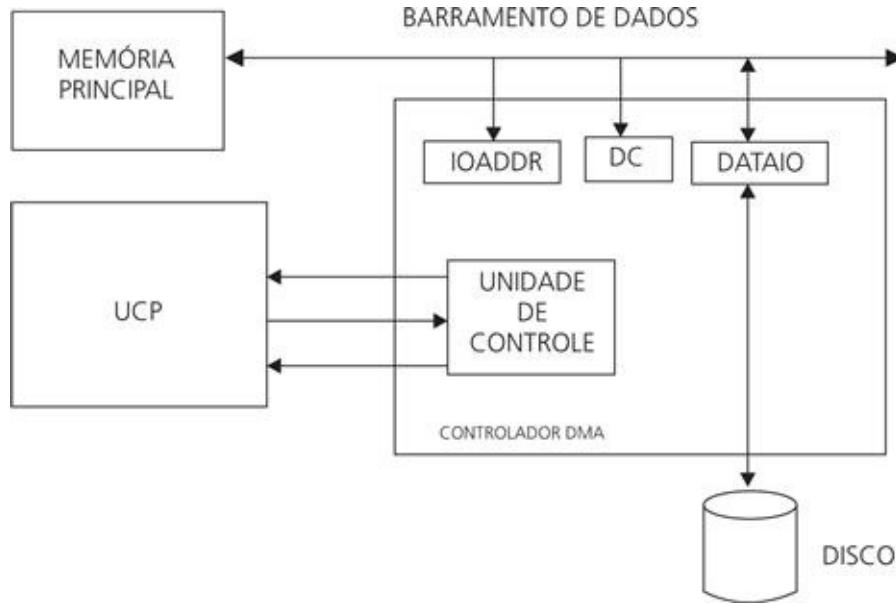
**FIGURA 11.14** Interface paralela e sua ligação com a unidade central de processamento e dispositivos de E/S.

A interface padrão RS-488 é um dos exemplos deste tipo de interface.

### **Canais de entrada e saída – E/S**

Os chamados canais de E/S são circuitos com maior capacidade de processamento. Com a ajuda desses canais, a UCP tem uma quantidade menor de trabalho ao gerenciar a E/S de dados. Os controladores de acesso direto à memória (DMA – *Direct Access Memory*) são exemplos destas unidades.

Os controladores DMA permitem que os dados sejam transferidos em blocos diretamente entre o disco e a memória, e também fazem o controle destas transferências. A UCP só tem a responsabilidade de iniciar este processo. Na Figura 11.15 temos o esquema básico de um controlador DMA.

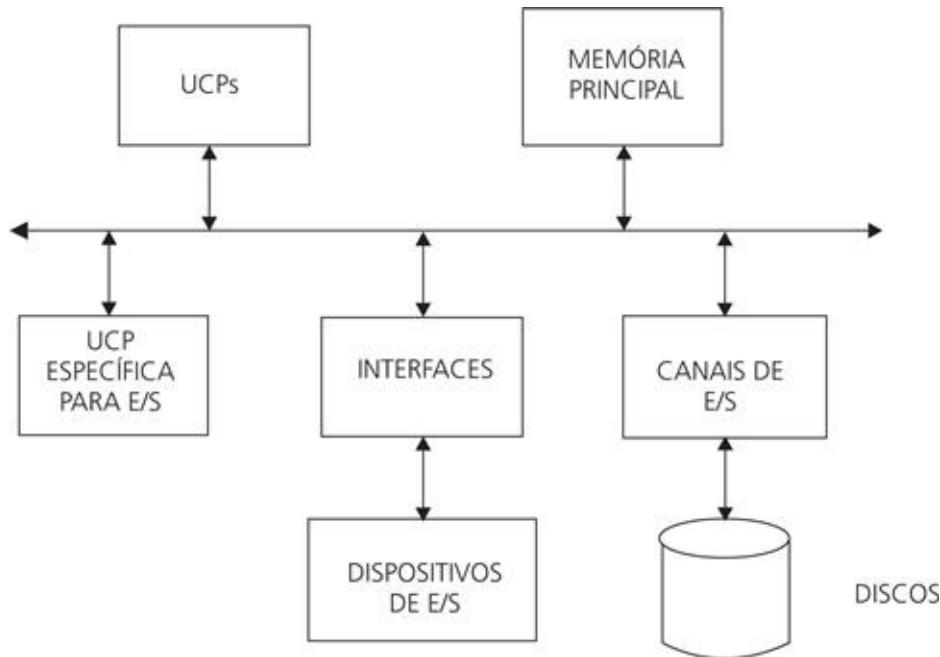


**FIGURA 11.15** Controlador de acesso direto à memória (DMA) e sua ligação com a unidade central de processamento e a memória principal.

O registrador IOADDR tem a função de guardar o endereço de memória que será acessado, e o contador DC inicialmente contém o número de palavras que serão transferidas. À medida que a transferência é feita, este contador é decrementado, e quando chega a zero a operação DMA é finalizada. O DATAIO é um buffer, uma vez que os tempos de acesso ao disco e à memória são diferentes. A unidade de controle é responsável pelo controle da operação de transferência e se comunica com a UCP por meio de sinais de controle.

### **Processadores de entrada e saída – E/S**

Um processador de E/S é uma UCP com um conjunto de instruções específico para operações de entrada e saída. Este processador é utilizado em sistemas de computação nos quais há uma necessidade de transferência de grandes volumes de dados entre o meio externo e o computador. Os grandes usuários deste tipo de unidade são os computadores com multiprocessadores utilizados em processamento paralelo – como os computadores vetoriais. Na verdade, chamamos de processadores de E/S o conjunto formado por essa UCP específica para E/S e mais as interfaces e os controladores DMA ([Figura 11.16](#)).



**FIGURA 11.16** Processador de E/S e sua ligação com a unidade central de processamento e dispositivos de E/S.

## ***Dispositivos de entrada e saída – E/S***

Os dispositivos de E/S são equipamentos que permitem que o mundo exterior se comunique com um sistema computacional. Estes dispositivos podem ser classificados, segundo alguns autores, levando-se em conta seu comportamento (entrada, saída, entrada e saída ou armazenamento) e tipo de interação apresentada – se com o ser humano ou com outra máquina.

A [Tabela 11.1](#) mostra alguns dos dispositivos mais comuns e sua classificação.

---

### **Tabela 11.1**

**Classificação dos dispositivos de entrada e saída**  
**(Reproduzido de Patterson, David A.; Hennessy, John L.**  
***Organização e projeto de computadores – a interface hardware/software. 2. ed. São Paulo: LTC, 2000.)***

---

Dispositivos	Comportamento	Parceiro
Teclado	Entrada	Humano
Mouse	Entrada	Humano
Microfone	Entrada	Humano
Scanner	Entrada	Humano
Alto-falante	Saída	Humano

Impressora	Saída	Humano
Monitor de vídeo	Saída	Humano
Modem	Saída ou Entrada	Máquina
Disco Óptico	Armazenamento	Máquina
Fita Magnética	Armazenamento	Máquina
Disco Magnético	Armazenamento	Máquina



## O Que Vem Depois

Apresentamos neste capítulo a organização de um computador digital, introduzindo também o conceito de memória hierárquica e os tipos de memória existentes, assim como as unidades responsáveis pela comunicação do computador com o mundo externo. No próximo capítulo, mostraremos como uma instrução em linguagem de máquina é executada pela UCP.

---

## CAPÍTULO 12

---

# A Execução de uma instrução e a linguagem de montagem

---

### Objetivo Do Capítulo

Ao fim deste capítulo o leitor estará apto a:

- Entender como uma instrução é executada por um processador.
- Aprender conceitos que serão imprescindíveis na programação em linguagem de montagem.
- Aprender a escrever programas mais eficientes considerando tanto seu tempo total de execução quanto seu tamanho.



### Apresentação

Seguir uma receita consiste em executar uma sequência de passos logicamente organizados para que um prato *gourmet* seja preparado.



O programa de computador se assemelha a uma receita: suas instruções são os

passos a serem seguidos pelo computador, e o prato pronto é a solução de um problema. Neste capítulo veremos quais são as instruções de um processador e como cada uma é executada.

## Fundamentos

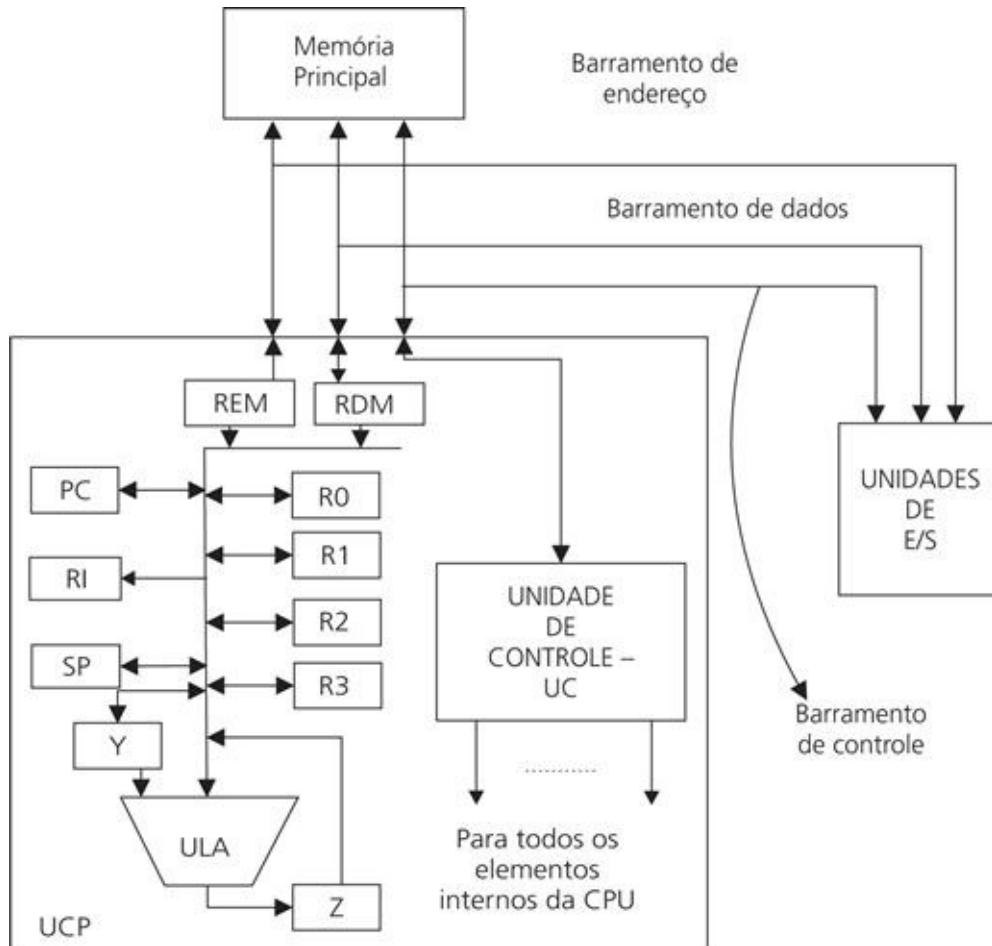
### ***A unidade central de processamento e a execução de um programa***

No [Capítulo 1](#) foi visto que um programa em linguagem de alto nível precisa ser traduzido para um programa na linguagem de máquina (código executável) a fim de ser executado e armazenado na memória principal. Vamos agora mostrar como cada instrução em linguagem de máquina é executada por uma UCP.

### ***Uma UCP hipotética***

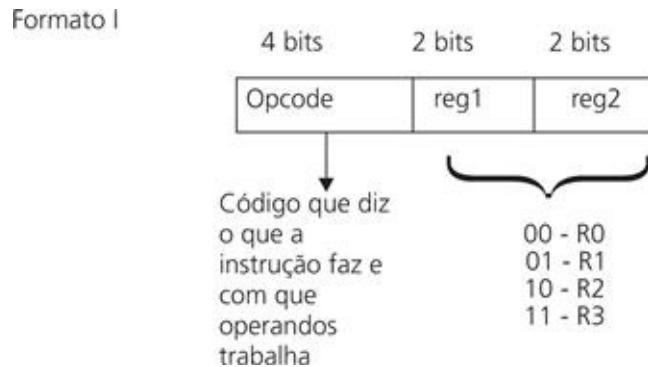
Para trabalharmos com um processador em uma linguagem de baixo nível é importante conhecer a organização da UCP (principalmente), o formato das instruções e o conjunto de instruções deste processador.

Tomaremos como base a organização da UCP mostrada na [Figura 12.1](#). Trata-se de um processador hipotético, que trabalha com quatro registradores de propósito geral: R0, R1, R2 e R3. Além destes, existem dois outros registradores auxiliares que possibilitam a correta utilização da unidade lógica e aritmética: são os registradores Y e Z. O registrador Y armazenará temporariamente um dos operandos que será usado pela ULA, e o registrador Z armazenará o resultado de uma operação lógica ou aritmética.



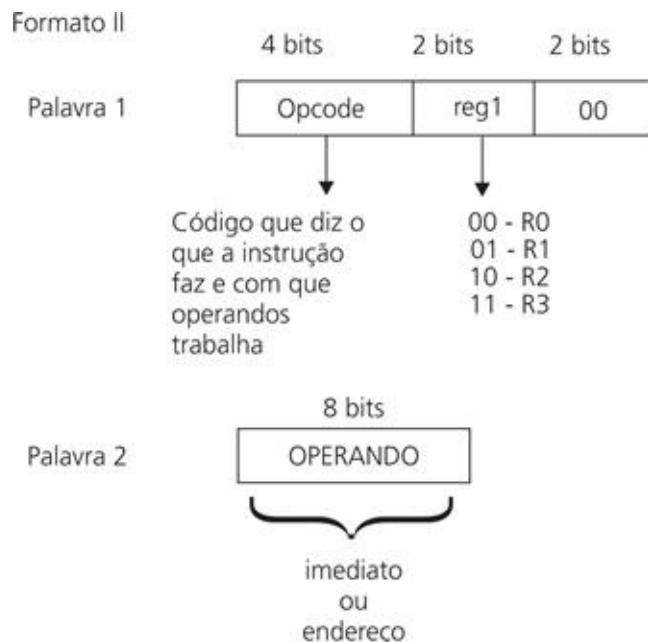
**FIGURA 12.1** Unidade central de processamento – UCP hipotética.

O conjunto de instruções dessa UCP hipotética tem dois formatos de instruções: o formato I ([Figura 12.2](#)), referente às instruções com apenas uma palavra de 8 bits, no qual o código de operação da instrução (*opcode*) tem 4 bits e os operandos têm dois campos de 2 bits (reg1 e reg2) para definir os registradores que serão manipulados pela instrução.



**FIGURA 12.2** Formato tipo I das instruções da UCP hipotética.

O formato II ([Figura 12.3](#)) é utilizado pelas instruções de duas palavras. Na primeira palavra de 8 bits, o código de operação da instrução (*opcode*) tem 4 bits e o operando possui um campo de 2 bits (*reg*) para representar o registrador que será manipulado pela instrução, junto com o outro operando de 8 bits, que é a segunda palavra. Este operando pode ser um número (imediato) ou um endereço de memória.



**FIGURA 12.3** Formato tipo II das instruções da UCP hipotética.

O conjunto de instruções desse processador consiste em quatro classes de instruções, referentes à movimentação de dados, às operações lógicas e

aritméticas, à manipulação da pilha e ao controle de fluxo de execução.

A Tabela 12.1 lista o conjunto de instruções da UCP hipotética considerada. A penúltima coluna nos mostra o significado da instrução, e a última, o seu formato.

**Tabela 12.1**

### Conjunto de instruções da UCP Hipotética

Mnemônico	Operandos	Opcode	Significado	Formato
Instruções de movimentação de dados				
MOV	Reg1,Reg2	0000	Reg1 $\leftarrow$ Reg2	I
MOV	Reg,imed	1000	Reg $\leftarrow$ imed	II
MOV	Reg,[end]	1001	Reg $\leftarrow$ [end]	II
MOV	[end],Reg	1010	[end] $\leftarrow$ Reg	II
Instruções aritméticas e lógicas				
ADD	Reg1,Reg2	0001	Reg1 $\leftarrow$ Reg1 + Reg2	I
ADD	Reg,imed	1011	Reg $\leftarrow$ Reg + imed	II
SUB	Reg1,Reg2	0010	Reg1 $\leftarrow$ Reg1 - Reg2	I
SUB	Reg,imed	1100	Reg $\leftarrow$ Reg - imed	II
AND	Reg1,Reg2	0011	Reg1 $\leftarrow$ Reg1 <u>e</u> Reg2	I
AND	Reg,imed	1101	Reg $\leftarrow$ Reg <u>e</u> imed	I
OR	Reg1,Reg2	0100	Reg1 $\leftarrow$ Reg1 <u>ou</u> Reg2	I
Instruções de manipulação de pilha				
PUSH	Reg	0101	SP--, [SP] $\leftarrow$ Reg	I
POP	Reg	0110	Reg $\leftarrow$ [SP], SP++	I
Instruções de controle de fluxo de execução				
JMP	end	1110	PC $\leftarrow$ end	II
CALL	end	1111	SP--,[SP] $\leftarrow$ PC,PC $\leftarrow$ end	II
RET	---	0111	PC $\leftarrow$ [SP], SP++	I

## **O ciclo de execução de uma instrução**

Chamamos de ciclo de execução de uma instrução as etapas realizadas pela UCP para executar uma instrução em código de máquina. O ciclo é geralmente formado por quatro etapas, as quais veremos a seguir.

### **Busca Da Instrução Na Memória (Fetch Da Instrução)**

A busca da instrução na memória principal, chamada de fetch da instrução, consiste em ler a posição da memória apontada pelo PC, armazenar a instrução lida no registrador RI e atualizar o conteúdo de PC, para que no próximo ciclo ele possa apontar para a próxima instrução. Esquematicamente, utilizando a nossa UCP hipotética, teremos:

REM $\leftarrow$ PC ;	REM recebe o conteúdo do PC.
read ;	a UC gera o sinal de leitura.
RDM $\leftarrow$ [REM] ;	RDM recebe o conteúdo do endereço que está em REM.
PC $\leftarrow$ PC + n ;	atualiza o PC para que ele aponte para a próxima palavra, que pode ser o operando desta instrução (formato II) ou outra instrução (esta instrução tem o formato I).

Para ler um conteúdo de memória, temos de enviar o endereço da posição a ser lida para a memória. Como o que iremos ler é uma instrução, o endereço está armazenado no PC; portanto, temos de colocar o valor do PC em REM. Depois disso, a unidade de controle enviará um sinal de leitura (*read*) para a memória saber qual operação será feita (leitura ou escrita). Quando a informação lida chega da memória, ela é armazenada em RDM. Como vimos anteriormente, REM e RDM são os registradores que estão diretamente ligados nos barramentos externos de endereços e dados, respectivamente. Finalmente, o PC é atualizado

somando-se um número n, exatamente o número de bytes da palavra do processador.

## Decodificação Da Instrução

Após o fetch, a instrução fica armazenada em RDM. Precisamos agora armazená-la em RI para que possamos decodificá-la. Decodificar uma instrução é saber o que a instrução faz (qual operação ela faz) e com o que ela faz (quais são seus operandos). Esta decodificação é feita pela unidade de controle, e após este passo a UC sabe quais sinais precisa gerar para executar a instrução. No nosso exemplo, temos os seguintes passos:

RI ← RDM	; RI recebe a instrução lida, que está em RDM.
decodificação	; a UC faz a decodificação da instrução.

## Leitura Dos Operandos (Busca Dos Operandos)

Se a instrução tiver um operando que ainda está na memória (notando que em nossa UCP isso só ocorre para as instruções de formato II), ele terá de ler este operando para poder executá-la. Como “operando” é a segunda palavra da instrução, ele está armazenado no próximo endereço da memória, após a primeira palavra da instrução. Portanto, quem aponta para ele é o PC. Na nossa UCP ficaria:

REM ← PC	; REM recebe o endereço do operando que está no PC.
read	; a UC gera o sinal de leitura.
RDM ← [REM]	; RDM recebe o conteúdo do endereço que está em REM.
PC ←	; atualiza o PC para que ele aponte para a próxima palavra, que pode ser o

PC + n

operando desta instrução (formato II) ou outra instrução (esta instrução tem o formato I).

Como era de se esperar, os passos são os mesmos do fetch da instrução, pois ambos são leituras de memória.

## Execução

Após realizar a decodificação e a busca de operando (caso este último passo tenha sido necessário), a UC estará pronta para executar a instrução. Executar a instrução significa fazer o que o *opcode* indica (depois da decodificação) com os operandos armazenados na UCP. As etapas da execução naturalmente dependem da instrução.

### ***Exemplos de execução das instruções da UCP hipotética***

Mostraremos, esquematicamente, a execução de todas as instruções do conjunto de instruções (veja a Tabela 12.2) da nossa UCP hipotética.

#### ■ **MOV registrador1, registrador2**

Para essa instrução, vamos considerar:

MOV R0,R1 ;R0 ← R1

É uma instrução que teria o seguinte formato:

00000001

Os quatro primeiros bits correspondem ao *opcode* da instrução **MOV registrador, número** (veja a [Tabela 12.1](#)), e os dois seguintes endereçariam o registrador de propósito geral R0. Os dois últimos correspondem ao registrador R1.

Suponha que a instrução está armazenada na memória da seguinte forma:

01	00000001
02	11111111
03	11111111
04	11111111
05	11111111
06	11111111
07	11111111

Isso significa que a instrução está armazenada no endereço 1.

Esquematicamente, teremos:

## Fetch Da Instrução

REM $\leftarrow$ PC	; REM recebe o endereço 1 para leitura.
Read	; a UC gera o sinal read para a memória.
RDM $\leftarrow$ [REM]	; o conteúdo do endereço 1 é lido e enviado pela memória para o RDM. Portanto, RDM recebe 00000001.
PC $\leftarrow$ PC + 1	; o PC é atualizado com o endereço da próxima instrução, neste caso, 2.

## Decodificação

RI $\leftarrow$ RDM	; em RDM temos a instrução e armazenamos em RI para a decodificação.
---------------------	--

Na decodificação, a UC verifica que o *opcode* 0000 (os quatro bits mais significativos) corresponde à instrução de movimentação de dados, e que o destino é o registrador R0 (os dois bits seguintes) assim como a origem é o registrador R1.

## Busca De Operando

Neste caso, como a instrução é do formato I, não há busca de operando.

## Execução

```
R0 ← R1 ; o valor armazenado em R1 será copiado para R0.
```

### ■ MOV registrador, imediato

Para esta instrução, vamos supor:

```
MOV R0,3 ;R0 ← 3
```

É uma instrução que teria o seguinte formato:

10000000
00000011

Os quatro primeiros bits correspondem ao *opcode* da instrução **MOV registrador, imediato** (veja a [Tabela 12.1](#)), e os dois seguintes endereçariam o registrador de propósito geral **R0**; os dois últimos não são considerados neste caso.

Suponha que a instrução está armazenada na memória da seguinte forma:

01	10000000
02	00000011
03	.....
04	.....
.....	.....
.....	.....

Isso significa que a instrução está armazenada no endereço 1 de memória, assim como o operando 3 está no endereço 2.

Esquematicamente, teremos:

## Fetch Da Instrução

REM $\leftarrow$ PC	; REM recebe o endereço 1 para leitura.
Read	; a UC gera o sinal read para a memória.
RDM $\leftarrow$ [REM]	; o conteúdo do endereço 1 é lido e enviado pela memória para o RDM. Portanto, RDM recebe 10000000.
PC $\leftarrow$ PC + 1	; o PC é atualizado com o endereço do operando desta instrução, neste caso 2.

## Decodificação

RI ← RDM	; em RDM temos a instrução e armazenamos em RI para a decodificação.
----------	--

Na decodificação, a UC verifica que o *opcode* 1000 (os quatro bits mais significativos) é da instrução de movimentação de dados, assim como o destino é o registrador R0 (os dois bits seguintes), além de verificar que o operando está armazenado no próximo endereço da memória.

## Busca De Operando

REM ← PC	; REM recebe o endereço 2 para leitura.
Read	; a UC gera o sinal read para a memória.
RDM ← [REM]	; o conteúdo do endereço 2 é lido e enviado pela memória para o RDM. Portanto, RDM recebe 00000011.
PC ← PC + 1	; o PC é atualizado com o endereço do operando desta instrução, neste caso 3.

## Execução

R0 ← RDM	; o valor 3 foi lido e armazenado em RDM, portanto, fazemos a transferência para o R0.
----------	--

## ■ MOV registrador, [endereço]

Esta é uma instrução de leitura de memória. Para exemplificar, usaremos:

```
MOV R2,[8] ; R2 ← [8]
```

É uma instrução que teria o seguinte formato:

```
10011000  
00001000
```

Os quatro primeiros bits correspondem ao *opcode* da instrução **MOV registrador,[endereço]** (veja a [Tabela 12.1](#)), e os dois seguintes endereçariam o registrador de propósito geral **R2**; os dois últimos não são considerados neste caso.

Suponha que a instrução está armazenada na memória da seguinte forma:

01	10011000
02	00001000
03	.....
.. .. ..	.....
08	00001001
.....	.....

Isso significa que a instrução está armazenada no endereço 1 da memória, enquanto o operando 8 está no endereço 2. Diferentemente do exemplo anterior,

aqui o operando é o endereço do número que será armazenado em R2.  
Esquematicamente, teremos:

## Fetch Da Instrução

```
REM ← PC      ; REM recebe o endereço 1 para leitura.  
Read          ; a UC gera o sinal read para a memória.  
  
RDM ← [REM] ; o conteúdo do endereço 1 é lido e  
              enviado pela memória para o RDM.  
              Portanto, RDM recebe 10011000.  
PC ← PC + 1 ; o PC é atualizado com o endereço do  
              operando desta instrução, neste caso, 2.
```

## Decodificação

```
RI ← RDM ; em RDM temos a instrução e armazenamos em RI para a decodificação.
```

Na decodificação, a UC verifica que o *opcode* 1001 (os quatro bits mais significativos) é da instrução de movimentação de dados, e que o destino é o registrador R2 (os dois bits seguintes); o operando é o endereço do dado que será armazenado no registrador.

## Busca De Operando

REM $\leftarrow$ PC	; REM recebe o endereço 2 para leitura.
read	; a UC gera o sinal read para a memória.
RDM $\leftarrow$ [REM]	; o conteúdo do endereço 2 é lido e enviado pela memória para o RDM. Portanto, RDM recebe 00001000.
PC $\leftarrow$ PC + 1	; o PC é atualizado com o endereço do operando desta instrução, neste caso, 3.

## Execução

REM $\leftarrow$ RDM	; o valor lido 8 é armazenado em REM. read a UC gera o sinal de leitura.
RDM $\leftarrow$ [REM]	; o RDM neste caso recebe o valor 9, que é o conteúdo do endereço 8.
R2 $\leftarrow$ RDM	; R2 recebe o conteúdo de RDM, neste caso, 9.

### ■ MOV [endereço], registrador

Esta é uma instrução de escrita de memória. Para exemplificar, usaremos:

MOV [8],R3 ;[8]  $\leftarrow$  R3

É uma instrução que teria o seguinte formato:

10101100
00001000

Os quatro primeiros bits correspondem ao *opcode* da instrução **MOV [endereço],registrador** (veja a [Tabela 12.1](#)), e os dois seguintes estariam endereçando o registrador de propósito geral **R3**; os dois últimos não são considerados neste caso.

Suponha que a instrução está armazenada na memória da seguinte forma:

01	10101100
02	00001000
03	.....
.....	.....
08	.....
.....	.....

Isso significa que a instrução está armazenada no endereço 1 de memória e que o operando 8 está no endereço 2.

Esquematicamente, teremos:

## Fetch Da Instrução

REM $\leftarrow$ PC ; REM recebe o endereço 1 para leitura.
read ; a UC gera o sinal read para a memória.
RDM $\leftarrow$ [REM] ; o conteúdo do endereço 1 é lido e enviado pela memória para o RDM. Portanto, RDM recebe 10101100.

$PC \leftarrow PC + 1$  ; o PC é atualizado com o endereço do operando desta instrução, neste caso, 2.

## Decodificação

$RI \leftarrow RDM$  ; em RDM temos a instrução e armazenamos em RI para a decodificação.

Na decodificação, a UC verifica que o *opcode* 1010 (os quatro bits mais significativos) corresponde à instrução de movimentação de dados e que o destino é a memória, localizada no endereço 8. Este é o operando, e a origem é o registrador de propósito geral R3 indicado pelos dois bits que estão depois dos bits de *opcode*.

## Busca De Operando

$REM \leftarrow PC$	; REM recebe o endereço 2 para leitura.
read	; a UC gera o sinal read para a memória.
$RDM \leftarrow [REM]$	; o conteúdo do endereço 2 é lido e enviado pela memória para o RDM. Portanto, RDM recebe 00001000.
$PC \leftarrow PC + 1$	; o PC é atualizado com o endereço da próxima instrução, neste caso, 3.

## Execução

REM ← RDM	; o endereço 8 foi lido e
RDM ← R3	; armazenado em RDM, portanto fazemos a transferência para o REM do valor a ser escrito em RDM.
write	; a UC gera o sinal write para a memória.
[REM] ← RDM	; a escrita é feita.

### ■ ADD registrador1, registrador2

Para esta instrução, usaremos o exemplo:

```
ADD R0, R1 ; R0 ← R0 + R1
```

É uma instrução que teria o seguinte formato:

```
00010001
```

Os quatro primeiros bits correspondem ao *opcode* da instrução **ADD registrador,registraror** (veja a [Tabela 12.1](#)), e os dois seguintes endereçariam o registrador de propósito geral **R0**. Os dois últimos bits correspondem ao registrador **R1**.

Suponha que a instrução está armazenada na memória da seguinte forma:

01	00010001
02	.....
03	.....
04	.....
.....	.....
.....	.....

Isso significa que a instrução está armazenada no endereço 1. Esquematicamente, teremos:

## Fetch Da Instrução

REM $\leftarrow$ PC	; REM recebe o endereço 1 para leitura.
read	; a UC gera o sinal read para a memória.
RDM $\leftarrow$ [REM]	; o conteúdo do endereço 1 é lido e enviado pela memória para o RDM. Portanto, RDM recebe 00010001.
PC $\leftarrow$ PC + 1	; o PC é atualizado com o endereço da próxima instrução, neste caso, 2.

## Decodificação

RI  $\leftarrow$  RDM ; em RDM temos a instrução e armazenamos em RI para a decodificação.

Na decodificação, a UC verifica que o *opcode* 0001 (os quatro bits mais

significativos) é da instrução de soma, e que o destino é o registrador R0 (os dois bits seguintes), além de verificar também que os registradores R0 e R1 são a origem.

## Busca De Operando

Neste caso, como a instrução é do formato I, não há busca de operando.

## Execução

$Y \leftarrow R0$	; o valor armazenado em R0 é transferido para Y. Y sempre vai armazenar um dos valores que serão usados pela ULA em suas operações.
$Z \leftarrow Y + R1$	; o resultado da operação é armazenado em Z.
$R0 \leftarrow Z$	; o conteúdo de Z é copiado para R0.

Este exemplo serve como base para todas as instruções que utilizam a ULA (SUB, AND e OR), além da ADD, em operações entre dois registradores.

### ■ AND registrador,imediato

Usaremos a seguinte instrução:

AND R0,3	; R0 $\leftarrow$ R0 and 3
----------	----------------------------

Esta é uma instrução que teria o seguinte formato:

11010000

00000011

Os quatro primeiros bits correspondem ao *opcode* da instrução **AND registrador,imediato** (veja a [Tabela 12.1](#)), e os dois seguintes estariam endereçando o registrador de propósito geral **R0**. Os dois últimos não são considerados neste caso.

Suponha que a instrução está armazenada na memória da seguinte forma:

01	11010000
02	00000011
03	.....
04	.....
.....	.....
.....	.....

Isso significa que a instrução está armazenada no endereço 1 de memória e que o operando 3 está no endereço 2.

Esquematicamente, teremos:

## Fetch Da Instrução

REM $\leftarrow$ PC	; REM recebe o endereço 1 para leitura.
read	; a UC gera o sinal read para a memória.
RDM $\leftarrow$ [REM]	; o conteúdo do endereço 1 é lido e enviado pela memória para o RDM. Portanto, RDM recebe 11010000.
PC $\leftarrow$ PC + 1	; o PC é atualizado com o endereço do operando desta instrução, neste caso, 2.

## Decodificação

RI ← RDM ; em RDM temos a instrução e armazenamos em RI para a decodificação.
---

Na decodificação, a UC verifica que o *opcode* 1101 (os quatro bits mais significativos) é da instrução lógica (AND), cujos termos são o conteúdo do registrador R0 (os dois bits seguintes), além de verificar que o operando está armazenado no próximo endereço da memória. O resultado será armazenado no próprio R0.

## Busca De Operando

REM ← PC	; REM recebe o endereço 2 para leitura.
read	; a UC gera o sinal read para a memória.
RDM ← [REM]	; o conteúdo do endereço 2 é lido e enviado pela memória para o RDM. Portanto, RDM recebe 00000011.
PC ← PC + 1	; o PC é atualizado com o endereço da próxima instrução, neste caso, 3.

## Execução

Y ← R0	; o valor armazenado em R0 é transferido para Y. Y sempre vai armazenar um dos valores que serão usados pela ULA em suas operações.
Z ← Y and RDM	; o outro valor foi lido e está em RDM. O resultado da operação é armazenado em Z.

R0  $\leftarrow$  Z ; o conteúdo de Z é copiado para R0.

Este exemplo serve como base para todas as instruções que utilizam a ULA (ADD, SUB e OR), além da AND, em operações entre um registrador e um imediato.

### ■ PUSH registrador

Para esta instrução, vamos supor:

PUSH R0 ; SP  $\leftarrow$  SP-1 e [SP]  $\leftarrow$  R0

É uma instrução que teria o seguinte formato:

01010000

Os quatro primeiros bits correspondem ao *opcode* da instrução **PUSH registrador** (veja a [Tabela 12.1](#)), e os dois seguintes estariam endereçando o registrador de propósito geral **R0**. Os dois últimos não são considerados.

Suponha que a instrução está armazenada na memória da seguinte forma:

01	01010000
02	.....
03	.....
04	.....
.....	.....
.....	.....

Isso significa que a instrução está armazenada no endereço 1.

Esquematicamente, teremos:

## Fetch Da Instrução

REM $\leftarrow$ PC	; REM recebe o endereço 1 para leitura.
read	; a UC gera o sinal <i>read</i> para a memória.
REM $\leftarrow$ [REM]	; o conteúdo do endereço 1 é lido e enviado pela memória para o RDM. Portanto, RDM recebe 01010000.
PC $\leftarrow$ PC +	; o PC é atualizado com o endereço da próxima instrução, neste caso, 2.

## Decodificação

RI $\leftarrow$ RDM ; em RDM temos a instrução e armazenamos em RI para a decodificação.
--

Na decodificação, a UC verifica que o *opcode* 0101 (os quatro bits mais significativos) é da instrução escrita na pilha, ou seja, uma instrução que escreve na memória, no topo da pilha. Para isso, iremos primeiro fazer com que o SP aponte para uma posição livre da pilha (o novo topo), subtraindo 1 da posição

atual. O valor a ser empilhado é, neste caso, o registrador R0 (os dois bits seguintes).

## Busca De Operando

Neste caso, como a instrução é do formato I, não há busca de operando.

## Execução

SP $\leftarrow$ SP-1	; o endereço do topo da pilha é atualizado.
REM $\leftarrow$ SP	; é armazenado em REM, pois será feito uma escrita na memória.
RDM $\leftarrow$ R0	; o valor a ser escrito é armazenado em RDM.
write	; a UC gera o sinal write para a memória.
[REM] $\leftarrow$ RDM	; a escrita é feita.

### ■POP registrador

Para esta instrução, vamos supor:

POP R1 ; R1 $\leftarrow$ [SP] e SP $\leftarrow$ SP+1
--

É uma instrução que teria o seguinte formato:

01100100
----------

Os quatro primeiros bits correspondem ao *opcode* da instrução **POP registrador** (veja a [Tabela 12.1](#)), e os dois seguintes estariam endereçando o registrador de propósito geral **R1** (os dois últimos não são considerados).

Suponha que a instrução está armazenada na memória da seguinte forma:

01	01100100
02	.....
03	.....
04	.....
.....	.....
.....	.....

Isso significa que a instrução está armazenada no endereço 1.

Esquematicamente, teremos:

## Fetch Da Instrução

REM $\leftarrow$ PC	; REM recebe o endereço 1 para leitura.
read	; a UC gera o sinal read para a memória.
RDM $\leftarrow$ [REM]	; o conteúdo do endereço 1 é lido e enviado pela memória para o RDM. Portanto, RDM recebe 01100100.
PC $\leftarrow$ PC + 1	; o PC é atualizado com o endereço da próxima instrução, neste caso 2.

## Decodificação

RI $\leftarrow$ RDM	; em RDM temos a instrução e armazenamos em RI para a decodificação.
---------------------	--

Na decodificação, a UC verifica que o *opcode* 0110 (os quatro bits mais significativos) é da instrução leitura na pilha, ou seja, uma instrução que lê da memória, no topo da pilha. O valor a ser lido é, neste caso, armazenado no registrador R1 (os dois bits seguintes). Depois da leitura é necessário somar 1 ao SP, para que ele aponte para o novo topo.

## Busca De Operando

Neste caso, como a instrução é do formato I, não há busca de operando.

## Execução

REM $\leftarrow$ SP	; é armazenado em REM, pois será feita uma leitura da memória.
SP $\leftarrow$ SP+1	; o endereço do topo da pilha é atualizado.
read	; a UC gera o sinal read para a memória.
RDM $\leftarrow$ [REM]	; a leitura é feita.
R1 $\leftarrow$ RDM	; o valor lido é armazenado em R1.

## ■JMP endereço

Para esta instrução, vamos supor:

JMP 8	; PC $\leftarrow$ 8;
-------	----------------------

É uma instrução que teria o seguinte formato:

11100000
00001000

Os quatro primeiros bits correspondem ao *opcode* da instrução **JMP endereço** (veja a [Tabela 12.1](#)), e os demais bits da primeira palavra não são considerados. A segunda palavra contém o endereço para onde se quer desviar, ou seja, o endereço da próxima instrução que será executada.

Suponha que a instrução está armazenada na memória da seguinte forma:

01	11100000
02	00001000
03	.....
04	.....
.....	.....
.....	.....

Isso significa que a instrução está armazenada no endereço 1 e que o operando está no endereço 2.

Esquematicamente, teremos:

## Fetch Da Instrução

REM $\leftarrow$ PC	; REM recebe o endereço 1 para leitura.
read	; a UC gera o sinal read para a memória.
RDM $\leftarrow$	; o conteúdo do endereço 1 é lido e enviado pela memória para o RDM.

[REM]	Portanto, RDM recebe 11100000.
PC $\leftarrow$ PC + 1	; o PC é atualizado com o endereço do operando desta instrução, neste caso 2.

## Decodificação

RI $\leftarrow$ RDM	; em RDM temos a instrução e armazenamos em RI para a decodificação.
---------------------	--

Na decodificação, a UC verifica que o *opcode* 1110 (os quatro bits mais significativos) é da instrução desvio, ou seja, uma instrução que alterará o PC com o valor do operando.

## Busca De Operando

REM $\leftarrow$ PC	; REM recebe o endereço 2 para leitura.
read	; a UC gera o sinal read para a memória.
RDM $\leftarrow$ [REM]	; o conteúdo do endereço 2 é lido e enviado pela memória para o RDM. Portanto, RDM recebe 00001000.
PC $\leftarrow$ PC + 1	; o PC é atualizado com o endereço da próxima instrução, neste caso, 3, embora a execução desta instrução vá modificá-lo.

## Execução

**PC ← RDM** ; o endereço a ser armazenado em PC, como já foi lido, está em RDM. Portanto, temos que copiá-lo em PC.

## ■CALL endereço

Para esta instrução, vamos supor:

**CALL 8** ; SP ← SP-1, [SP] ← PC e PC ← 8;

Esta é uma instrução que teria o seguinte formato:

11110000  
00001000

Os quatro primeiros bits correspondem ao *opcode* da instrução **CALL endereço** (veja a [Tabela 12.1](#)), e os demais bits da primeira palavra não são considerados. A segunda palavra contém o endereço para onde se quer desviar, ou seja, o endereço da próxima instrução que será executada, que é a primeira instrução de uma sub-rotina (procedimento ou função). A segunda palavra da instrução é o endereço do procedimento.

Suponha que a instrução está armazenada na memória da seguinte forma:

01	11110000
02	00001000
03	.....
04	.....
.....	.....
.....	.....

Isso significa que a instrução está armazenada no endereço 1 e o operando no endereço 2.

Esquematicamente, teremos:

## Fetch Da Instrução

REM $\leftarrow$ PC	; REM recebe o endereço 1 para leitura.
read	; a UC gera o sinal read para a memória.
RDM $\leftarrow$ [REM]	; o conteúdo do endereço 1 é lido e enviado pela memória para o RDM. Portanto, RDM recebe 11110000.
PC $\leftarrow$ PC + 1	; o PC é atualizado com o endereço do operando desta instrução, neste caso, 2.

## Decodificação

RI $\leftarrow$ RDM ; em RDM temos a instrução e armazenamos em RI para a decodificação.
--

Na decodificação, a UC verifica que o *opcode* 1111 (os quatro bits mais significativos) é da instrução desvio, ou seja, uma instrução que alterará o PC com o valor do operando. Mas como temos de armazenar o endereço de retorno

(PC atual)

## Busca De Operando

```
REM ← PC      ; REM recebe o endereço 2 para leitura.  
read          ; a UC gera o sinal read para a memória.  
  
RDM ← [REM] ; o conteúdo do endereço 2 é lido e  
              enviado pela memória para o RDM.  
              Portanto, RDM recebe 00001000.  
PC ← PC + 1 ; o PC é atualizado com o endereço da  
              próxima instrução, neste caso, 3,  
              embora a execução desta instrução  
              irá modificá-lo.
```

## Execução

Y ← RDM	; armazenamos temporariamente o endereço lido.
SP ← SP-1	; o endereço do topo da pilha é atualizado.
REM ← SP	; é armazenado em REM, pois será feita uma escrita na memória.
RDM ← PC	; o valor de PC vai para RDM.
write	; a UC gera o sinal write para a memória.
[REM] ← RDM	; o endereço de retorno é empilhado.
PC ← Y	; o endereço do procedimento é armazenado no PC.

## ■RET

Para esta instrução, vamos supor:

RET	; PC ← [SP]; SP ← SP+1
-----	------------------------

Esta é uma instrução que teria o seguinte formato:

01110000
----------

Os quatro primeiros bits correspondem ao *opcode* da instrução **RET** (veja a [Tabela 12.1](#)); os demais não são considerados.

Suponha que a instrução está armazenada na memória da seguinte forma:

01	01110000
02	.....
03	.....
04	.....
.....	.....
.....	.....

Isso significa que a instrução está armazenada no endereço 1.

Esquematicamente, teremos:

## Fetch Da Instrução

$REM \leftarrow PC$	; REM recebe o endereço 1 para leitura.
read	; a UC gera o sinal read para a memória.
$RDM \leftarrow [REM]$	; o conteúdo do endereço 1 é lido e enviado pela memória para o RDM. Portanto, RDM recebe 01110000.
$PC \leftarrow PC + 1$	; o PC é atualizado com o endereço da próxima instrução, neste caso, 2.

## Decodificação

$RI \leftarrow RDM$	; em RDM temos a instrução e armazenamos em RI para a decodificação.
---------------------	--

Na decodificação, a UC verifica que o *opcode* 0111 (os quatro bits mais significativos) é da instrução retorno de sub-rotina, ou seja, uma instrução que lê da memória, no topo da pilha, o novo valor de PC. Depois da leitura é necessário somar 1 ao SP, para que ele aponte para o novo topo.

## Busca De Operando

Neste caso, como a instrução é do formato I, não há busca de operando.

## Execução

```
REM ← SP ; é armazenado em REM, pois será feita  
          uma leitura da memória.  
SP ← SP+1 ; o endereço do topo da pilha é  
          atualizado.
```

```
read      ; a UC gera o sinal read para a memória.  
RDM ← [REM] ; a leitura é feita.  
PC ← RDM ; o valor lido é armazenado no PC.
```

### A linguagem de montagem – *assembly*

Como já vimos, a linguagem de montagem é o resultado da tradução de uma linguagem de alto nível, feita antes da geração do código executável (linguagem de máquina). Cada instrução em linguagem de montagem corresponde a apenas uma instrução em linguagem de máquina, e cada instrução em linguagem de máquina corresponde a um ou a mais bytes.

Como a programação em linguagem de máquina é tediosa e suscetível a erros, pois só trabalhamos com 0's e 1's, fica mais fácil trabalhar com a linguagem de montagem.

Poderíamos nos questionar sobre o porquê de utilizar a linguagem de montagem se temos as linguagens de alto nível. Algumas razões para isso são:

- A possibilidade de desenvolver rotinas mais eficientes que as traduzidas pelos compiladores.
- Uma maior simplicidade na execução de operações de escrita e leitura em posições específicas de memória e em portas de E/S.
- A possibilidade de reescrever as partes críticas de um programa em linguagem de montagem para torná-las mais eficientes.
- Por meio da linguagem de montagem, há uma maior compreensão da maneira como o computador executa suas tarefas.

## **Linguagem de montagem: repertório de instruções**

As instruções de uma Linguagem de Montagem podem ser agrupadas em:

- **Instruções de entrada e saída:** são as instruções de leitura e escrita, tanto em memória como em unidades de E/S, que em alguns processadores são classificadas como instruções de movimentação de dados.
- **Instruções de movimentação de dados:** são instruções que permitem que os dados possam ser copiados de um local para outro, ou simplesmente trocados de local.
- **Instruções aritméticas e lógicas:** são as instruções que permitem que as operações aritméticas (soma, subtração, multiplicação e divisão) e lógicas (AND, OR, NOT e XOR) sejam executadas.
- **Instruções de desvios:** são instruções que permitem a construção de estruturas comumente utilizadas em linguagem de alto nível, como os *loops*, os desvios incondicionais, os desvios condicionais (se maior, se menor, se igual, se maior ou igual, se menor ou igual, e outros) e as chamadas de procedimentos.
- **Instruções de deslocamento e rotação:** são instruções que permitem a manipulação dos bits de uma palavra de dados.

### **Observações:**

1. Além das instruções, temos também as pseudoinstruções ou macros, que são implementações simples de funções específicas. No processo de montagem, estas pseudoinstruções são traduzidas pelo conjunto de instruções que a compõe.
2. As diretivas são instruções para os montadores, e não geram linguagem de máquina para o processador.

## **Elementos básicos da linguagem de montagem.**

Os elementos necessários para escrever um programa em linguagem de montagem são:

- **Rótulo (Label):** é um símbolo necessário para a identificação de endereços no código fonte, usado principalmente para saltos. Devem ser alfanuméricos e começar por letras.
- **Mnemônico:** é o símbolo que especifica o tipo de instrução.
- **Operandos:** são os registradores, imediatos ou endereços que a instrução irá

manipular.

■ **Comentário:** forma de documentar a natureza da ideia codificada.

Exemplo de linhas de programa *assembly* do 80x86:

[Rótulo]	[Mnemônico]	[Operando(s)]	[Comentário]
salto:	MOV	AX, 25	; inicializa AX com 25h
	ADD	AX, AX	; AX <-- AX + AX
	DEC	cont	; cont é uma variável
			; cont ← cont - 1
	JNZ	salto	; se cont é diferente de zero, salta para o rótulo salto



## O Que vem Depois

Agora já sabemos como uma instrução em *assembly* é executada pela UCP e quais são os elementos básicos de uma linguagem de montagem; portanto, já estamos preparados para aplicar estes conhecimentos em dois estudos de caso. O primeiro considerará um processador da INTEL 8086, e o segundo, um microcontrolador da MICROCHIP, o PIC 16F84.

---

## CAÍTULO 13

# O processador Intel® X86

---

### Objetivos do capítulo

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Trabalhar com processadores, utilizando sua linguagem de montagem.
- Desenvolver rotinas otimizadas e incluí-las em código escrito em linguagem de alto nível.
- Colocar em prática os conceitos vistos nos capítulos anteriores.
- Conhecer a família INTEL ×86.0



### Apresentação

Quando alguém vai comprar um computador, certamente se questiona se deve adquirir um com processador Intel ou outro qualquer.



Na maioria das vezes, escolhe-se um da família Intel, pelo fato de a marca ter maior penetração de mercado. Em virtude disto, utilizaremos para estudo de caso um dos processadores desta família.

# Fundamentos

## A família Intel® x86

Como vimos na Parte 3 do livro, podemos definir um microprocessador como um circuito integrado onde estão implementadas as funções de uma unidade central de processamento. São exemplos de microprocessadores a família Intel® x86, a família AMD, a família PowerPC, entre outros.

Utilizaremos um dos processadores da família Intel como objeto de estudo de caso.

A família x86 da Intel® é formada por microprocessadores (aos quais se convencionou chamar apenas de processadores) utilizados na maioria dos computadores pessoais desde a década de 1980. Os processadores 8086, 8088, 80186, 80188, 80286, 80386, 80486 e Pentium® pertencem a esta família.

Os processadores 8086 e 8088, introduzidos no final da década de 1970, foram os primeiros microcomputadores de 16 bits pela Intel®. A diferença entre estes dois processadores estava na largura do formato de dados: enquanto o 8086 trabalhava com dados de 16 bits, o 8088 trabalhava com dados de 8 bits, apesar de os dois serem iguais internamente.

Os processadores 80186 e 80188 eram versões melhoradas dos anteriores, e além de suportarem outras funções tinham a vantagem de executar um conjunto maior de instruções, chamada de *extended instruction set*.

Os processadores 80286, de 1982, são também processadores de 16 bits, mais rápidos que os anteriores, e com a vantagem de terem dois modos de operação: o modo de endereçamento real (como os anteriores) e o modo de endereçamento virtual ou endereçamento protegido, que dá suporte ao processamento multitarefa (no qual várias tarefas são executadas simultaneamente), protegendo a memória utilizada por um programa da ação de outro; no modo protegido, era possível endereçar um espaço maior de memória em comparação ao processador 8086.

Os processadores 80386 e o 80386SX, de 1985, foram os primeiros processadores de 32 bits da Intel®. Mais rápidos que os anteriores, trabalhavam com barramento de 32 bits e com um *clock* de frequência maior, possibilitando que a execução de uma instrução fosse feita mais rapidamente. Estes processadores também trabalhavam no modo real ou protegido; no modo protegido, podiam emular o 80286. Já no modo 8086 virtual, era possível

executar múltiplas aplicações compatíveis com o 8086. O 80386SX ainda tinha a mesma estrutura do processador 386, só que com barramento de dados de 16 bits.

Os processadores 80486 e 80486SX, de 1989, também eram processadores de 32 bits, e como era de se esperar, com maior capacidade de processamento que os anteriores. O 386 podia operar junto com um coprocessador aritmético (80387) para operações de ponto flutuante, enquanto as funções de ponto flutuante no 486 eram implementadas no próprio processador. O 486SX tinha a mesma estrutura do processador 486, mas sem as funções do processador de ponto flutuante.

Os processadores Pentium® (a raiz deste nome, *pent-*, significa “cinco” em grego, e Pentium significaria algo como “o quinto”), de 1993, pertencem à geração que sucedeu os 486, tanto que inicialmente seriam chamados de 586. A escolha do nome foi apenas uma questão de possibilidade de registro: como números não poderiam ser usados para registrar a marca, eles utilizaram o nome Pentium®. Apesar de serem processadores CISC, os processadores Pentium® eram superescalares, ou seja, eram processadores com duplicidades nos caminhos de execução, possibilitando que em um ciclo de clock mais de uma instrução fossem executadas. O Pentium®, um processador de 32 bits, suportava também um barramento de dados externo de 64 bits; com isso, a leitura da memória era feita com maior rapidez. Outra importante característica introduzida era o conjunto de instruções MMX (extensão multimídia – **MultiMedia eXtension**), com as quais era possível lidar com múltiplos dados (SIMD – *Single Instruction, multiple Data*), imprescindível para aplicações multimídia. Já o processador Pentium® Pro, de 1995, foi o primeiro processador da Intel® a ter um núcleo RISC (**Reduced Instruction Set Computer**), o que melhorou o seu desempenho em comparação aos processadores Pentium® tradicionais. Além disso, a memória *cache* nível 2 era integrada no próprio processador.

Os processadores Intel® atuais são *multicore*, ou seja, têm mais de um núcleo processador integrado em um mesmo circuito integrado. O primeiro processador *multicore* Intel® surgiu em 2005. O Pentium® D, um DualCore (dois *cores*, ou dois núcleos processadores em um único circuito), deu início à geração dos processadores Intel® Core™2 Duo. Em 2007, com o avanço da tecnologia *multicore*, surgiram os processadores de quatro núcleos, os chamados Intel® Core™2 Quad.

Os processadores Celeron® são processadores de baixo custo, baseados nos processadores Pentium® e Core™2 Duo. São processadores de menor custo e

consequentemente de menor desempenho, pois o tamanho de sua memória *cache* nível 2 é menor, além de trabalharem em uma frequência de *clock* interna também menor em relação aos processadores Intel® Core™2 Duo.

Os processadores Intel® Xeon® são processadores de alto desempenho específicos para servidores. Eles têm mais níveis *cache* e suportam o multiprocessamento (várias execuções de processos de maneira simultânea) quando se lida com projetos que utilizam mais de um processador na placa-mãe.

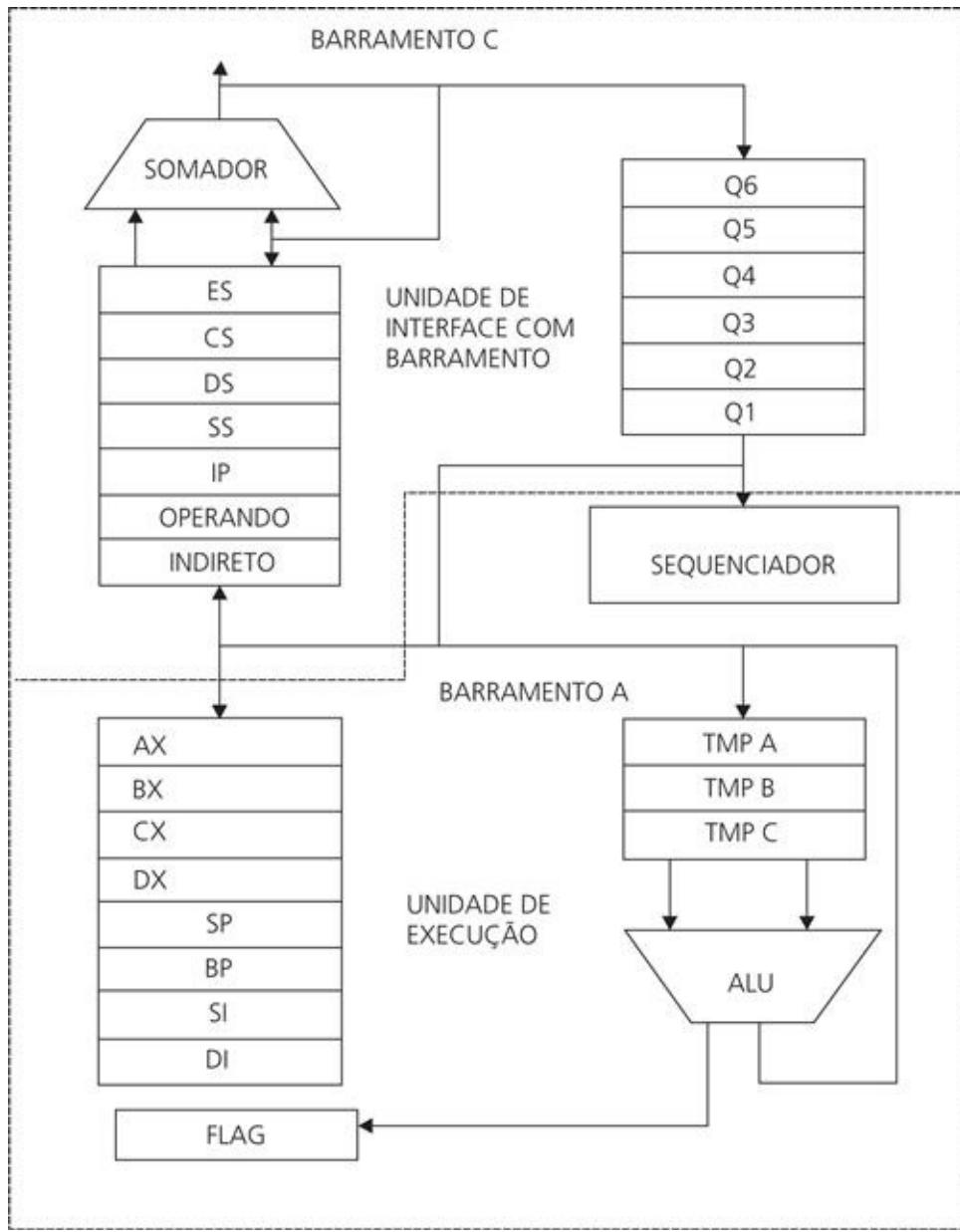
Os processadores Intel® Atom™ são processadores projetados para serem utilizados em dispositivos portáteis que necessitam ter um baixo consumo de energia, tais como computadores *netbooks*, *smartphones*, entre outros.

## O Processador 8086

### A arquitetura do processador 8086

O processador 8086 é um processador com uma estrutura e um conjunto de instruções simples, facilitando o aprendizado da programação em linguagem de montagem. Este aprendizado será a base para estudarmos não só os processadores da família ×86 como os outros processadores também.

Como vimos anteriormente, para programar em linguagem de montagem é necessário conhecer não só o conjunto de instruções do processador como também sua organização. A [Figura 13.1](#) mostra a organização do processador 8086.



**FIGURA 13.1** Organização do processador 8086. Adaptado do texto *Intel Microprocessors Documentation*. Disponível em: <http://intel.com>

## Unidades do processador 8086

A Figura 13.1 mostra que o processador 8086 divide-se internamente em duas unidades: a unidade de execução e a unidade de interface com o barramento.

### A Unidade De Execução (Execution Unit – EU)

Esta unidade é responsável pela execução das instruções. Nela encontramos a unidade lógica e aritmética (ULA), que executa as operações aritméticas (soma, subtração, divisão e multiplicação) e as operações lógicas (AND, OR, NOT e XOR).

## A Unidade De Interface Com O Barramento (BUS Interface Unit – BIU)

Esta unidade é responsável pela comunicação de dados entre a unidade de execução e o meio externo (memória, unidade de E/S). Ela também controla a transmissão de sinais de endereços, dados e controle, e com isso a sequência de busca de instruções. No processador 8086, em vez de se realizar o fetch de uma instrução somente após a execução da anterior, lê-se até seis instruções, armazenando-as na fila de instruções (*instruction queue*), com o objetivo de aumentar a velocidade de execução de um programa. Chamamos esta leitura de *pre-fetch*.

## Registradores do processador 8086

Os registradores do processador 8086 são divididos em registradores de propósito geral (ou de dados) e registradores específicos, como vimos no [Capítulo 11](#) deste livro. Os registradores específicos armazenam endereços (segmentos, apontadores e índices) e sinalizadores de estado e controle (*flags*).

### Registradores De Propósito Geral (Ou De Dados)

O 8086 tem quatro registradores de 16 bits que também podem ser usados como registradores de 8 bits. São os registradores AX, BX, CX e DX, utilizados nas operações aritméticas e lógicas. Quando utilizados como registradores de 8 bits, eles passam a ter em seu nome as letras L, para identificar o byte menos significativo ou inferior, e H para o byte mais significativo ou superior. Em outras palavras, se:

- AX = 4B3Ch → AH = 4Bh e AL = 3Ch
- BX = 3456h → BH = 34 h e BL = 56 h

- CX = 1267h → CH = 12 h e CL = 57 h
- DX = ABFFh → DH = ABh e DL = FFh

Apesar de serem registradores de propósito geral, eles também têm algumas funções especiais quanto registradores de dados, definidas pelas instruções:

- *Registrador AX (Acumulador)*: é utilizado como acumulador em operações aritméticas e lógicas, em instruções de E/S e em instruções de operações de ajuste BCD.
- *Registrador BX (base)*: é usado como registrador base para referenciar posições de memória, ou seja, armazena o endereço base de uma tabela ou vetor de dados, a partir do qual as posições são obtidas adicionando-se um valor de deslocamento (*offset*).
- *Registrador CX (contador)*: é utilizado em operações iterativas e repetitivas para “contar” bits, bytes ou palavras, podendo ser incrementado ou decrementado. A parte menos significativa de CX, o CL, funciona como um contador de 8 bits.
- *Registrador DX (dados)*: é utilizado em operações de multiplicação para armazenar parte de um produto de 32 bits, ou em operações de divisão de 32 bits, para armazenar o resto. Também é utilizado em operações de E/S para especificar o endereço de uma porta de E/S.

Quando tratarmos das instruções especificamente, mostraremos exemplos de utilização dos registradores nessas funções.

## Registradores Com Funções Específicas: Os Registradores De Segmentos

Como o 8086 trabalha com blocos de 64 Kbytes de memória, ele necessita de registradores para armazenar o endereço base desses segmentos: são os registradores de segmentos. Há quatro tipos de segmento: segmento de código, segmento de dados, segmento de pilha e segmento extra (este último, um segundo segmento de dados).

Os registradores de segmento do 8086, todos de 16 bits, são:

- *Registrador de segmento de código – CS (Code Segment);*
- *Registrador de segmento de dados – DS (Data Segment);*

- Registrador de segmento de pilha – SS (Stack Segment);
- Registrador de segmento extra de dados – ES (Extra Segment).

O endereçamento no 8086 apresenta diferenças quando consideramos individualmente o código de programa (instruções), os dados armazenados na memória principal e os dados armazenados na pilha. Cada um desses tipos é alocado em segmentos (blocos de memória de 64 Kbytes, para o 8086) endereçáveis. Durante a execução de um programa no 8086, há quatro segmentos ativos:

- Segmento de código endereçado por CS.
- Segmento de dados endereçado por DS.
- Segmento de pilha endereçado por SS.
- Segmento extra (de dados) endereçado por ES.

Como os registradores anteriormente mencionados armazenam o endereço base de seu respectivo segmento, necessitamos de outros registradores para armazenar o deslocamento (*offset*) dentro de um dado segmento ([Figura 13.2](#)).



**FIGURA 13.2** Segmento: endereço base e deslocamento (*offset*).

Esses registradores, também de 16 bits, são:

- **Registrador IP (Instruction Pointer):** é o PC (*program counter*) do Intel x86. É utilizado, em conjunto com o registrador CS, para localizar a posição, dentro do segmento de código corrente, da próxima instrução a ser executada. Ele é automaticamente incrementado em função do número de bytes da instrução executada, após o fetch desta instrução.
- **Registrador SP (Stack Pointer):** é utilizado, em conjunto com o registrador

SS, para acessar a área de pilha na memória. Ele sempre aponta para o topo da pilha.

- **Registrador BP (Base Pointer):** um ponteiro que, em conjunto com o registrador SS, permite acesso aos dados que estão dentro do segmento de pilha, como se fosse um vetor de dados.
- **Registrador SI (Source Index):** usado como registrador de índice em alguns modos de endereçamento indireto, em conjunto com o registrador DS. É utilizado, por exemplo, em instruções específicas de manipulação de *string* ou em acesso a vetores armazenados no segmento de dados. Ele pode também ser usado como registrador de dados de 16 bits.
- **Registrador DI (Destination Index):** similar ao SI, este registrador atua em conjunto com ES, quando utilizado em instruções de manipulação de *string*, ou com DS, em acesso a vetores armazenados no segmento de dados. Como o SI, pode também ser usado como registrador geral de dados de 16 bits.
- **Registrador de sinalizadores (flags):** é um registrador de 16 bits, que tem a função de indicar o estado do processador após a execução de uma instrução. Os bits subdividem-se em dois grupos: 6 bits para os *flags* de estado (*status*) e 3 bits para os *flags* de controle. Os demais bits não são utilizados e servem para “futuras implementações”. A seguir faremos um estudo detalhado sobre a função de cada bit de estado e veremos como utilizá-los.

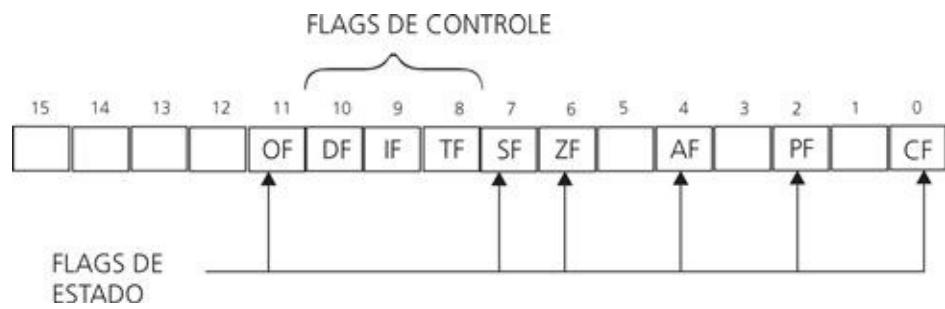
## O Status Do Processador E Os Sinalizadores (Flags)

Uma importante característica que distingue um computador de outras máquinas é a habilidade de tomar decisões. Os circuitos da UCP podem realizar decisões simples, baseadas no estado atual dos processadores; especificamente no processador 8086, este estado é verificado por nove bits individuais chamados de sinalizadores ou *flags*. Cada decisão tomada pelo 8086 é baseada nos valores destes *flags*.

Os *flags* estão localizados no registrador sinalizador e são classificados em *flags de status* e *flags de controle*.

- Os *flags de status* refletem o resultado de uma operação aritmética ou lógica.
- Os *flags de controle* são utilizados para habilitar e desabilitar certas operações do processador.

Na [Figura 13.3](#) temos o registrador *de flags*, com sua organização:



**FIGURA 13.3** O registrador *Flag*.

A [Tabela 13.1](#) mostra os *flags* e seus significados.

---

**Tabela 13.1**  
**Flags do processador 8086**

---

<b>Flags de estado</b>	
CF – flag de carry	Indica a ocorrência de <i>carry out</i> após a execução de uma instrução aritmética. Se CF = 1 ocorreu o “vai um” (adição) ou o “emprestimo” (subtração). Se CF = 0, tal não ocorreu.
PF – flag de paridade	Caso o byte inferior do resultado de alguma operação aritmética ou lógica apresente um número par de “1’s”, PF = 1 (paridade par); caso contrário, PF = 0 (paridade ímpar).
AF – flag de carry auxiliar – operações BCD	Se AF = 1 existiu o “vai um” do bit 3 para o bit 4 de uma adição de números BCD, ou caso exista “emprestimo” do bit 4 para o bit 3 em uma subtração de números BCD. Se AF = 0, tal não ocorreu.
ZF – flag zero	Caso o resultado da última operação aritmética ou lógica seja igual a zero, ZF = 1. Caso contrário, ZF = 0.
SF – flag de sinal	Utilizado para indicar se o resultado é positivo (SF = 1) ou negativo (SF = 0). Usado também para indicar a ocorrência de <i>overflow</i> em operações em complemento de 2.
OF – flag de overflow	Utilizado para indicar a ocorrência de <i>overflow</i> em operações de números sinalizados: se OF = 1, ocorreu <i>overflow</i> , e, se não, OF = 0.
<b>Flags de controle</b>	
TF – flag de trap (do inglês <i>trap</i> , armadilha)	Quando forçamos TF = 1, após a execução da próxima instrução, ocorrerá uma interrupção na execução do programa. A própria interrupção faz TF = 0. Se TF = 0, a execução será normal. Utilizada nos <i>debuggers</i> (programas que auxiliam a correção de erros).
IF – flag de interrupção	A ocorrência de interrupções ocorre quando IF = 1. IF = 0 desabilita a ocorrência de interrupções.
DF – flag de direção	Usado para indicar a direção em que as operações com <i>strings</i> são realizadas. Se DF = 1, o endereço de memória é decrementado, e se DF = 0, ele é incrementado.

## Overflow

Para a verificação da ocorrência de erro de *overflow*, temos que saber se a operação foi feita com número sinalizado ou não sinalizado.

## Overflow Decorrente De Operações Com Números Não Sinalizados

Nas operações de adição, um *overflow* não sinalizado ocorre quando há vai-um no MSB. Isto significa que a resposta (resultado) é maior do que o tamanho do número, ou seja, maior que FFFFh para uma palavra e maior que FFh para um byte.

Nas operações de subtração, um *overflow* não sinalizado ocorre quando há empresta-um no MSB. Isto significa que a resposta (resultado) é menor que 0 (zero). Em ambos os casos, basta verificar o *flag* CF.

## Overflow Decorrente De Operações Com Números Sinalizados

Na adição de números com o mesmo sinal, um *overflow* sinalizado ocorre quando o resultado da soma tem um sinal diferente. Como um exemplo, podemos ver que isto acontece quando estamos somando dois números positivos e o resultado é um número negativo (ex.: 7Fh + 7Fh = FEh).

**Observação:** A subtração de números com sinais diferentes é como a adição de números com o mesmo sinal. Por exemplo:

$$A - (-B) = A + B \text{ e } -A - (+B) = -A + (-B)$$

Na adição de números com sinais diferentes, *overflow* é impossível, pois a soma torna-se uma subtração: por exemplo,  $A + (-B) = A - B$ . Como A e B podem ser representados pelo número de bits dos registradores, o resultado da diferença entre eles também poderá ser. Analogamente, subtração de números com o mesmo sinal não pode resultar em *overflow*.

Por isso, para verificarmos a ocorrência de *overflow* em operações sinalizadas, temos de verificar os *flags* CF e OF. O *flag* CF indica se há um vai-um na posição do MSB, e o *flag* OF verifica o vem-um que chega e o vai-um gerado no MSB. Se os dois forem iguais (0 e 0 ou 1 e 1), teremos  $OF = 0$ , e se forem diferentes, teremos  $OF = 1$ .

## **Exemplo 1**

ADD AL, BL ; AL $\leftarrow$ AL + BL e AL contém FFh e BL contém 01h			
	representação não sinalizada		representação sinalizada
FFh	1111 1111b	255	-1
01h	+ 0000 0001b	+ 1	+ 1
	<b>1</b> 0000 0000b	<u>256</u> (fora da faixa)	<u>0</u> (OK)

Logo após a execução da instrução:

CF = 1 e OF = 0, pois no MSB o vem-um é igual ao vai-um(ambos 1). Neste caso temos um *overflow* sinalizado, e não um *overflow* não sinalizado.

## **Exemplo 2**

Suponha que a instrução a seguir faça a soma de AL com BL (veremos as instruções aritméticas no próximo capítulo).

ADD AL, BL ; AL  $\leftarrow$  AL + BL e ambos AL e BL contém 7Fh

	representação não sinalizada	representação sinalizada
7Fh	0111 1111b	127
7Fh	+ 0111 1111b	+ 127
	<u>0 1111 1110b</u>	<u>+ 127</u>
	254 (OK)	+ 254 (fora)

Logo após a execução da instrução:

CF = 0 e OF = 1, pois no MSB o vem-um é diferente do vai-um.

## Resumo Dos Registradores Do Processador 8086

Na [Tabela 13.2](#) encontramos um resumo dos registradores do processador 8086.

---

### Tabela 13.2

#### Resumo dos registradores do processador 8086

---

Registradores de dados			
AH	AL	→	AX
BH	BL	→	BX
CH	CL	→	CX
DH	DL	→	DX

Registradores de segmentos			
	CS		
	DS		
	SS		
	ES		

Registradores índices e apontadores			
	SI		
	DI		
	SP		
	BP		
	IP		

Registrador de sinalizadores			
	<i>Flags</i>		

## Barramentos do processador 8086

Como vimos no [Capítulo 11](#) deste livro, os barramentos são vias que permitem a movimentação de informações entre diversas unidades. O processador 8086 tem um único barramento que trabalha como barramento de endereços e de dados, ou seja, tem um barramento multiplexado. Sua largura é de 20 bits, por isso acessa até 1MByte de memória ( $2^{20} = 1.048.576 = 1M$ ). Quando utilizado como barramento de dados, só utilizam 16 bits dos 20 bits disponíveis. O barramento de controle é independente e possui 16 bits.

## Organização de memória

Como acabamos de ver, o processador 8086 possui um barramento que usa 20 bits para acessar posições de memória física, quando é utilizado para endereços, ou seja, pode acessar memórias de até 1MB, do endereço 00000 h até FFFFFh.

Apesar de ter um barramento de 20 bits, o 8086 trabalha com palavras de endereço de 16 bits. Então, como gerar endereços com 20 bits se o processador trabalha com palavras de 16 bits? A solução é simples: basta utilizar a ideia de segmentação de memória. Um segmento de memória, aqui, é um bloco de 64 KB ( $2^{16} = 2^6 \times 2^{10} = 65.536 = 64K$ ) de posições de memória consecutivas, identificadas por um endereço de segmento (endereço base) e um deslocamento (*offset*) em relação ao início do segmento. Em outras palavras, este processador pode utilizar 64K segmentos de memória (“memória lógica”) mapeados em uma memória de 1MB (“memória física”). O par *segment:offset* é o endereço lógico que nos dá a posição que queremos acessar dentro do segmento apontado por *segment*, com um deslocamento dado por *offset* ([Figura 13.2](#)).

Considere o endereço lógico 8350 h:0420 h; temos aí um endereço do segmento que se inicia em 8350 h, com deslocamento de 0420 h.

Para que seja acessado na memória física, o endereço lógico tem de ser transformado em endereço físico. Todo início de segmento, os quatro bits menos significativos do endereço físico são iguais a zero. Portanto para transformar um endereço lógico em endereço físico, basta acrescentar 0 h (0000b) no final do *segment* (deslocamento de quatro bits para a esquerda) e somar com o *offset*. O endereço lógico 8350 h:0420 h, por exemplo, corresponderá à localização na memória física 83920 h, pois:

83500h	→ desloca-se uma casa hexadecimal (quatro casas binárias)
+ 0420h	→ soma-se o deslocamento
<u>83920h</u>	→ endereço físico resultante (20 bits)

O identificador de segmento (*segment*) é o endereço base. Neste exemplo, ele

equivale a 8350 h, e aponta para uma região da memória. Já o *offset* aponta para um local dentro deste segmento, e equivale a 0420 h, neste exemplo.

Utilizando a segmentação, temos que endereços lógicos diferentes podem representar o mesmo endereço físico. Por exemplo os endereços lógicos 028Ch:0003 h e 0287 h:0053 h representam o mesmo endereço físico 028C3 h.

Na programação do 8086, veremos que o endereço base do *segment* sempre tem de estar armazenado no registrador de segmento (CS, DS, SS ou ES), e o *offset*, ou nos registradores de propósito geral ou nos específicos (IP, SP ou BP), ou ainda indicado pelos rótulos. Se quisermos acessar uma posição no segmento de código, por exemplo, o CS tem de ser iniciado com o endereço base do segmento e o IP com o respectivo offset, ou seja, se CS = 0F0Fh e IP = 000Fh, o endereço real será 0F0FFh.

**Observação:** A segmentação é um esquema muito útil para gerar códigos realocáveis, ou seja, códigos executáveis que podem ser alocados em qualquer posição de memória.

## **Interrupções**

Uma interrupção é um evento ocasional que deve ser prontamente atendido durante a execução de um processamento pelo computador. O atendimento desta interrupção causa a suspensão do processamento em curso.

No 8086, temos as seguintes causas de interrupções:

- Pela ocorrência de eventos “catastróficos”: falta de energia, erro de memória, erro de paridade em comunicações etc. Este tipo de interrupção não pode ser inibido.
- Pela ação de dispositivos externos (periféricos): entrada e saída por teclado, mouse etc. Estas interrupções podem ser habilitadas ou inibidas.
- Pelo próprio programa em curso: erro de divisão, erro de transbordamento (*overflow*). Este tipo de interrupção é chamado de exceção.

## **Portas de entrada e saída**

O processador 8086 usa até 64 KB para endereçar as portas de E/S. A [Tabela 13.3](#) nos mostra as portas de E/S mais utilizadas, as quais podem ser endereçadas por meio de instruções de entrada e de saída ou por chamadas a rotina do sistema operacional.

---

### **Tabela 13.3**

## Portas de E/S

---

Endereço da Porta	Descrição
20 h-21 h	Controlador de interrupção
60 h-63 h	Controlador de teclado
2F8 h-2FFh	Porta serial – COM 2
320 h-32Fh	Disco rígido
378 h-37Fh	Porta paralela
3F8 h-3FFh	Porta serial – COM 2



### Ovem Depois

Apresentamos neste capítulo um pouco da família ×86, das origens até os dias de hoje, e introduzimos o processador 8086, por este ser o processador ideal para o aprendizado de programação em linguagem de montagem, o que ajudou também na compreensão do funcionamento de um processador. Utilizaremos este processador para o aprendizado da programação em linguagem de montagem.

---

## CAPÍTULO 14

---

# Introdução à programação em linguagem de montagem do Intel 8086

---

### Objetivos do capítulo

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Utilizar o programa DEBUG para execução de pequenos trechos de código *assembly*.
- Entender a estrutura dos programas em linguagem *assembly*.
- Conhecer e utilizar o formato dos dados, variáveis e constantes na linguagem *assembly*.
- Conhecer e utilizar os comandos de entrada e saída e os sinalizadores (*flags*).
- Criar e executar programas em linguagem *assembly*.



### Apresentação



Você já deve ter ouvido e sabe muito bem que, para iniciar uma jornada, seja

ela longa ou não, tudo o que precisa fazer é dar o primeiro passo. Podemos dizer que para dominar os princípios básicos da linguagem *assembly* Intel x86, a compreensão dos conceitos tratados neste capítulo seja este primeiro passo.

Iniciaremos com uma explanação sobre como podemos interagir diretamente com áreas da memória, carregarmos informações e comandos, executarmos e obtermos os resultados de maneira bem interativa e ilustrativa utilizando um programa simples (mas muito poderoso) conhecido como DEBUG.

Após esse entendimento, passaremos para o reconhecimento dos tipos de dados, variáveis e constantes na linguagem *assembly*, e veremos também os principais comandos de entrada e saída. Terminaremos o capítulo mostrando o que são e como podemos utilizar os sinalizadores (ou flags).

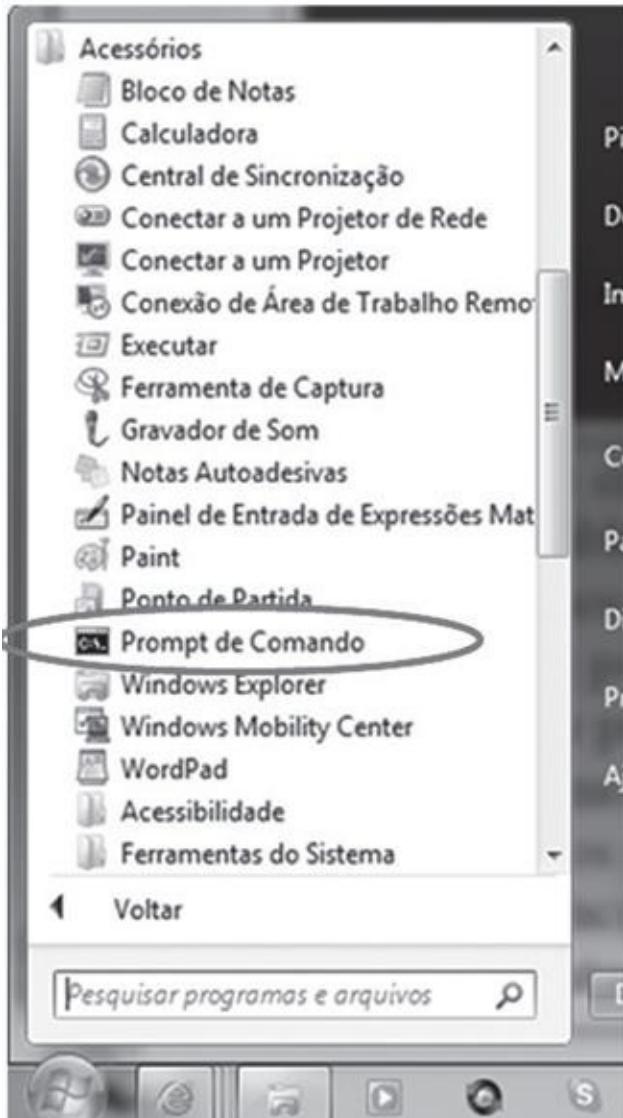
Vamos lá! Lembre-se que esse é só o primeiro passo!

## Fundamentos

### ***Programando com o programa DEBUG***

Antes de iniciarmos, é importante explicar a razão de começar os conceitos básicos da linguagem de montagem com um programa de depuração de código (DEBUG). Trata-se de uma estratégia de ensino, prática e visual, para mostrar como as informações e os comandos são armazenados na memória do computador e em qual ordem são agrupadas e executadas/lidas. Após entender e visualizar a apresentação das informações e comandos utilizando o programa DEBUG, geralmente os estudantes não apresentam maiores dificuldades na abstração de comandos e operações mais complexas. Isto vem facilitando muito o entendimento em nossas aulas práticas nas disciplinas iniciais de programação *assembly*. Vamos ao DEBUG!

Para executar o programa em computadores com o sistema operacional Windows instalado, basta executar o “*Prompt de Comando*”, localizado geralmente no botão Iniciar → Acessórios. A [Figura 14.1](#) ilustra um possível local para o “*prompt de comando*” no sistema operacional Windows 7.

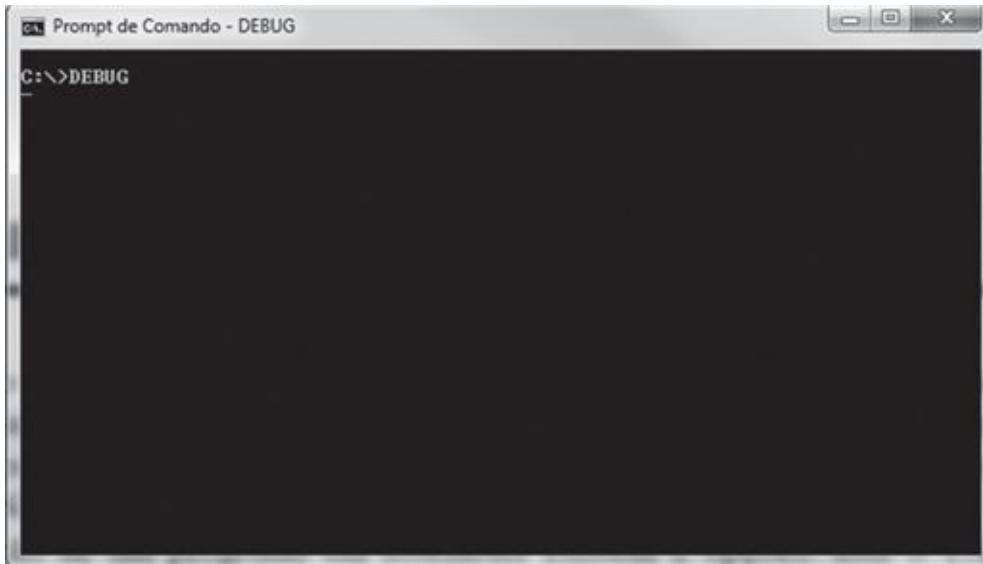


**FIGURA 14.1** Localização do “*Prompt de Comando*” no sistema Windows 7.

Uma vez executado o “*Prompt de Comando*”, digite na linha de comando o seguinte comando, finalizando com a tecla “*enter*” (<enter>).

```
C:\> DEBUG <Enter>
```

A [Figura 14.2](#) ilustra o resultado desse comando.



**FIGURA 14.2** Resultado da Execução do Comando DEBUG no “Prompt de Comando”.

Se você foi atento, percebeu que abaixo da linha de comando C:\>DEBUG há um “hífen”. Isto significa que você executou com êxito o programa DEBUG. Para sair do DEBUG, basta informar na linha de comando (após o hífen) a letra “Q” (*quit*) para que o programa retorne à linha de comando do “*Prompt de Comando*”.

Dentro do programa DEBUG, você pode verificar todos os comandos que podem ser executados. Basta digitar “?” seguido de <enter>.

Após digitar “?” será mostrada a Lista de Comandos do programa DEBUG, como ilustrado na [Figura 14.3](#), a seguir.

```
C:\>DEBUG
?
assemble      A [address]
compare       C range address
dump          D [range]
enter         E address [list]
fill          F range list
go            G [=address] [addresses]
hex           H value1 value2
input          I port
load          L [address] [drive] [firstsector] [number]
move          M range address
name          N [pathname] [arglist]
output         O port byte
proceed        P [=address] [number]
quit          Q
register       R [register]
search         S range list
trace          T [=address] [value]
unassemble    U [range]
write          V [address] [drive] [firstsector] [number]
allocate expanded memory   XA [#pages]
deallocate expanded memory  XD [handle]
map expanded memory pages  XM [Lpage] [Ppage] [handle]
display expanded memory status XS
```

**FIGURA 14.3** Lista de Comandos do programa DEBUG.

Para iniciar, vamos começar com algo bem simples, e aproveitar para fazer uma revisão de aritmética binária!

Utilizemos o Comando H, que faz cálculos de adição e subtração em hexadecimal (hex).

#### Para Adição e Subtração: Comando H

Supondo dois números em hex: 0002 e 0001, escreva no programa DEBUG o seguinte comando, terminando com a tecla <enter>:

```
H 0002 0001 <Enter>
```

Após a execução, o resultado será:

```
-H 0002 0001    < Enter >  
0003 0001
```

O primeiro valor exibido é o resultado da adição dos dois números fornecidos como parâmetro do comando H, e o segundo é o resultado da subtração destes mesmos dois números.

Considere agora os valores: **0009** e **0001**.

Resultado:

```
000A 0008
```

Note que o resultado é apresentado em hexadecimal. Porém, a resposta a qual estamos acostumados para a soma de 9 + 1 é 10. Lembre-se que  $9A_{16} = 10_{10}$

Suponha agora outros valores: **B000** e **A000**

Resultado:

```
5000 1000
```

Note que aqui também tem alguma coisa errada. Agora, segundo a aritmética hexadecimal, o primeiro valor, resultado da adição dos dois valores, deveria ser 15000, porém, o DEBUG só apresenta os quatro valores à esquerda. A [Figura 14.4](#) apresenta os comandos e resultados descritos.

```
C:\>debug
-H 0002 0001
0003 0001
-H 0009 0001
000A 0000
-H B000 A000
5000 1000
```

**FIGURA 14.4** Comandos de adição e subtração em hexadecimal.

Não sabemos se você já tentou, mas no nosso primeiro comando utilizando o comando H, fizemos a adição e a subtração dos números 0002 e 0001 (nessa ordem). E se invertêssemos os números, o que poderia acontecer? Vamos tentar?

-H 0001 0002 <Enter>

Resultado:

0003 FFFF

O segundo valor não deveria ser: -0001 ???

Novamente, lembre-se que estamos trabalhando com números hexadecimais, e  $FFFF_{16}$  corresponde a  $-0001_{16}$  em complemento de 2.

Tente mais esse comando:

```
-H 0001 FFFF
```

Resultado:

```
0000 0002
```

Existe uma explicação bem fácil para isso: na realidade, os dois valores (0001 e FFFF) são somados. O resultado é  $0001 + FFFF = 10002$ , e aí ocorreu um overflow; o número 1 é desconsiderado neste caso.

Vamos partir para os cálculos e passar a utilizar os códigos de máquina da linguagem *assembly*. O primeiro passo é aprendermos a verificar os registradores e os valores neles armazenados.

Você pode verificar os registradores com o comando R.

Na linha de comando do programa DEBUG, digite:

```
-R
```

O resultado será algo semelhante às seguintes informações:

```
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000  
SI=0000 DI=0000  
DS=0B19 ES=0B19 SS=0B19 CS=0B19 IP=0100 NV UP
```

```
EI PL NZ NA PO NC
```

0B19:0100 7438 JZ 013A

Podemos ver que os registradores AX, BX, CX e DX estão zerados.

Para visualizarmos o conteúdo de um registrador específico, podemos utilizar o mesmo comando R, seguido do nome do registrador. Se quisermos saber o conteúdo do registrador AX, por exemplo, podemos utilizar o comando RAX. Assim:

**-RAX <enter>**

O resultado seria:

AX 0000

:

Note que, após apresentar o conteúdo do registrador AX, foi exibido na linha seguinte o sinal de dois pontos (“:”).

Se quisermos inserir um valor diferente, basta digitar após os dois pontos o valor desejado e a tecla <enter>. Se não quisermos alterar, basta teclar <enter>. Vamos alterar o valor para 0001. Digite após os dois pontos 0001 e tecle <enter>.

Agora, se quisermos verificar se o novo valor foi inserido no registrador AX, podemos digitar o comando R. Assim, teríamos o seguinte resultado:

```
AX=0001  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  
SI=0000  DI=0000  
DS=0B19  ES=0B19  SS=0B19  CS=0B19  IP=0100  NV UP EI
```

```
PL NZ NA PO NC  
0B19:0100 7438 JZ 013A
```

Note que agora o registrador AX apresenta o valor 0001, como queríamos.

Poderíamos montar instruções e sequências de comando utilizando esse método, com o auxílio dos códigos de operação (*opcode* ou *operate code*). Entretanto, fazer um programa *assembly* seria muito trabalhoso e complexo.

Para agilizar um pouco mais o processo, podemos digitar os comandos diretamente na memória, em posições sequenciais, informando apenas o endereço de início do código.

Para tanto, temos de conhecer mais um comando do programa DEBUG. O comando A (*address* ou endereço). Digitamos A seguido do valor em hexadecimal, geralmente 0100 h, local onde os programas .COM devem iniciar. No caso de omissão, o endereço inicial é o especificado pelos registradores CS:IP.

Vamos digitar um pequeno programa que colocará o valor 0002 no registrador AX e o valor 0003 no registrador BX. Em seguida, os valores serão somados e o resultado será colocado em AX.

O programa em linguagem *assembly* é o seguinte:

```
MOV AX, 0002  
MOV BX, 0003  
ADD AX, BX
```

A sequência de comandos necessários para registrarmos esse pequeno

programa na memória é a seguinte:

```
A 100 <enter>
MOV AX, 0002 <enter>
MOV BX, 0003 <enter>
ADD AX, BX <enter>
NOP <enter> <enter>
```

A [Figura 14.5](#) a seguir ilustra a entrada de dados desse pequeno programa.



**FIGURA 14.5** Digitação de um pequeno programa utilizando o DEBUG.

O último comando, NOP (*No operation* ou nenhuma operação), é um comando que indica o final de uma sequência de comandos, ou seja, o final de um programa.

Para executar esse pequeno programa, utilizaremos outro comando do DEBUG: o comando **t** (*trace*). Ele executa comando a comando, mostrando o estado dos registrados ao final de cada execução.

Vamos lá!

-t (executa o comando MOV AX,0002)

```
AX=0002  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  
SI=0000  DI=0000  
DS=0B19  ES=0B19  SS=0B19  CS=0B19  IP=0103  NV  UP  EI
```

PL NZ NA PO NC  
0B19:0103 BB0300 MOV BX, 0003

-t (executa o comando MOV BX,0003)

```
AX=0002  BX=0003  CX=0000  DX=0000  SP=FFEE  BP=0000  
SI=0000  DI=0000  
DS=0B19  ES=0B19  SS=0B19  CS=0B19  IP=0106  NV  UP  EI
```

PL NZ NA PO NC  
0B19:0106 01D8 ADD AX,BX

-t (executa o comando ADD AX,BX)

```
AX=0005  BX=0003  CX=0000  DX=0000  SP=FFEE  BP=0000  
SI=0000  DI=0000  
DS=0B19  ES=0B19  SS=0B19  CS=0B19  IP=0108  NV  UP  EI
```

PL NZ NA PE NC

```
0B19:0108 90 NOP
```

-t (executa o comando NOP)

```
AX=0005 BX=0003 CX=0000 DX=0000 SP=FFEE BP=0000  
SI=0000 DI=0000  
DS=0B19 ES=0B19 SS=0B19 CS=0B19 IP=0109 NV UP EI
```

```
PL NZ NA PE NC  
0B19:0109 96 XCHG SI,AX
```

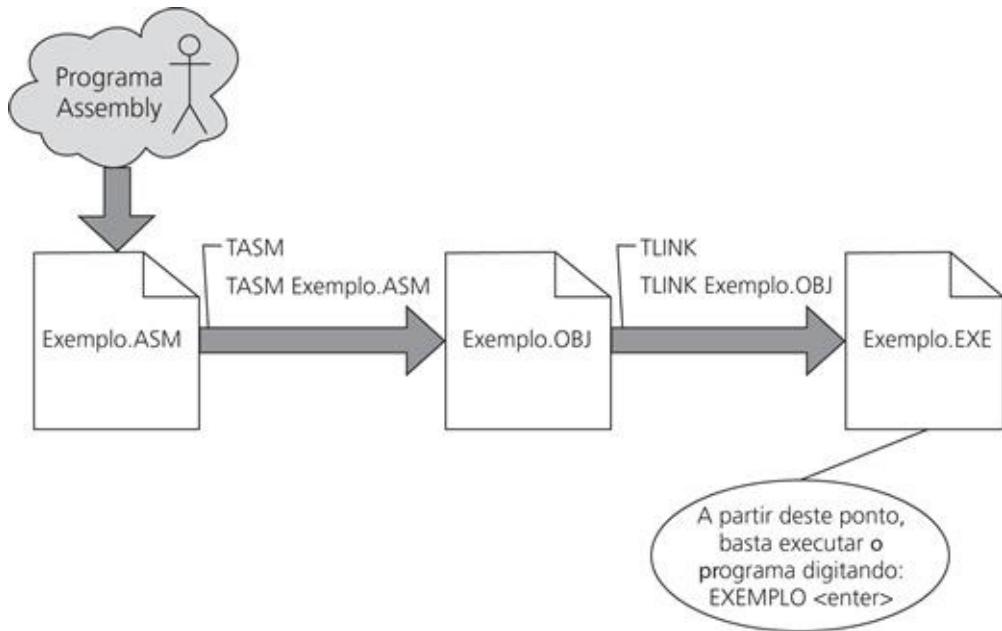
Com essa sequência, o programa foi executado. O resultado da adição encontra-se no registrador AX.

Fazer programas em linguagem de montagem não é uma tarefa muito simples, e utilizar o DEBUG para isso fica mais difícil ainda – mesmo utilizando o modo de introduzir comandos diretamente na memória.

Como dissemos no início deste capítulo, a utilização do programa DEBUG é uma estratégia didática para que o leitor perceba de forma prática e visual como as informações e os comandos são armazenados na memória, e assim possa fazer processos de abstração mais complexos de forma mais fácil.

### ***Programando com o montador TASM***

Ilustramos na [Figura 14.6](#) a maneira como faremos a programação em *assembly*:



**FIGURA 14.6** Processo de construção de programas assembly.

Editaremos a sequência de instruções em um editor de texto (um que não utilize códigos de controle, como o Bloco de Notas). Salvaremos o arquivo com um determinado nome de extensão **.ASM**.

Depois, faremos a compilação, utilizando o TASM (*Turbo Assembler – Borland*). Se algum erro for evidenciado, editaremos novamente o arquivo e faremos as devidas correções.

Ao chegarmos a um processo de compilação sem erros, o TASM gerará um arquivo objeto (com extensão **.OBJ**). Depois disso, faremos a ligação com o TLINK (*Turbo Linker – Borland*), e se tudo der certo teremos o arquivo executável (**.EXE** ou **.COM** – conforme configuração).

Para exemplificar tal processo, vamos escrever um programa simples, que escreverá uma mensagem na tela (“Linguagem de montagem é fácil!”).

Abra um editor de texto – o Bloco de Notas, por exemplo –, e digite o seguinte programa:

```

TITLE PGM1: MENSAGEM
• MODEL SMALL
• STACK 100H
• DATA
mensagem DB 'Linguagem de montagem eh facil!$'

```

```
• CODE
MAIN PROC
    MOV AX, @DATA
    MOV DS, AX
    LEA DX, mensagem
    MOV AH, 9
    INT 21H
    MOV AH, 4CH
    INT 21H
MAIN ENDP
END MAIN
```

Grave o arquivo como **MENSAGEM.ASM**

Na linha de comando do DOS (no mesmo diretório onde você salvou o arquivo) digite:

```
TASM MENSAGEM.ASM <enter>
```

Será apresentada a seguinte mensagem:

```
C:\Tasm>edit mensagem.asm
C:\Tasm>TASM MENSAGEM.ASM
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland
International
Assembling file: MENSAGEM.ASM
**Error** MENSAGEM.ASM(5) Extra characters on line
Error messages: 1
Warning messages: None
Passes: 1
Remaining memory: 452k
```

```
C:\Tasm>
```

Observe que no exemplo considerado ocorreu um erro no processo de compilação, ocorrido na linha 5 (conforme indicado na mensagem).

Basta editar o arquivo .ASM, corrigir o erro (colocando um sinal ‘no fim da linha) e compilar novamente o código.

Se tudo der certo, você obterá as seguintes mensagens:

```
C:\Tasm>TASM MENSAGEM.ASM~
```

```
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland  
International
```

Assembling file:	MENSAGEM.ASM
Error messages:	None
Warning messages:	None
Passes:	1
Remaining memory:	452k

```
C:\Tasm>
```

Agora basta fazer a ligação com o TLINK.

```
TLINK MENSAGEM.OBJ <enter>
```

Se tudo der certo, você terá a seguinte mensagem:

```
C:\Tasm>TLINK MENSAGEM.OBJ
Turbo Link Version 7.1.30.1. Copyright (c)1987, 1996 Borland
International
C:\TASM>
```

Para executar o programa, basta digitar o nome na linha de *prompt*.

```
C:\TASM>MENSAGEM
Linguagem de montagem eh facil!
C:\TASM>
```

Esses são os passos para gerar, montar e executar um programa em linguagem de montagem, utilizando as ferramentas da Borland.

Sem nos importarmos ainda com a estrutura e os comandos da linguagem *assembly*, vamos digitar o programa a seguir, o qual carregará dois registradores, AX e BX, com os valores 000B e 00A1, e fará a soma destes dois valores armazenando o resultado em AX. Logo em seguida, carregará o valor 0005 em BX e fará a subtração com o resultado obtido na operação anterior, armazenando o resultado em AX. Por fim, o programa encerrará sua execução, devolvendo o controle ao sistema operacional.

Programa 14.1

```
TITLE PRM141:ADICAO E SUBTRACAO
;Programa para ilustração do programa DEBUG
• MODEL SMALL
• STACK 100H
• CODE
MAIN PROC
```

```
MOV AX, 000BH ; carrega o primeiro valor em AX  
MOV BX, 00A1H ; carrega o segundo valor em BX  
ADD AX, BX ; soma os dois valores e armazena  
             ; o resultado em AX  
MOV BX, 0005H ; carrega o terceiro valor em BX  
SUB AX, BX ; subtrai o terceiro valor do  
             ; resultado
```

```
; da operação anterior  
MOV AH, 4CH ; código para devolver o  
             ; controle p/ DOS  
INT 21H ; interrupção que executa a  
             ; função em AH
```

```
MAIN ENDP  
END MAIN
```

Vamos agora gerar o código executável, realizando as mesmas etapas vistas no exemplo anterior.

*Montagem do programa para geração do .OBJ*

```
C:\TASM>TASM PRM141.ASM <enter>
```

```
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland  
International
```

Assembling file:	PRM141.ASM
Error messages:	None
Warning messages:	None
Passes:	1
Remaining memory:	413k

*Ligaçao para geração do .EXE*

```
C:\TASM>TLINK PRM141.OBJ <enter>
Turbo Link Version 7.1.30.1. Copyright (c) 1987, 1996 Borland
International
```

*Execução do programa*

```
C:\TASM>PRM141 <enter>
C:\TASM>_
```

Como não mandamos escrever o resultado, não sabemos se as operações foram executadas corretamente. Uma das maneiras de verificar se a execução foi correta é utilizar o programa DEBUG.

Para usar o DEBUG, vamos digitar o seguinte na linha de comando, logo após a execução:

```
C:\TASM>DEBUG PRM141.EXE
```

```
_
```

Utilizando os comandos do programa DEBUG que já vimos anteriormente, vamos verificar a situação dos registradores:

*Tecle R <enter>*

```
-R
```

```
AX=0000 BX=0000 CX=0011 DX=0000 SP=0100 BP=0000  
SI=0000 DI=0000  
DS=152B ES=152B SS=153D CS=153B IP=0000 NV UP EI
```

```
PL NZ NA PO NC  
153B:0000 B80B00 MOV AX, 000B
```

Como podemos notar, os registradores de dados estão “zerados”, e a próxima instrução a ser executada é a instrução `MOV AX,000B`, que é a primeira instrução do programa PRM141.

Ao executar a instrução corrente com o comando **T** (*trace*), temos:

```
-T
```

```
AX=000B BX=0000 CX=0011 DX=0000 SP=0100 BP=0000  
SI=0000 DI=0000  
DS=152B ES=152B SS=153D CS=153B IP=0003 NV UP EI  
PL NZ NA PO NC  
153B:0003 BBA100 MOV BX, 00A1
```

Note que, após a execução da instrução corrente, o registrador AX agora armazena o valor 000B como queríamos. Veja que a instrução corrente agora é `MOV BX,00A1`, a segunda instrução do programa PRM141.

Concluindo, ao executar instrução a instrução ou executar um determinado trecho de programa (até a posição final, com o comando G), podemos verificar que o programa está sendo executado adequadamente.

Para ilustrar o exemplo, a seguir mostraremos a execução do programa utilizando os comandos **T** e **G**:

```
-T
```

```
AX=000B  BX=00A1  CX=0011  DX=0000  SP=0100  BP=0000
SI=0000  DI=0000
DS=152B  ES=152B  SS=153D  CS=153B  IP=0006  NV  UP
```

```
EI PL NZ NA PO NC
153B:0006 03C3 ADD AX,BX
```

```
-T
```

```
AX=00AC  BX=00A1  CX=0011  DX=0000  SP=0100  BP=0000
SI=0000  DI=0000
DS=152B  ES=152B  SS=153D  CS=153B  IP=0008  NV  UP
```

```
EI PL NZ NA PE NC
153B:0008 BB0500 MOV BX,0005
```

```
-T
```

```
AX=00AC  BX=0005  CX=0011  DX=0000  SP=0100  BP=0000
SI=0000  DI=0000
DS=152B  ES=152B  SS=153D  CS=153B  IP=000B  NV  UP
```

```
EI PL NZ NA PE NC
153B:000B 2BC3 SUB AX,BX
```

```
-T
```

```
AX=00A7  BX=0005  CX=0011  DX=0000  SP=0100  BP=0000
SI=0000  DI=0000
DS=152B  ES=152B  SS=153D  CS=153B  IP=000D  NV  UP
```

```
EI PL NZ NA PO NC  
153B:000D B44C MOV AH, 4C  
-G  
Program terminated normally  
-
```

Mesmo que haja a possibilidade de verificar o que está ocorrendo no processador a cada instrução com o DEBUG, não se utiliza muito essa forma de trabalho, por sua visibilidade limitada.

Porém, existe um outro programa chamado TURBO DEBUGGER, da Borland, que permite a mesma verificação mas em um ambiente mais amigável.

Entretanto, para fazer essa verificação, no processo de montagem-ligaçāo é necessário dar informações (juntamente com o código) ao montador e ao ligador (*linker*) para que o programa TURBO DEBUGGER possa controlar a execuāo do programa.

Para isso, no processo de compilação, vamos utilizar a diretiva /zi, e no processo de ligaçāo, a diretiva /v.

Assim, a montagem e a ligaçāo do programa PRM141.ASM ficariam da seguinte forma:

```
C:\TASM>TASM /zi PRM141.ASM <enter>  
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland  
International
```

Assembling file:	PRM141.ASM
Error messages:	None
Warning messages:	None
Passes:	1
Remaining memory:	411k

```
C:\TASM>TLINK /v PRM141.OBJ <enter>  
Turbo Link Version 7.1.30.1. Copyright (c) 1987, 1996 Borland  
International
```

```
C:\TASM>
```

Dessa forma, vemos que o arquivo PRM141.EXE possui as informações que possibilitarão ao TD (TURBO DEBUGGER) gerenciar a execução do código.

Para executar o programa com o TURBO DEBUGGER, digite o seguinte na linha de comando do DOS:

```
TD <nome do programa.exe> <enter>
```

No nosso caso:

```
TD PRM141.EXE <enter>
```

Aparecerá na tela:

The screenshot shows a window titled "[■]-CPU 80486" with the following details:

- Menu Bar:** File, Edit, View, Run, Breakpoints, Data, Options, Window, Help, READY
- Registers:** ax=0000, bx=0000, cx=0000, dx=0000, si=0000, di=0000, bp=0000, sp=0100, ds=5C6D, es=5C6D, ss=5C7F, cs=5C7D, ip=0000
- Assembly Code:**

```

    - File Edit View Run Breakpoints Data Options Window Help
-[■]-CPU 80486
    cs:0000>B80B00    mov    ax,000B      ▲ ax 0000 |c=0|
    cs:0003 BBA100    mov    bx,00A1      ■ bx 0000 |z=0|
    cs:0006 03C3      add    ax,bx      cx 0000 |s=0|
    cs:0008 BB0500    mov    bx,0005      dx 0000 |o=0|
    cs:000B 2BC3      sub    ax,bx      si 0000 |p=0|
    cs:000D B44C      mov    ah,4C      di 0000 |a=0|
    cs:000F CD21      int    21          bp 0000 |i=1|
    cs:0011 0000      add    [bx+si],al   sp 0100 |d=0|
    cs:0013 0000      add    [bx+si],al   ds 5C6D |
    cs:0015 0000      add    [bx+si],al   es 5C6D |
    cs:0017 0000      add    [bx+si],al   ss 5C7F |
    cs:0019 0000      add    [bx+si],al   cs 5C7D |
    cs:001B 0000      add    [bx+si],al   ip 0000 |

```
- Memory Dump:**

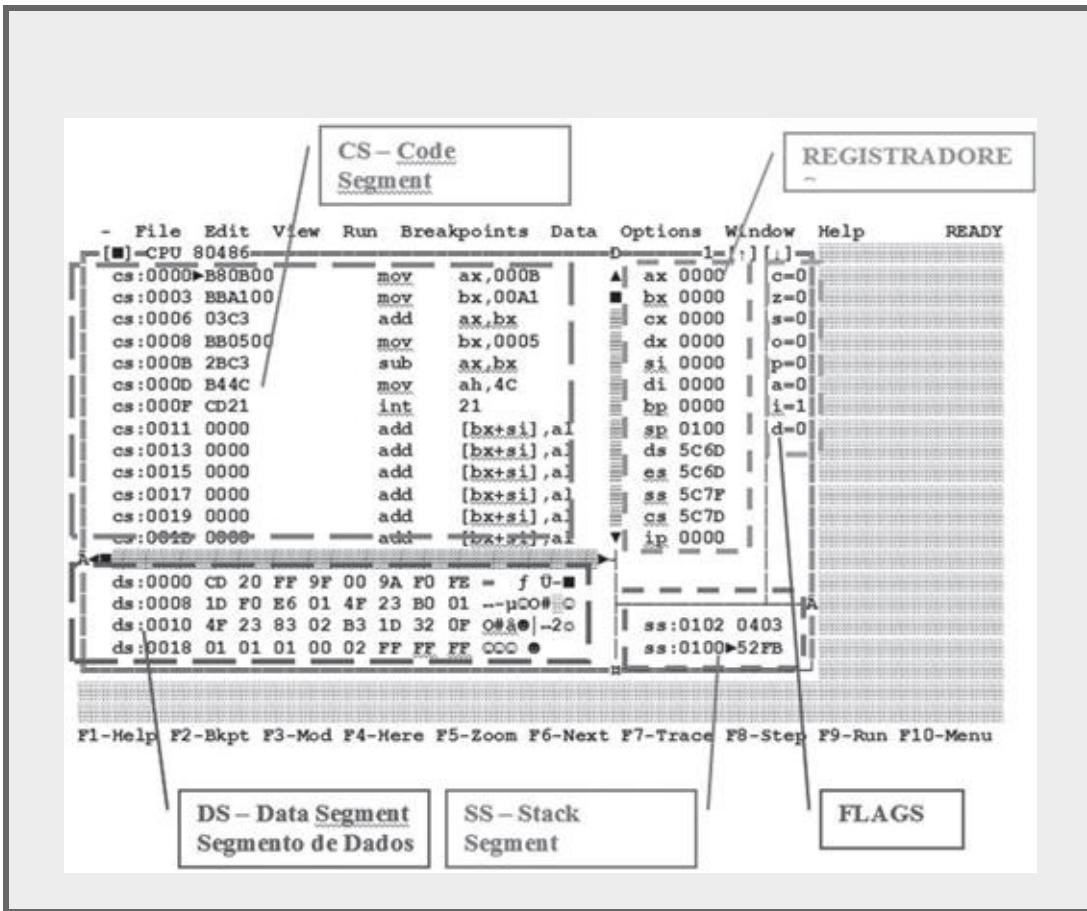
```

ds:0000 CD 20 FF 9F 00 9A F0 FE = f U-■
ds:0008 1D F0 E6 01 4F 23 B0 01 --p2O#|○
ds:0010 4F 23 83 02 B3 1D 32 0F 09@|.-2o
ds:0018 01 01 01 00 02 FF FF 000 * ss:0102 0403
ds:0018 01 01 01 00 02 FF FF 000 * ss:0100>52FB

```
- Keyboard Shortcuts:** F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

Na primeira linha, temos uma série de opções de comandos, e na última linha, as teclas de atalho para os comandos mais frequentes, tais como F8 – Step (executa o próximo comando – um passo do programa), F9 – Run (executa o programa até encontrar um final).

Além disso, podemos ver que existem áreas de exibição:



Observe que no segmento do código o ponteiro (que no segmento de registradores é identificado por IP – *instruction pointer*) aponta para a instrução MOV AX,000B, ou seja, a primeira instrução, pronta para ser executada.

Ao executar, pressionando F8, podemos observar:

The screenshot shows the Z80 CPU Emulator interface. The top menu bar includes File, Edit, View, Run, Breakpoints, Data, Options, Window, Help, and READY. The main window displays assembly code in the CPU window, registers in the Registers window, stack in the Stack window, and memory dump in the Dump window.

**CPU Window:**

```

- File Edit View Run Breakpoints Data Options Window Help READY
[■]-CPU 80486 D 1-[:][.]-
cs:0000 B80B00 mov ax,000B ▲ ax 000B |c=0|
cs:0003>BBA100 mov bx,00A1 ■ bx 0000 |z=0|
cs:0006 03C3 add ax,bx □ cx 0000 |s=0|
cs:0008 BB0500 mov bx,0005 □ dx 0000 |o=0|
cs:000B 2BC3 sub ax,bx □ si 0000 |p=0|
cs:000D B44C mov ah,4C □ di 0000 |a=0|
cs:000F CD21 int 21 □ bp 0000 |i=1|
cs:0011 0000 add [bx+si],al □ sp 0100 |d=0|
cs:0013 0000 add [bx+si],al □ ds 5C6D |
cs:0015 0000 add [bx+si],al □ es 5C6D |
cs:0017 0000 add [bx+si],al □ ss 5C7F |
cs:0019 0000 add [bx+si],al □ cs 5C7D |
cs:001B 0000 add [bx+si],al □ ip 0003 |

```

**Dump Window:**

```

ds:0000 CD 20 FF 9F 00 9A F0 FE = f Ü-■
ds:0008 1D F0 E6 01 4F 23 B0 01 --µCO#○
ds:0010 4F 23 63 02 B3 1D 32 0F O#â@|→○
ds:0018 01 01 01 00 02 FF FF 000 e
ss:0102 0403
ss:0100>52FB

```

**Bottom Bar:**

F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

Na opção RUN, no menu, temos também alguns comandos úteis:

- **Run – Run** Executa todo o código
- **Run – Programa Reset®** Inicia a execução do início.
- **Run – Go to cursor®** Executa até a linha que está sob o cursor. Por exemplo, reinicie a execução, selecione a quarta linha de instrução com o ponteiro do mouse. Execute novamente o código, com a opção Rn – Go to cursor. Observe que agora ele parou onde você tinha selecionado.
- **Run – Animated®** Por exemplo, reinicie a execução e execute o programa Run – Animated. Coloque um tempo igual a 20 e tecle OK.
- **Breakpoints – Toggle®** Estabelece um ponto de parada para a execução. Por exemplo, determine um ponto de parada no endereço 000B. Depois reinicie o programa (Run – Program Reset) e logo em seguida execute-o com Run – Run. Veja que agora o programa é executado até o ponto de parada estabelecido.

## ***Alguns comandos básicos e a sintaxe da linguagem de montagem***

Você já deve ter notado que devemos seguir algumas regras para podermos

escrever um programa que o montador entenda. A seguir, mostraremos estas regras:

## **Linhas de comando (statements)**

Uma linha de comando da linguagem de montagem deve ser formada como:

nome	operação	operando(s)	;	comentário
------	----------	-------------	---	------------

### **Exemplo:**

START: MOV CX, 5 ; inicializa o contador
--

Esses campos devem ser separados por espaço(s) definidos pelas teclas de espaço de “tab”. O símbolo START é um rótulo.

## **Campo nome**

O campo nome pode ser um rótulo de instrução, um nome de sub-rotina ou um nome de variável, podendo conter de **1 a 31 caracteres**. Ele deve iniciar com uma letra e conter somente letras, números e os caracteres ? . @ \_ : \$ %.

### **Exemplos:**

Nomes válidos	Nomes inválidos
LOOP1:	DOIS BITS

.TEST	2abc
@caracter	A42.25
SOMA_TOTAL4	#33
\$100	

**Observação:** O montador traduz os nomes para endereços de memória.

## Dados do programa

Os dados dos programas podem ser classificados como variáveis ou como constantes.

As variáveis devem ser declaradas da seguinte maneira:

*Variáveis do tipo Byte*

Nome DB valor\_inicial

### Exemplo:

ALPHA	DB	4
TESTE	DB	?

*Variáveis do tipo Word*

Nome DW valor\_inicial

## **Exemplo:**

WRD	DW	-2
TESTE	DW	?

As constantes são declaradas como:

*EQU (Equates)*

Nome EQU constante

## **Exemplo:**

LF EQU OAH

Isso associa à LF (constante) o valor em hexa 0A, que

corresponde ao código ASCII do caractere *Line Feed*.

**Observação:** Para mostrar em que base de numeração estão, os números devem ter uma letra para essa distinção. Se não colocarmos nenhuma, o montador conclui que estamos trabalhando no sistema decimal. Os identificadores de base são os seguintes:

- B ou b – binário.

*Exemplo:* 1110101b ou 1110101B

- D ou d decimal.

*Exemplo:* 64223 ou 64223d ou 64223D, 1110101 é considerado decimal (ausência do B), -2184D (número negativo)

- H ou h – hexadecimal.

*Exemplo:* 64223h ou 64223H, 0FFFFh (começa com um decimal e termina com h), 1B4Dh

## **Estrutura de um programa em linguagem de montagem**

A estrutura geral de um programa em linguagem de montagem é a seguinte:

### **TITLE NOME: DESCRIÇÃO**

• **MODEL SMALL**

• **STACK tamanho em hexa**

• **DATA ;os dados vão aqui**

• **CODE**

**MAIN PROC** ;as instruções vão aqui

**MAIN ENDP** ;outros procedimentos vão aqui

END MAIN

*Exemplos de programa:*

1. Nossa primeiro programa irá mostrar o caractere “\*” na tela.

**TITLE EX01: MOSTRA UM ASTERISCO**

; Primeiro programa exemplo. Mostra um asterisco na tela

• MODEL SMALL

• STACK 100H

• CODE

MAIN PROC

```
        ; Exibição do caractere
MOV AH,2    ; função para exibição de caractere
MOV DL,'*' ; caractere a ser exibido
INT 21H    ; interrupção que executa a função armaz.
            ; em AH (exibir caractere na tela)
            ; Retorna para o DOS
```

```
MOV AH,4CH    ; função para retorno ao DOS
INT 21H      ; interrupção que executa a função
            ; em AH
```

MAIN ENDP

END MAIN

Digite esse programa em um editor de texto, salvando-o com um nome significativo e com a extensão **.ASM**. Por exemplo: EX01.ASM

Faça a montagem (TASM) e a ligação (TLINK). Em seguida, execute o arquivo e verifique se ele mostra o caractere “\*” na tela.

Neste exemplo de programa foi apresentado o comando INT 21H. Este comando pode ser utilizado para realizar uma série de funções do DOS. A função que será executada tem o seu código (número) armazenado em AH indicando o que deve ser feito.

Dependendo da função a ser executada, outros registradores devem ser utilizados para guardar as informações de entrada e saída. No nosso exemplo, está sendo executada a função de escrita no dispositivo de saída padrão (código 2 em AH), que no nosso caso é o vídeo. Em DL estará o que se quer exibir.

Em resumo:

02H – Função para exibição de um caractere	
Entradas	AH = 2 DL = código ASCII do caractere a ser exibido
Saídas	AL = código ASCII do caractere exibido

2. Vamos montar um segundo programa, que agora irá solicitar a digitação de um caractere. Ao digitar uma determinada tecla, o caractere correspondente deverá ser exibido na tela. Em seguida, deve-se exibir novamente este mesmo caractere, porém, separado por um hífen do primeiro.

Programa para entrada de caractere:

```
TITLE EX02:RECEBE E MOSTRA UM CARACTERE
; Segundo programa exemplo. Recebe um caractere e o exibe
• MODEL SMALL
• STACK 100H
• CODE
MAIN PROC
; Entrada de um Caractere
```

MOV AH, 1	; função para entrada de caractere
INT 21H	; executa a função em AH

```
;;
; Depois de pressionada uma tecla, seu código ascii é armaz.
; em AL. Se for digitada uma tecla não caractere, AL recebe 0
; Salva o caractere digitado em BL
;
```

MOV BL,AL	; salva o caractere digitado em BL
-----------	------------------------------------

```
;;
; Exibe o Hifen
```

MOV AH, 2	; função para exibição de caractere
MOV DL, ‘-’	; caractere a ser exibido
INT 21H	; interrupção que executa a função armaz.
	; em AH (exibir caractere na tela)

; Recupera e Exibe o caractere digitado  
**MOV DL,BL**

MOV AH, 2	; função para exibição de caractere
INT 21H	; interrupção que executa a função armaz.
	; em AH (exibir caractere na tela)

; Retorna para o DOS

MOV AH, 4CH	; função para retorno ao DOS
INT 21H	; interrupção que executa a função em AH

**MAIN ENDP**  
**END MAIN**

Apresentamos neste programa mais uma função que pode ser utilizada: a entrada de caractere.

<b>4CH – Função para retorno ao S.O. (DOS)</b>	
Entradas	AH = 4CH
	AL = código de retorno
Saídas	Nenhuma

Usamos nos dois programas anteriores uma terceira função, que é o retorno do controle ao sistema operacional.

### 4CH – Função para retorno ao S.O. (DOS)

Entradas	AH = 4CH
	AL = código de retorno
Saídas	Nenhuma

3. Agora, vamos montar um programa para exibir uma sequência de caracteres (conhecida como *string*).

Para fazê-lo, devemos armazenar essa *string* em um local de memória, que pode ser uma variável chamada MSG:

```
MSG DB 'HELLO!$'
```

Note que o último caractere da mensagem é “\$”. Este caractere indica o final da *string* para a função de saída, e não é exibido.

Devemos utilizar uma função que faz a saída de uma *string*:

### 09H – Função para exibição de um a String

Entradas	AH = 9
	DX = endereço de offset da String
Saídas	Nenhuma

Esta função precisa carregar o *offset* de MSG em DX. Uma das formas de se fazer isso é utilizar a instrução LEA (*Load Effective Address*).

```
LEA destino,fonte
```

No nosso exemplo, no lugar onde colocamos a nossa mensagem na variável MSG, teríamos o seguinte comando:

```
LEA DX, MSG
```

Quando um programa é carregado na memória, o DOS cria e faz uso de um segmento de memória de 256 bytes que contém informações sobre o programa (este segmento é conhecido como PSP – *Program Segment Prefix*).

Com isso, o DOS coloca o número desse segmento nos registradores DS e ES antes de executar o programa.

O resultado disso é que, no início do programa, DS não conterá o endereço do segmento de dados.

Dessa forma, devemos colocar em DS o endereço correto do segmento de dados corrente. Para tanto, usamos:

```
MOV AX, @DATA  
MOV DS, AX
```

**@ DATA** é o nome do segmento de dados definido em .DATA. O montador traduz @DATA para o endereço base do segmento. Outro detalhe é que não podemos mover esse número diretamente para DS. Por isso, temos de utilizar duas instruções.

Concluindo, o programa que deve exibir uma mensagem fica assim:

```
TITLE MSG01:MOSTRA UMA MENSAGEM  
; Terceiro programa exemplo. Mostra uma mensagem na tela  
• MODEL SMALL  
• STACK 100H  
• DATA
```

```

MSG DB 'HELLO!$'
.CODE
MAIN PROC
; Inicializa DS
MOV AX, @DATA
MOV DS, AX ; inicializa DS
;
;exibe mensagem

```

LEA DX,MSG	; carrega o endereço da mensagem
MOV AH,9	; função para exibição de <i>string</i>
INT 21H	; executa a função em AH

;  
; Retorna para o DOS

MOV AH, 4CH	; função para retorno ao DOS
INT 21H	; interrupção que executa a função em AH

MAIN ENDP  
END MAIN



## O que vem depois

Bem, agora que já sabemos como construir programas simples utilizando a linguagem *assembly* e as ferramentas de edição, compilação e ligação, estudaremos a partir do próximo capítulo as instruções do conjunto de instruções do 8086.

---

## CAPÍTALO 15

---

# Instruções de movimentação de dados, lógicas, de deslocamento, de rotação e instruções aritméticas

---

### Objetivos Do Capítulo

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Compreender o mecanismo das instruções de movimentação de dados e das instruções que utilizam a ULA.
- Conhecer e utilizar as principais instruções em linguagem de montagem para armazenamentos de dados em registradores e na memória, para as operações lógicas e aritméticas.



### Apresentação



O armazenamento de informação é imprescindível na execução de um programa, pois sem isso o processador não conseguiria sequer decodificar uma

instrução.

Com essas instruções podemos ler da memória e escrever nela, carregar um determinado registrador com uma constante ou com o conteúdo de outro registrador, armazenar o resultado de uma operação em um determinado registrador, entre outras.

As operações de adição e subtração são importantes, pois a maioria dos programas necessita que estas operações sejam processadas. Não devemos esquecer que a ULA é a responsável por estas operações.

## Fundamentos

### **Instruções de movimentação de dados**

Como já vimos, as instruções de movimentação de dados nos permitem trazer as informações para dentro do processador, armazenando-as em um registrador para que elas possam ser processadas. Além disso, podemos também escrever uma informação na memória e copiar em outro registrador.

Os processadores da família x86 só não permitem que uma informação seja transferida de uma posição da memória para outra com a execução de apenas uma instrução de movimentação de dados. Para isso será necessário ler da memória, armazenar em um registrador e depois escrever novamente o que foi armazenado na memória.

Vamos agora apresentar o elenco das instruções de movimentação de dados, sua sintaxe e seu funcionamento.

O mnemônico para essas instruções é o símbolo **MOV** (*move*), e a sintaxe é:

MOV destino, fonte

A [Tabela 15.1](#) mostra as possibilidades de movimentação de dados. É permitida a transferência de dados entre dois registradores, entre um registrador e uma posição de memória e a movimentação de um número (constante) diretamente para um registrador ou para uma posição de memória.

---

**Tabela 15.1**

## Possibilidades de movimentação de dados – instrução MOV

Operando fonte	Operando destino		
	Registrador de dados	Registrador de segmento	Memória
Registrador de dados	SIM	SIM	SIM
Registrador de segmento	SIM	NÃO	SIM
Memória	SIM	SIM	NÃO
Constante	SIM	NÃO	SIM

A seguir temos exemplos de instruções válidas de movimentação de dados:

MOV AX,WORD1	; movimenta o conteúdo da posição de memória WORD1 para o registrador AX
MOV AH,'A'	; transfere o caractere ASCII 'A' para AH
MOV AH,41h	; idem anterior: 41h corresponde ao caractere A
MOV AH,BL	; move o conteúdo do byte baixo de BX ao byte alto de AX
MOV AX,CS	; transfere cópia do conteúdo de CS para AX

Agora, considere a execução da instrução:

```
MOV AX,WORD1
```

A [Tabela 15.2](#) nos mostra o conteúdo de AX antes e depois da execução da instrução.

---

### Tabela 15.2

#### Execução de MOV AX,WORD1

---

	Antes	Depois
AX	0006h	8FFFh
WORD1	8FFFh	8FFFh

A seguir, temos um exemplo de instruções de movimentação de dados inválidas e a solução para contornar o problema:

MOV WORD1,WORD2	; instrução inválida esta restrição é contornada como segue
MOV AX,WORD2 MOV WORD1,AX	; primeiro o conteúdo de WORD2 vai para AX
MOV WORD1,AX	; depois, o conteúdo de AX é movido para a posição de memória WORD1

Outra instrução de movimentação de dados permite a troca do conteúdo de duas localizações. O mnemônico desta instrução é **XCHG** (*exchange*), e sua sintaxe é:

XCHG destino,fonte

A [Tabela 15.3](#) nos mostra as possibilidades de movimentação de dados. É permitida a transferência de dados entre dois registradores de dados e entre um registrador e uma posição de memória.

---

### Tabela 15.3

#### Possibilidades de movimentação de dados – instrução XCHG

---

Operando fonte	Operando destino	
	Registrador de dados	Memória
Registrador de dados	SIM	SIM
Memória	SIM	NÃO

A seguir temos exemplos de instruções válidas para a troca de informações.

XCHG AX, WORD1	; troca o conteúdo da posição de memória WORD1 com o do registrador AX
XCHG AH, BL	; troca o conteúdo do byte baixo de BX com o do byte alto de AX

Agora, considere a execução da instrução:

XCHG AX,BX

A [Tabela 15.4](#) nos mostra o conteúdo de AX e BX antes e depois da execução da instrução.

---

#### Tabela 15.4

#### Execução de XCHG AX,BX

---

	Antes	Depois
AX	0006h	8FFFh
BX	8FFFh	0006h

Como exercício, vamos traduzir um comando de atribuição de uma linguagem de alto nível:

b = a ; (b recebe uma cópia do valor armazenado em a)

A tradução para a linguagem *assembly* seria:

MOV AX,a	; transfere o conteúdo da posição de memória a para AX
MOV b,AX	; transfere AX para a posição de memória b

Outra instrução de movimentação de dados é a instrução **LEA** (*Load Effective Address*), que permite que se carregue em um registrador o *offset* de um endereço lógico.

LEA destino,fonte

Considere o seguinte trecho de programa:

```
. DATA  
    MENSAGEM DB 'Adoro assembly!$'  
    ...  
.CODE  
    LEA DX,MENSAGEM ; DX carregado com o offset de  
    MENSAGEM
```

Após a execução da instrução LEA, DX conterá o *offset* do endereço lógico

formado por DS:MENSAGEM.

## Instruções De Adição E Subtração

As instruções de adição e subtração permitirão a realização dessas tão importantes operações nos programas que desenvolvermos.

A operação de adição usa o mnemônico ADD, e sua sintaxe é:

ADD destino, fonte

Já a operação de subtração usa o mnemônico SUB, e sua sintaxe é:

SUB destino, fonte

A [Tabela 15.5](#) nos mostra as possibilidades de adição e subtração de dados. É permitida a soma e a subtração de dados entre dois registradores de dados, entre um registrador e uma posição de memória, entre uma constante e um registrador e também entre uma constante e uma posição de memória.

---

### Tabela 15.5

#### Possibilidades de adição e subtração de dados – instruções ADD e SUB

---

Operando fonte	Operando destino	
	Registrador de dados	Memória
Registrador de dados	SIM	SIM
Memória	SIM	NÃO
Constante	SIM	SIM

A seguir temos exemplos de instruções válidas para a soma e subtração de operandos.

ADD AX,BX	; soma o conteúdo de AX com BX, resultado em AX
ADD AX,WORD1	; soma o conteúdo da posição de AX com o da memória WORD1, guardando em AX
ADD AL,5	; soma o conteúdo de AL com 5, resultando em AL
SUB AX,BX	; subtrai o conteúdo de AX com BX
SUB AX,WORD1	; subtrai o conteúdo da posição de AX com o da memória WORD1, guardando em AX
SUB AL,5	; subtrai o conteúdo de AL com 5, resultado em AL

Agora, considere a execução da instrução:

ADD AX, BX

A [Tabela 15.6](#) nos mostra o conteúdo de AX e BX antes e depois da execução da instrução.

---

### Tabela 15.6

## Execução de ADD AX,BX

---

	Antes	Depois
AX	0006h	8FF6h
BX	8FF0h	8FF0h

Admita agora a execução dessa instrução:

SUB AX, BX

A [Tabela 15.7](#) nos mostra o conteúdo de AX e BX antes e depois da execução da instrução.

---

**Tabela 15.7**

## Execução de ADD AX,BX

---

	Antes	Depois
AX	0006h	0001h
BX	0005h	0005h

A seguir apresentamos um exemplo de instruções inválidas de adição de dados e a solução para contornar o problema (o mesmo pode ser aplicado à subtração):

ADD BYTE1,BYTE2	; instrução inválida esta restrição é contornada como segue
MOV AL,BYTE2	; primeiro o conteúdo de BYTE2 vai para AL
MOV BYTE1,AL	; depois, o conteúdo de AL é subtraído de BYTE1 e armazenado na posição de memória BYTE1

Existem outras instruções aritméticas, como as operações de incremento,

decremento e cálculo de complemento de 2.

A operação de incremento (somar 1 ao operando) utiliza o mnemônico **INC**, e sua sintaxe é:

```
INC destino
```

A operação de decremento usa o mnemônico **DEC**, e sua sintaxe é

```
DEC destino
```

Essas instruções utilizam como fonte e destino um mesmo operando, que pode ser um registrador ou uma posição de memória.

```
INC CX      ; incrementa o conteúdo de CX
INC WORD1   ; incrementa conteúdo posição de memória
              WORD1
DEC BYTE2   ; decrementa conteúdo posição de
              memória BYTE2
```

Vemos agora como ficaria o conteúdo do operando após a execução da instrução:

## INC BYTE1

A [Tabela 15.8](#) nos mostra o conteúdo de BYTE1 antes e depois da execução da instrução.

---

**Tabela 15.8**  
**Execução de INC BYTE1**

---

BYTE1	
Antes	Depois
0006h	0007h

Outra instrução aritmética bastante usada é a que calcula o complemento de 2 de um número. O mnemônico para esta instrução é **NEG**, e o operando pode ser um registrador ou uma posição de memória. A sintaxe desta instrução é:

NEG destino

As instruções a seguir são instruções válidas para o **NEG**:

NEG BX	; gera o complemento de 2 do conteúdo de BX
NEG WORD1	; idem, no conteúdo da posição de memória WORD1

Vemos a seguir como ficaria o conteúdo do operando após a execução da instrução:

NEG BX

A [Tabela 15.9](#) nos mostra o conteúdo de BYTE1 antes e depois da execução da instrução.

---

**Tabela 15.9**  
**Execução de NEG BX**

---

BX	
Antes	Depois
0002h	FFFFh

Agora vamos mostrar com alguns exemplos o uso dessas instruções na tradução de sentenças escritas em linguagem de alto nível.

**Exemplos:**

1. Considere a sentença em linguagem de alto nível a seguir:

$$A = 5 - A$$

Essa sentença significa que a variável **A** recebe o resultado de 5 menos o valor anterior de **A**.

A tradução em linguagem *assembly* 8086 é:

```
MOV AX, 5  
SUB AX, A  
MOV A, AX
```

2. Considere a sentença em linguagem de alto nível a seguir:

$$A = B - 2 \times A$$

Essa sentença significa que a variável **A** recebe o resultado de B menos duas vezes o valor anterior de **A**.

Sua tradução em linguagem *assembly* 8086 é:

```
MOV AX, B  
SUB AX, A  
SUB AX, A  
MOV A, AX
```

# As Instruções De Movimentação De Dados, De Adição E De Subtração, E Os Flags

Como vimos anteriormente, os *flags* são responsáveis por mostrar o estado da UCP após a execução das instruções. Na [Tabela 15](#) a seguir, podemos ver quais *flags* são afetados pelas instruções vistas neste capítulo.

---

**Tabela 15.10**

## Os *flags* e as instruções de movimentação de dados e aritméticas

---

Instrução	Flags alterados
MOV	NENHUM
XCHG	NENHUM
LEA	NENHUM
ADD/SUB	TODOS
INC/DEC	TODOS, EXCETO CF
NEG	Todos (CF= 1, a não ser que o resultado seja igual a zero, of = 1 se o operando for a palavra 8000h ou um byte 80h)

# Instruções Lógicas

As instruções lógicas, além de obviamente executarem as respectivas funções lógicas, são utilizadas para:

- Zerar (*reset*) ou limpar (*clear*) um bit, ou seja, forçá-lo a ter o valor 0.
- Forçar ao valor 1 (*set*) um bit.
- Verificar valores de bits por meio de operações lógicas com máscaras específicas.

As instruções lógicas válidas para o 8086 seguem as funções constantes na [Tabela 15.11](#), que na verdade representa as tabelas-verdade das funções **AND** (e lógico), **OR** (ou lógico), **NOT** (inversão) e **XOR** (ou-exclusivo), como foi visto na Parte 2 deste livro.

---

**Tabela 15.11**

## Tabela-verdade das funções lógicas constantes do conjunto de instruções do 8086

---

<b>a</b>	<b>b</b>	<b>a AND b</b>	<b>a OR b</b>	<b>a XOR b</b>	<b>NOT A</b>
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

## ***As instruções lógicas AND, OR e XOR***

As instruções lógicas AND, OR e XOR têm a mesma sintaxe, apenas o mnemônico é diferente.

Portanto, temos:

Sintaxe da instrução AND:

AND destino, fonte

Sintaxe da instrução OR:

OR destino, fonte

Sintaxe da instrução XOR:

XOR destino, fonte

A [Tabela 15.12](#) nos mostra as possibilidades de operações lógicas de dados. É

permitida a operação lógica de dados entre dois registradores de dados, entre um registrador e uma posição de memória, entre uma constante e um registrador e também entre uma constante e uma posição de memória.

**Tabela 15.12**

### Possibilidades de adição e subtração de dados – instruções lógicas

Operando fonte	Operando destino	
	Registrador de dados	Memória
Registrador de dados	SIM	SIM
Memória	SIM	NÃO
Constante	SIM	SIM

A seguir temos exemplos de instruções válidas para a soma e subtração de operandos.

Essas instruções alteram os flags SF, ZF, PF, de acordo com o valor armazenado no operando destino. O flag AF não é alterado, e os flags CF e OF são zerados.

As instruções seguintes são exemplos de instruções válidas:

```
XOR AX,BX ; operador XOR aplicado aos conteúdos  
              de AX e BX, resultado em AX
```

```
AND CH,01h ; operador AND aplicado ao conteúdo de  
              CH, tendo como fonte o valor imediato  
              01h = 0000 0001b
```

```
OR WORD1,BX ; operador OR entre conteúdos da  
              posição de memória WORD1 e de BX,  
              resultado armazenado em WORD1
```

Vamos supor inicialmente que AL tenha 0Ah (1010h) e BL 0Eh (1110h). Executaremos então:

AND BL, AL
OR BL, AL
XOR BL, AL

Depois das execuções anteriores, teremos:

---

**Tabela 15.13**  
**Valores após a execução de instruções lógicas**

---

Valores após a execução	BL	AL
AND BL,AL	1010h	1110h
OR BL,AL	1110h	1110h
XOR BL,AL	0000h	1110h

## Criação De Uma Máscara

As máscaras são números binários, projetados especificamente para a manipulação de bits por meio de operações lógicas. Para o projeto destas máscaras, temos de levar em conta os seguintes princípios:

- A função AND pode ser utilizada para zerar (*clear* ou *reset*) bits específicos; para isso, basta ter um 0 na posição em que se deseja este efeito.
  - A função OR pode ser utilizada para forçar ao valor 1 (*set*) bits específicos; para isso, basta ter um 1 na posição em que se deseja este efeito.
  - A função XOR pode ser utilizada para complementar (*inverter*) bits específicos; para isso, basta ter um 1 na posição em que se deseja este efeito.
- A seguir vamos exemplificar a definição de máscaras para um fim específico.

<b>Exemplos:</b>
------------------

1. Para forçar ao valor 1 os bits MSB e LSB do registrador AX, dado AX = 7444h, executaremos a instrução:

```
OR AX,8001h
```

A palavra 8001h é a máscara. Ela foi definida porque queremos fazer com que os bits MSB e LSB sejam forçados ao valor 1. Por isso a função é o OR e a máscara deve ter 1 no MSB e 1 no LSB. Os demais bits têm de ser zero para manter os bits originais de AX:

AX (antes) →	0111 0100 0100 0100b	→ 7444h
	1000 0000 0000 0001b	→ 8001h
OR		
AX (depois) →	1111 0100 0100 0101b	→ F445h

2. Vamos converter o código ASCII de um dígito numérico para seu valor binário:

```
AND AL,0Fh ;em substituição a: SUB AL,30h
```

```
AL (antes) → 0011 0111b → 37h = "7" = 55d  
          0000 1111b → 0Fh  
AND  
AL (depois) → 0000 0111b → 07h = 7d  
                  (valor sete)
```

3. Vamos converter uma letra minúscula em maiúscula, considerando o caractere em AL:

```
AND AL,0DFh
```

```
AL (antes) → 0110 0001 → 61h = "a"  
          1101 1111b → AND  
  
AL (depois) à 0100 0001b à 41h = "A"
```

**Observação:** Para essa conversão, tem-se apenas de zerar o bit 5 de AL.

4. Limpando (zerando) um registrador:

```
XOR AX, AX
```

```
AX (antes) → 0111 0100 0100 0100b → 7444h  
AX (antes) → 0111 0100 0100 0100b → 7444h  
XOR  
AX (depois) → 0000 0000 0000 0000b → 0000h = 0
```

**Observação:** Essa forma é mais rápida de executar do que as opções MOV AX,0000h e SUB AX,AX.

5. Testando se o conteúdo de algum registrador é zero:

```
OR CX,CX
```

```
CX (antes) → 0111 0100 0100 0100b → 7444h  
CX (antes) → 0111 0100 0100 0100b → 7444h  
OR  
CX (depois) → 0111 0100 0100 0100b → 7444h  
(não é 0)
```

**Observação:** Essa operação deixa o registrador CX inalterado, e modifica o *flag ZF* somente quando o conteúdo de CX é realmente zero. É mais rápida do que CMP CX,0000h.

## A instrução lógica NOT

A instrução lógica NOT apresenta a seguinte sintaxe:

```
NOT destino
```

Elas é usada para aplicar o operador lógico NOT em todos os bits de um registrador e de uma posição de memória. O resultado é a complementação (inversão) de todos os bits. Nenhum *flag* é alterado.

A seguir temos exemplos válidos da utilização da instrução NOT.

NOT AX	; inverte todos os bits de AX
NOT AL	; inverte todos os bits de AL
NOT BYTE1	; inverte todos os bits do conteúdo posição de memória definida pelo nome BYTE1

Considere a palavra 81h armazenada na posição de memória WORD1, e a posterior execução da instrução:

```
NOT WORD1
```

Assim, teremos:

---

**Tabela 15.14**  
**Execução da instrução NOT WORD1**

---

WORD1	
Antes	81h = 1000 0001b
Depois	7Eh = 0111 1110b

## A Instrução Lógica TEST

A instrução lógica TEST tem a seguinte sintaxe:

TEST destino,fonte

Essa instrução é usada para aplicar o operador lógico AND em dois operadores, como mostra a [Tabela 15.15](#).

**Tabela 15.15**

**Possibilidade de combinação de operandos para a instrução TEST**

Operando fonte	Operando destino	
	Registrador de dados	Memória
Registrador de dados	SIM	SIM
Memória	SIM	NÃO
Constante	SIM	SIM

A diferença dessa instrução para a instrução AND é o destino, não alterado na instrução aqui considerada. Apenas os *flags* SF, ZF e PF são alterados de acordo com o resultado. O *flag* AF não é afetado e CF e OF ficam zerados.

Alguns exemplos de utilização válida para as instruções TEST:

TEST AX,BX	; operação AND entre AX e BX, não há resultado, mas apenas alteração dos FLAGS ZF, SF e PF
TEST AL,01h	; operação AND entre AL e o valor imediato 01h

## **Exemplos:**

1. Suponha que queiramos testar se um dado número é par sem alterá-lo; o número está armazenado em AX e inicialmente 44h. A instrução a seguir faz exatamente isso:

```
TEST AX, 0001h
```

A máscara 0001h serve para testar se o conteúdo de AX é par (todo número binário par possui um zero no LSB). O número 4444h é par, pois o seu LSB vale zero (01000100b). A operação AND entre 4444h e 0001h produz como resultado 0000h, alterando ZF para 1. Portanto, se testarmos ZF saberemos se o número em AX é par ou ímpar. O resultado não é armazenado em AX, e somente ZF é modificado por TEST.

2. Vamos escrever um trecho de programa que salte para o rótulo PONTO2 se o conteúdo de CL for negativo:

```
....  
TEST CL,80h ; 80h é a máscara 10000000b  
JNZ PT2  
....  
(o programa prossegue, pois o número é positivo)  
....
```

PONTO2: ....  
(o programa opera aqui com o número negativo)  
....

## Instruções De Deslocamento E Rotação

Além de permitirem a alteração de posição de bits, com as instruções de deslocamento (*shift*) podemos:

- Multiplicar um número por dois a cada deslocamento para a esquerda, de uma casa binária.
- Dividir um número por dois a cada deslocamento para a direita, de uma casa binária.

**Observação:** Os bits deslocados para fora da palavra ou do byte são perdidos.

Já as instruções de rotação (*rotate*) permitem deslocar de forma circular (em anel) para a esquerda ou para a direita, sem que nenhum bit seja perdido.

### *Instruções de deslocamento*

Temos dois tipos de deslocamento: o deslocamento lógico e o deslocamento aritmético. A diferença entre os dois é que o aritmético leva em conta o sinal do número, e o lógico não. Na prática, os deslocamentos lógicos e aritméticos para a esquerda funcionam da mesma maneira, ou seja, o bit mais à esquerda (MSB) sai da palavra, todos os outros são deslocados uma posição para a direita e colocamos um zero no lugar LSB. No deslocamento lógico à direita, deslocamos todos os bits uma posição para a direita e colocamos um zero no MSB. No deslocamento aritmético para a direita, procedemos da mesma maneira, com exceção do bit colocado no MSB, que precisa ter o mesmo valor que o anterior; assim, manteremos o sinal do número.

As instruções de deslocamentos obedecem à seguinte sintaxe:

Sxx destino, 1

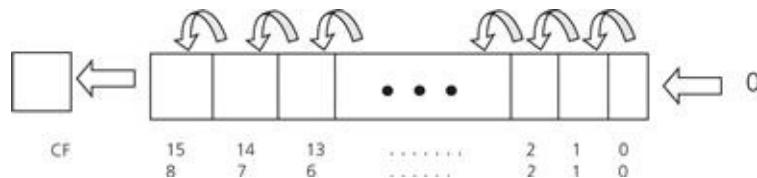
## Sxx destino, CL

Essas instruções são usadas para deslocar, para a esquerda ou para a direita, de 1 bit ou tantos quantos CL indicar. Eles podem deslocar o conteúdo de um registrador ou de uma posição de memória.

A generalização do mnemônico Sxx corresponde a:

- **SHL (Shift Left)** – Deslocamento para a esquerda.

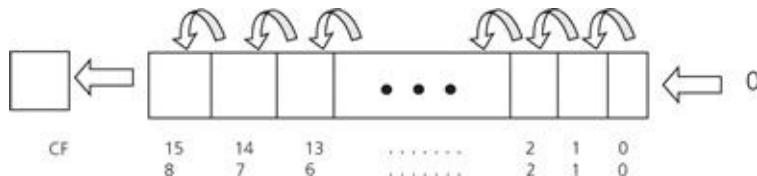
A [Figura 15.1](#) mostra o esquema de funcionamento da instrução SHL.



**FIGURA 15.1** Esquema de funcionamento da instrução SHL.

- **SAL (Shift Arithmetic Left)** – Deslocamento aritmético para a esquerda

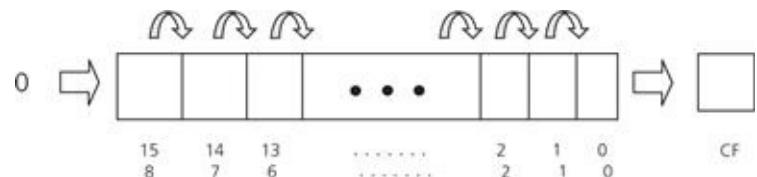
A [Figura 15.2](#) mostra o esquema de funcionamento da instrução SAL.



**FIGURA 15.2** Esquema de funcionamento da instrução SAL.

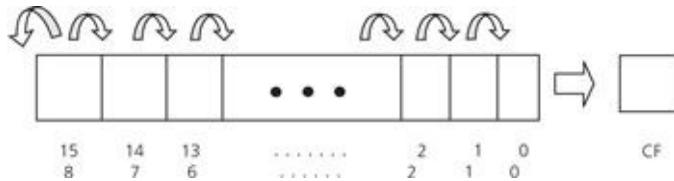
- **SHR (Shift Right)** – Deslocamento para a direita

A [Figura 15.3](#) mostra o esquema de funcionamento da instrução SHR.



**FIGURA 15.3** Esquema de funcionamento da instrução SHR.

- **SAR (Shift Arithmetic Right)** - Deslocamento aritmético para a direita  
A [Figura 15.4](#) mostra o esquema de funcionamento da instrução SAR.



**FIGURA 15.4** Esquema de funcionamento da instrução SAR.

Essas instruções afetam os flags SF, ZF e PF, de acordo com o resultado do último deslocamento. Já o flag AF não é afetado, e o CF contém o último bit deslocado para fora. Por fim, o OF será 1 se ocorrer troca de sinal após o último deslocamento.

A seguir, alguns exemplos de utilização válida dessas instruções:

SHL AX, 1	; desloca os bits de AX para a esquerda 1 casa binária, sendo o LSB igual a zero
SAL BL, CL	; desloca os bits de BL para a esquerda tantas casas binárias quantas CL indicar, os bits menos significativos são zero (mesmo efeito de SHL)
SAR DH, 1	; desloca os bits de DH para a direita 1 casa binária, sendo que o MSB mantém o sinal

## Instruções de rotação

Diferentemente das instruções de deslocamento, nas instruções de rotação nenhum bit é perdido, uma vez que o bit que sai de uma extremidade entra pela outra. Temos aqui também dois tipos de rotação: uma acontece através do CF, e a outra não. No segundo caso (rotação através do CF), considerando a rotação para a esquerda, o bit que sai na posição MSB entra na posição LSB; já na rotação

para a direita, acontece o contrário. Na rotação à esquerda através do CF, o bit que sai do MSB entra no CF, e o que estava no CF entra no LSB; já na rotação para a direita, temos o contrário.

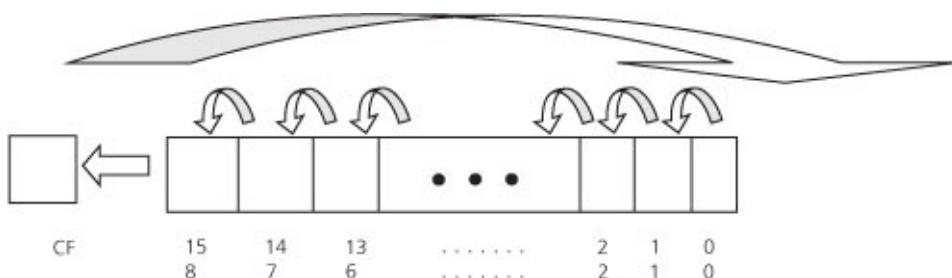
As instruções de rotação obedecem à seguinte sintaxe:

Rxx	destino, 1
Rxx	destino, CL

Essas instruções são usadas a fim de deslocar para a esquerda ou para a direita (e em anel) 1 bit ou tantos quantos CL indicar. Elas podem rotacionar o conteúdo de um registrador ou de uma posição de memória.

A generalização do mnemônico Rxx corresponde a:

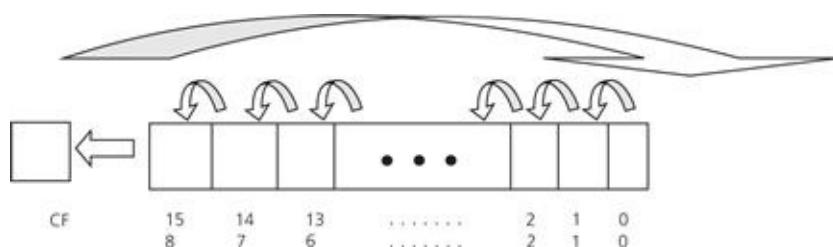
- **ROL (Rotate Left)** – Rotacionar para a esquerda.



**FIGURA 15.5** Esquema de funcionamento da instrução ROL.

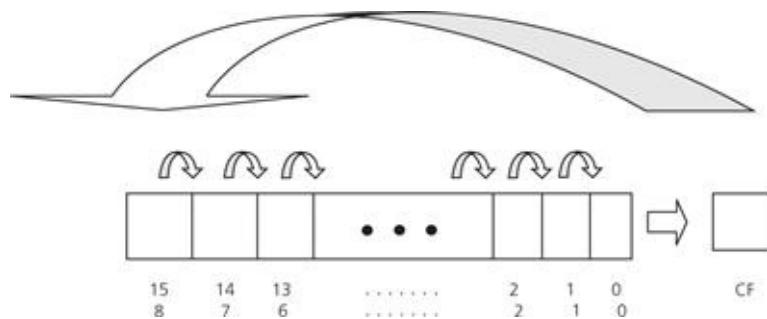
- **ROR (Rotate Right)** – Rotacionar para a direita.

A [Figura 15.6](#) mostra o esquema de funcionamento da instrução ROR.



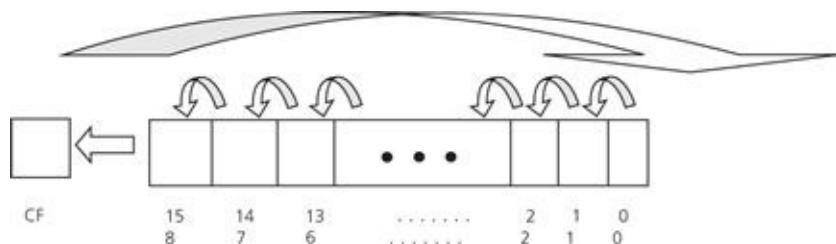
**FIGURA 15.6** Esquema de funcionamento da instrução ROR.

- **RCL (Rotate Carry Left)** – Rotacionar para a esquerda através do *flag* CF.  
A [Figura 15.7](#) mostra o esquema de funcionamento da instrução RCL.



**FIGURA 15.7** Esquema de funcionamento da instrução RCL.

- **RCR (Rotate Carry Right)** – Rotacionar para a direita através do *flag* CF.  
A [Figura 15.8](#) mostra o esquema de funcionamento da instrução RCL.



**FIGURA 15.8** Esquema de funcionamento da instrução RCL.

Essas instruções afetam os *flags* SF, ZF e PF, de acordo com o resultado da última rotação. O *flag* AF não é afetado, o CF contém o último bit deslocado para fora e OF será 1 se ocorrer troca de sinal após a última rotação.

A seguir, alguns exemplos de utilização válida dessas instruções:

ROL AX, 1	; desloca os bits de AX para a esquerda 1 casa binária, sendo que o MSB é reinserido na posição LSB
ROR BL,	; desloca os bits de BL para a direita tantas casas binárias quantas CL indicar, os bits menos significativos são reinseridos um-a-um no MSB

CL	
RCR DH, 1	; desloca os bits de DH para a direita 1 casa binária, sendo que o MSB recebe CF e o LSB é salvo em CF

## Exemplos:

1. Fazer um programa que permita a entrada de números binários, com as seguintes especificações:
    - String de caracteres 0 e 1 fornecidos pelo teclado.
    - CR como o marcador de fim de *string*.
    - BX como o registrador de armazenamento.
    - Número com 16 bits no máximo.
- Algoritmo básico em pseudocódigo:

```

Limpa BX
Entra um caractere “0” ou “1”
WHILE caractere diferente de CR DO
  Converte caractere para valor binário
  Desloca BX 1 casa para a esquerda
  Insere o valor binário lido no LSB de BX
  Entra novo caractere
END_WHILE
  
```

Trecho de programa implementado em linguagem de montagem:

...	
MOV CX, 16	; inicializa contador de dígitos
MOV AH, 1h	; função DOS para entrada pelo teclado
XOR BX, BX	; zera BX – > terá o resultado

INT 21h	; entra, caractere está no AL
; while	
PT: CMP AL, 0Dh	; é CR?
JE FIM	; se sim, termina o WHILE
AND AL, 0Fh	; se não, elimina 30h do caractere
SHL BX, 1	; abre espaço para o novo dígito
OR BL, AL	; insere o dígito no LSB de BL
INT 21h	; entra novo caractere
LOOP PT	; controla o máximo de 16 dígitos
; end_while	
FIM:	
...	

2. Fazer um programa que permita a saída de números binários, com as seguintes especificações:

- BX deverá ser o registrador de armazenamento.
- Deve haver um total de 16 bits de saída.
- Um *string* de caracteres 0 e 1 deve ser exibido no monitor de vídeo.

Algoritmo básico em pseudocódigo:

```

FOR 16 vezes DO
    rotação de BX à esquerda 1 casa binária (MSB vai
    para o CF)
    IF CF = 1
        THEN exibir no monitor caractere “1”
        ELSE exibir no monitor caractere “0”
    END_IF
END_FOR
  
```

Trecho de programa implementado em linguagem de montagem:

```

...
    MOV CX,16      ;inicializa contador de bits
    MOV AH,02h      ;prepara para exibir no monitor
;for 16 vezes do
PT1: ROL BX,1       ;desloca BX 1 casa à esquerda
;if CF = 1
    JNC PT2        ;salta se CF = 0
;then
    MOV DL, 31h     ;como CF = 1
    INT 21h         ;exibe na tela "1" = 31h
;else
PT2: MOV DL, 30h   ;como CF = 0
    INT 21h         ;exibe na tela "0" = 30h
;end_if
    LOOP PT1        ;repete 16 vezes
;end_for
Entrada de números hexadecimais

```

3. Fazer um programa que permita a entrada de números hexadecimais, com as seguintes especificações:

- Entrada de números hexadecimais.
- BX como o registrador de armazenamento.
- *String* de caracteres “0” a “9” ou “A” a “F” digitado no teclado.
- Máximo de 16 bits de entrada ou máximo de quatro **dígitos hexadecimais**.

Algoritmo básico em pseudocódigo

#### Inicializa BX

Entra um caractere hexa

WHILE caractere diferente de CR DO

Converte caractere para binário

Desloca BX 4 casas para a esquerda

Insere valor binário nos 4 bits inferiores de BX

Entra novo caractere

## END WHILE

Trecho de programa implementado em linguagem de montagem

...	
XOR BX, BX	;inicializa BX com zero
MOV CL, 4	;inicializa contador com 4
MOV AH, 1h	;prepara entrada pelo teclado
INT 21h	;entra o primeiro caractere
;while	
PT: CMP AL,0Dh	;é o CR?
JE FIM	
CMP AL, 39h	;caractere número ou letra?
JG LTR	;caractere já está na faixa ASCII
AND AL, OFh	;número: retira 30h do ASCII
JMP DSL	
LTR: SUB AL,37h	;converte letra para binário
DSL: SHL BX,CL	;desloca BX 4 casas à esquerda
OR BL,AL	;insere valor nos bits 0 a 3 de BX
INT 21h	;entra novo caractere
JMP PT	;faz o laço até que haja CR
;end_while	
FIM: ...	

Fazer um programa que permita a saída de números hexadecimais, com as seguintes especificações:

- BX deve ser o registrador de armazenamento.
- Deverá haver um total de 16 bits de saída.
- Um *string* de caracteres HEXA deve ser exibido no monitor de vídeo.

Algoritmo básico em pseudocódigo:

```
For 4 vezes DO
    Mover BH para DL
    Deslocar DL 4 casas para a direita
    IF DL < 10
        THEN converte para caractere na faixa 0 a 9
        ELSE converte para caractere na faixa A a F
    END_IF
    Exibição do caractere no monitor de vídeo
    Rodar BX 4 casas à esquerda
END_FOR
```

Trecho de programa implementado em linguagem de montagem:

```

...
;BX já contém número binário
    MOV CH,4      ; CH contador de caracteres hexa
    MOV CL,4      ; CL contador de deslocamentos
    MOV AH,2h     ; prepara exibição no monitor
;for 4 vezes do
PT: MOV DL,BH    ; captura em DL os oito bits mais
               significativos de BX
    SHR DL,CL    ; resta em DL os 4 bits mais
               significativos de BX
;if DL , 10
    CMP DL, 0Ah  ;testa se é letra ou número
    JAE LTR
;then
    ADD DL,30h   ;é número: soma-se 30h
    JMP PT1
;else
LTR: ADD DL,37h  ;ao valor soma-se 37h -> ASCII
;end_if
PT1: INT 21h    ;exibe
    ROL BX,CL    ;roda BX 4 casas para a direita
    DEC CH
    JNZ PT       ;faz o FOR 4 vezes
;end_for

```

## Instruções De Multiplicação E Divisão

### *Instruções de multiplicação*

Temos dois tipos de multiplicação, a sinalizada e a não sinalizada. Para cada uma delas temos um mnemônico: o **MUL** (de *multiply*), para a multiplicação de números não sinalizados, e o **IMUL** (de *integer multiply*), para a multiplicação de números sinalizados. As sintaxes destas duas instruções são:

**MUL** fonte

**IMUL** fonte

Para a multiplicação com números em formato *byte*, o operando fonte deverá ser um registrador de 8 bits ou uma variável de tipo DB. Neste caso, o segundo operando da multiplicação obrigatoriamente estará em AL, e o resultado, um número de 16 bits, será guardado em AX.

Quando a multiplicação for com números em formato *word*, o operando fonte deverá ser um registrador de 16 bits ou uma variável do tipo DW. Aqui, o segundo operando estará obrigatoriamente em AX, e o resultado de 32 bits (tamanho *doubleword*) será armazenado no par de registradores DX:AX, ficando em DX os 16 bits mais significativos (*high word*) e em AX os 16 bits menos significativos (*low word*).

As instruções de multiplicação alteram os *flags* desta forma:

SF, ZF, AF e PF ficarão indefinidos;

Após MUL, CF/OF (ambos) será 0, se a metade superior do resultado for 0; caso contrário, CF/OF será 1.

Após IMUL, CF/OF (ambos) será 0, se a metade superior do resultado for a extensão do sinal da metade inferior; caso contrário, CF/OF será 1,

### **Exemplos:**

1. Vamos supor inicialmente que AX contenha 0001h BX contenha FFFFh para a instrução:

**MUL BX**

Como AX = 1 (número não sinalizado) e BX = 65535, teríamos um novo valor em AX que seria 0000FFFFh (65535d), que é o resultado de uma multiplicação não sinalizada de AX por BX.

Para a instrução:

IMUL BX

Como AX = +1 (número sinalizado) e BX = -1, teríamos um novo valor em AX que seria 0000FFFFh (-1), que é o resultado de uma multiplicação sinalizada de AX por BX.

O quadro a seguir mostra o resultado completo da execução das duas instruções anteriores.

Instrução	Resultado decimal	Resultado hexadecimal	DX	AX	CF/OF
MUL BX	65535	0000FFFFh	0000h	FFFFh	0
IMUL BX	-1	FFFFFFFh	FFFFh	FFFFh	0

2. Suponha agora que AX contenha 0FFF h.

No quadro a seguir, temos a execução das instruções:

MUL AX

e

## IMUL AX

Instrução	Resultado decimal	Resultado hexadecimal	DX	AX	CF/OF
MUL AX	16769025	00FFE001h	00FFh	E001h	1
IMUL AX	+16769025	00FFE001h	00FFh	E001h	1

3. Agora, trabalhando com operações de 8 bits, vamos supor que AL contenha 80h ( $= 10000000_2 = 128_{10}$  ou  $-128_{10}$ , se sinalizado) e BL tenha FFh ( $= 11111111_2 = 255_{10}$  ou  $-1_{10}$ , se sinalizado):  
No quadro a seguir, temos a execução das instruções:

## MUL BL

## IMUL BL

Instrução	Resultado decimal	Resultado hexadecimal	AH	AL	CF/OF
MUL BL	32640	7F80h	7Fh	80h	1
IMUL BL	+128	0080h	00h	80h	1

## ***Instruções de divisão***

Analogamente à multiplicação, a divisão também tem dois tipos de operação: a sinalizada e a não sinalizada. Para a divisão de números não sinalizados temos o mnemônico **DIV** (*divide*), e para a divisão de números sinalizados temos o **IDIV** (*integer divide*). As sintaxes destas duas instruções são:

DIV fonte

IDIV fonte

O operando fonte é o divisor de 8 bits ou de 16 bits, se a operação for igualmente de 8 ou 16 bits. Ele não pode ser uma constante. Para uma operação de 8 bits, na qual o divisor é um registrador de 8 bits ou uma posição de memória de 8 bits (tipo DB), o dividendo tem de ser uma palavra de 16 bits, armazenada em AX. Após a execução da divisão, o quociente (de 8 bits) estará em AL e o resto (também de 8 bits) estará em AH. Se a divisão for de 16 bits, na qual o divisor é uma palavra de 16 bits, armazenada em AX ou em uma posição de memória de 16 bits (tipo DW), o dividendo será de 32 bits, armazenado no par DX:AX. O quociente desta divisão estará em AX (16 bits) e o resto em DX (16 bits). Nestas operações, todos os FLAGS são afetados e ficam indefinidos. O resto, em divisão com números sinalizados, tem o mesmo sinal do dividendo.

## **Exemplos:**

1. Suponha que DX e AX contenham, respectivamente, 0000h e 0005h (5d ou +5d, se sinalizado), e BX contenha FFFEh (=65534d ou -2d, se sinalizado).

O quadro a seguir mostra o resultado da execução das instruções:

DIV BX

IDIV BX

Instrução	Quociente decimal	Resto decimal	AX	DX
DIV BX	0	5	0000h	0005h
IDIV BX	-2	1	FFFEh	0001h

2. Suponha agora que AX contenha 0005h (= 5d ou +5d, se sinalizado) e BL contenha FFh (= 256d ou -1d, se sinalizado). O quadro a seguir mostra o resultado da execução das instruções:

DIV BX

IDIV BX

Instrução	Quociente decimal	Resto decimal	AL	AH
DIV BL	0	5	00h	05h
IDIV BL	-5	0	FBh	00h

3. Suponha que AX contenha 00FBh (= 251d ou +251d, se sinalizado) e BL contenha FFh (= 255d ou -1, se sinalizado):

Instrução	Quociente decimal	Resto decimal	AL	AH
DIV BL	0	251	00h	FBh
IDIV BL	-251 *	-	-	-

**Observação:** Como -251 não pode ser representado com 8 bits, ocorre o que chamamos de “estouro de divisão” (*divide overflow*), um erro que faz com que o programa termine.

### **Extensão do sinal do dividendo**

Nas operações de divisão sinalizadas, existe o cuidado de estender o valor do sinal para a parte superior da palavra, *byte* ou *word*, caso o dividendo de 16 bits ocupe apenas AL ou o dividendo de 32 bits ocupe apenas AX.

Para esclarecer melhor vamos verificar as seguintes situações:

■ **Operações em formato word:** caso o dividendo de uma divisão (composto de **DX:AX**) ocupe apenas o registrador AX, o registrador DX deve ser preparado, pois é sempre considerado na execução da operação. Se a instrução for **DIV**, o DX deve ser zerado, pois estamos tratando de operação de números não sinalizados. Se for **IDIV**, o DX deve ter a extensão de sinal de AX, pois se trata de operações de números sinalizados. A instrução **CWD** faz a conversão de um palavra *word* para *doubleword*, estendendo o sinal de AX para DX. A sintaxe é:

```
CWD
```

Note que essa é uma instrução sem operandos, e deve ser usada antes da operação IDIV, caso o dividendo só ocupe AX e o divisor seja uma palavra de 16 bits.

■ **Operações em formato byte:** analogamente ao visto no item anterior, caso o dividendo de uma divisão (composto por **AX**) ocupe apenas AL, AH deve ser preparado, pois é sempre considerado na execução da operação. Se a instrução for **DIV**, o registrador AH deve ser zerado, e se for **IDIV**, AH deve ter a extensão de sinal de AL. A instrução CBW faz a conversão byte para *word* e estende o sinal de AL para AH. A sintaxe é:

```
CBW
```

Também é uma instrução sem operandos e deve ser usada antes de IDIV quando o dividendo só ocupar AL e o divisor for uma palavra de 8 bits.

As instruções CWD e CBW não afetam os FLAGS.

## **Exemplos:**

1. Vamos criar um trecho de programa que divida -1250 por 7.

```
...  
MOV AX, -1250 ; AX recebe o dividendo  
CWD ; estende o sinal de AX para DX  
MOV BX, 7 ; BX recebe o divisor  
IDIV BX ; executa a divisão após a  
execução, AX recebe o quociente
```

e DX recebe o resto

...

2. Veremos agora como fica o trecho de programa que divide a variável sinalizada XBYTE por -7:

...	
MOV AL,XBYTE	; AL recebe o dividendo
CBW	; estende o sinal (eventual) de AL para AH
MOV BL, -7	; BL recebe o divisor
IDIV BL	; executa a divisão após a execução, AL recebe o quociente e AH recebe o resto
...	

# Entrada E Saída De Números Decimais

Já vimos, anteriormente, quando estudamos as instruções lógicas e de deslocamento, como ficariam as rotinas de entrada e saída de números hexadecimais e binários. Vamos agora mostrar as rotinas de entrada e saída de números decimais.

## ***Entrada de números decimais***

Vamos desenvolver uma rotina de entrada de números decimais, com as seguintes especificações:

- Devemos ter uma *string* de caracteres de números 0 a 9, fornecidos pelo teclado.
- CR deve ser o marcador de fim de *string*.
- AX deverá ser o registrador de armazenamento.
- Os valores decimais permitidos devem estar na faixa de -32768 a + 32767, ou seja, 16 bits sinalizados.
- O sinal negativo deve ser apresentado, se existir.

O algoritmo básico em pseudolinguagem para essa rotina ficaria:

```
total = 0
negativo = FALSO
ler um caractere
CASE caractere OF
    ‘ – ’ : negativo = VERDADEIRO e ler um caractere
    ‘ + ’ : ler um caractere
END_CASE
REPEAT
    converter caractere em valor binário
    total = 10 × total + valor binário
    ler um caractere
UNTIL caractere é um carriage return (CR)
IF negativo = VERDADEIRO
then total = - (total)
END_IF
```

Na linguagem *assembly* do 8086, o procedimento para essa rotina é:

```
ENTDEC      PROC
; Lê um número decimal da faixa de -32768 a +32767
; variáveis de entrada: nenhuma (entrada de dígitos
; pelo teclado)
; variáveis de saída: AX (valor binário
; equivalente do número decimal)
    PUSH BX
    PUSH CX
    PUSH DX          ; salvando registradores que
                      ; serão usados
    XOR BX,BX        ; BX acumula o total, valor
                      ; inicial 0
    XOR CX,CX        ; CX indicador de sinal
                      ; (negativo = 1), inicial = 0
    MOV AH,1h
    INT 21h          ; lê caractere no AL
    CMP AL,'-'       ; sinal negativo?
    JE MENOS
    CMP AL,'+'       ; sinal positivo?
    JE MAIS
```

```

JMP NUM          ; se não é sinal, então vá
                  ; processar o caractere

MENOS: MOV CX,1      ; negativo = verdadeiro
MAIS: INT 21h        ; lê um outro caractere
NUM:  AND AX,000Fh    ; junta AH a AL, converte
                      ; caractere para binário

PUSH AX          ; salva AX (valor binário)
                  ; na pilha

MOV AX,10          ; prepara constante 10
MUL BX            ; AX = 10 x total, total
                  ; está em BX

POP BX            ; retira da pilha o valor
                  ; salvo, vai para BX

ADD BX,AX          ; total = total x 10 + valor
                  ; binário

MOV AH,1h          ; lê um caractere
INT 21h            ; é o CR?
CMP AL,0Dh          ; se não, vai processar outro
JNE NUM            ; dígito em NUM

MOV AX,BX          ; se é CR, então coloca o
                  ; total calculado em AX

CMP CX,1            ; o número é negativo?
JNE SAIDA          ; não

NEG AX            ; sim, faz-se seu
                  ; complemento de 2

SAIDA:   POP DX
          POP CX
          POP BX          ; restaura os conteúdos
                  ; originais
          RET             ; retorna a rotina que
                  ; chamou

ENTDEC     ENDP

```

## **Saída de números decimais**

A rotina de saída de números decimais teria as seguintes especificações:

- AX deve ser o registrador de armazenamento.
- Os valores decimais devem estar na faixa de -32768 a +32767.
- O sinal negativo deve ser exibido, se o conteúdo de AX for negativo.
- Deverá ser exibida no monitor uma *string* de caracteres de números de 0 a 9, exibidos no monitor de vídeo.

O algoritmo em pseudolínguagem seria:

```
IF AX < 0 THEN exibe um sinal de menos substitui-se AX pelo
seu complemento de 2 END_IF
contador = 0
REPEAT
    dividir quociente por 10
    colocar o resto na pilha
    contador = contador + 1
UNTIL quociente = 0
FOR contador vezes DO
    retirar um resto (número) da pilha
    converter para caractere ASCII
    exibir o caractere no monitor
END_FOR
```

A ideia da técnica de decomposição decimal do número em AX, utilizando a pilha, pode ser vista na [Figura 15.9](#).

Passo			Pilha
5	2	dividido por 10=0 com resto 2	-> 0002h <- Topo
4	24	dividido por 10=2 com resto 4	-> 0004h
3	246	dividido por 10=24 com resto 6	-> 0006h
2	2461	dividido por 10=246 com resto 1	-> 0001h
1	24618	dividido por 10=2461 com resto 8	-> 0008h

**FIGURA 15.9** Técnica de decomposição decimal do número usando a pilha.

A técnica é simples: divide-se o número por 10 até o quociente ser 0, empilhando todos os restos. Os restos serão os algarismos que formam o número, empilhados do mais significativo (topo) para o menos significativo. Com isso, o número fica decomposto.

Em *assembly* 8086, o procedimento referente à rotina de saída de números decimais é:

```
SAIDEC      PROC
; exibe o conteúdo de AX como decimal inteiro com
; sinal
; variáveis de entrada:    AX (valor binário
; equivalente do número  ;(decimal
; variáveis de saída: nenhuma (exibição de
dígitos, direto no monitor de vídeo)
        PUSH AX
        PUSH BX
        PUSH CX
        PUSH DX      ; salva na pilha os
                      ; registradores usados
        OR AX,AX      ; prepara comparação de sinal
        JGE  PT1      ; se AX maior ou igual a 0, vai
                      ; para PT1
        PUSH AX      ; como AX menor que 0, salva o
                      ; número na pilha
        MOV DL,' - ' ; prepara o caractere ' - '
                      ; para sair
        MOV AH,2h      ; prepara exibição
        INT 21h      ; exibe ' - '
        POP AX      ; recupera o número
        NEG AX      ; troca o sinal de AX
                      ; (AX = - AX)obtendo dígitos
                      ; decimais e salvando-os
                      ; temporariamente na pilha
PT1:   XOR CX,CX      ; inicializa CX como contador
                      ; de dígitos
        MOV BX,10      ; BX possui o divisor
PT2:   XOR DX,DX      ; inicializa o byte alto do
                      ; dividendo em 0 restante é AX
```

```

        DIV BX      ; após a execução,
                    ; AX = quociente; DX = resto
        PUSH DX    ; salva o primeiro dígito
                    ; decimal na pilha (1º resto)
        INC CX     ; contador = contador + 1
        OR AX,AX   ; quociente = 0?
                    ; (teste de parada)
        JNE PT2    ; não, continuamos a repetir
                    ; o laço exibindo os dígitos
                    ; decimais (restos); no
                    ; monitor, na ordem inversa
        MOV AH,2h   ; sim, termina o processo,
                    ; prepara exibição dos restos
PT3:  POP DX    ; recupera dígito da pilha
                    ; colocando-o em DL (DH = 0)
        ADD DL,30h  ; converte valor binário do
                    ; dígito para caractere ASCII
        INT 21h    ; exibe caractere
        LOOP PT3   ; realiza o loop até que CX = 0
        POP DX     ; restaura o conteúdo dos
                    ; registros
        POP CX
        POP BX
        POP AX     ; restaura os conteúdos dos
                    ; registradores
        RET         ; retorna à rotina que chamou
SAIDEC    ENDP

```

Segue um exemplo de um programa que utiliza os procedimentos ENTDEC e SAIDEC.

```
TITLE PROGRAMA DE E/S DECIMAL
.MODEL SMALL
.STACK 100h
.DATA

MSG1 DB      'Entre um número decimal na faixa de
              -32768 a 32767:$>
MSG2 DB      0Dh,0Ah,>Confirmando, você digitou:$>
.CODE
INCLUDE:  C:\<diretorio_de_trabalho>\ENTDEC.ASM
INCLUDE:  C:\<diretorio_de_trabalho>\SAIDEC.ASM
PRINCIPAL PROC
        MOV AX,@DATA
        MOV DS,AX
        MOV AH,9h
        LEA DX,MSG1
        INT 21h
; entrada do número
        CALL ENTDEC ; chama procedimento ENTDEC
        PUSH AX      ; salva temporariamente o
                      ; número na pilha exibindo
                      ; a segunda mensagem
        MOV AH,9h
        LEA DX,MSG2
        INT 21h      ; exibe a segunda mensagem
                      ; exibindo o número lido
        POP AX      ; recupera o número na pilha
        CALL SAIDEC ; chama procedimento SAIDEC
; saída para o DOS
        MOV AH,4Ch
        INT 21h
PRNCIPAL ENDP
END PRINCIPAL
```



## O Que vem Depois

Agora temos de avançar e compreender estruturas mais complexas da linguagem. Iremos estudar as instruções de controle de fluxo, que permitem a utilização de estruturas mais complexas, como o laço (*loop*, em inglês).

---

## CAPÍTULO 16

# Instruções de controle de fluxo

---

### Objetivos O Capítulo

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Compreender a importância dos sinalizadores e sua utilização no controle do fluxo dos programas.
- Conhecer e utilizar os principais comandos em linguagem de montagem para controle de fluxos.



### Apresentação



Os comandos de controle de fluxo são recursos fundamentais utilizadas em todas as linguagens de programação. Podemos dizer que não se conseguiria fazer muito em linguagem de montagem sem a utilização destes recursos.

É com esses comandos que podemos definir se o fluxo do programa vai seguir esta ou aquela linha, e de acordo com as informações inseridas pela pessoa que estiver utilizando o programa, será executada uma ou outra sequência de comandos.

Primeiramente vamos aprender como é feita uma comparação e sua fundamentação. Uma vez entendida esta parte, introduziremos dois conceitos: saltos condicionais e incondicionais.

Por fim, faremos a implementação das principais estruturas de controle de fluxo encontradas em linguagens de alto nível, como Pascal, C, Java, entre outras.

## Fundamentos

### ***Para que instruções de salto?***

Os programas em linguagem de montagem precisam seguir uma determinada forma de tomar decisões e também repetir determinadas partes do código para que possam executar tarefas complexas. Vamos mostrar como isto pode ser feito utilizando instruções de salto.

As instruções de salto transferem o controle para outra parte do programa. Esta transferência pode ser incondicional ou pode depender de uma combinação particular dos *flags* de estado.

Há também as instruções de decisão e laço, implementadas em linguagem de alto nível, as quais podem ser escritas em linguagem de montagem, como mostra o exemplo a seguir.

Um exemplo de instruções de salto:

```
Title Exibicao De Caracteres Ascii  
.MODEL SMALL  
.STACK 100H  
.CODE  
MAIN PROC  
; inicialização de alguns registradores
```

MOV AH,2	; função DOS para exibição de caractere
MOV CX,256	; contador com o número total de caracteres
MOV DL,00H	; DL inicializado com o primeiro ASCII

```
;  
definição de um processo repetitivo de 256 vezes PRINT_LOOP:
```

INT 21H	; exibir caractere na tela
---------	----------------------------

INC DL	; incrementa o caractere ASCII
DEC CX	; decrementa o contador continua
JNZ PRINT_LOOP	; continua exibindo enquanto CX não for 0 quando CX = 0,o programa quebra a sequência do loop

```
;
; saída para o DOS
;
MOV AH,4CH
INT 21H
MAIN ENDP
END MAIN
```

Após a montagem, ligação e execução do código anterior, teríamos o seguinte resultado:

```
C:\Tasm>tasm ascii.asm
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland
International
Assembling file: ascii.asm
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 413k

C:\Tasm>tlink ascii.obj
Turbo Link Version 7.1.30.1. Copyright (c) 1987, 1996 Borland
International

C:\TASM>ascii
C:\TASM>
```

Na figura anterior, o retângulo indica o resultado da execução do programa: a exibição de todos os símbolos da tabela ASCII.

No caso considerado, a CPU executa JNZ PRINT\_LOOP verificando o *flag* ZF. Se ZF = 0, o controle é transferido para PRINT\_LOOP, se ZF = 1, o programa vai para a próxima instrução MOV AH,4CH.

## Instruções De Saltos Condicionais

Essas instruções devem atender uma determinada condição para que possam ser executadas.

As instruções de saltos condicionais têm a seguinte sintaxe:

```
JXXX rótulo_de_destino
```

Cabe notar que **XXX** é uma condição dependente de algum dos *flags* de estado.

Se a condição **XXX** for verdadeira, isto é, se for satisfeita, a próxima instrução a ser executada é aquela definida pelo rótulo\_de\_destino. Para isso, a UCP ajusta o registrador IP para que ele aponte a posição de memória dada por rótulo\_de\_destino. Se a condição **XXX** for falsa, isto é, não satisfeita, a próxima instrução é aquela que está imediatamente após a instrução de salto.

A faixa de endereçamento atingida pelas instruções de saltos condicionais é de 256 bytes, ou seja, o salto alcança no máximo 126 bytes antes da instrução de salto incondicional e 127 bytes depois dela. Isto já era de se esperar, uma vez que rótulo\_de\_destino tem 8 bits.

Existem três categorias de saltos condicionais:

- Saltos sinalizados
- Saltos não sinalizados
- Saltos de *flag* único

A [Tabela 16.1](#) nos mostra as instruções de saltos sinalizados, sua descrição e os *flags* responsáveis pela execução do salto.

---

### Tabela 16.1

## Saltos sinalizados

---

Saltos sinalizados		
Mnemônico	Descrição	Condições
JG ou JNLE	Salto se maior do que OU salto se não menor do que ou igual a	ZF = 0 e SF = OF
JGE ou JNL	Salto se maior do que ou igual a OU salto se não menor do que	SF = OF
JL ou JNGE	Salto se menor do que OU salto se não maior do que ou igual a	SF ≠ OF
JLE ou JNG	Salto se menor do que ou igual a OU salto se não maior do que	ZF = 1 ou SF ≠ OF

A seguir, a [Tabela 16.2](#) nos mostra as instruções de saltos não sinalizados, sua descrição e os *flags* responsáveis pela execução do salto.

---

**Tabela 16.2**  
**Saltos não sinalizados**

---

Saltos não sinalizados		
Mnemônico	Descrição	Condições
JA ou JNBE	Salto se acima do que OU salto se não abaixo do que ou igual a	ZF = 0 e CF = 0
JAE ou JNB	Salto se acima do que ou igual a OU salto se não abaixo do que	CF = 0
JB ou JNAE	Salto se abaixo do que OU salto se não acima do que ou igual a	CF = 1
JBE ou JNA	Salto se abaixo do que ou igual a OU salto se não acima do que	ZF = 1 ou CF = 1

As instruções de saltos de *flag* único são mostradas pela [Tabela 16.3](#), assim como sua descrição e os *flags* responsáveis pela execução do salto.

---

**Tabela 16.3****Saltos de flag único**

---

Saltos de flag único		
Mnemônico	Descrição	Condições
JE ou JZ	Salto se igual OU salto se igual a zero	ZF = 1
JNE ou JNZ	Salto se não igual OU salto se não igual a zero	ZF = 0
JC	Salto se há vai-um ( <i>carry</i> )	CF = 1
JNC	Salto se não há vai-um ( <i>not carry</i> )	CF = 0
JO	Salto se há overflow	OF = 1
JNO	Salto se não há overflow	OF = 0
JS	Salto se o sinal é negativo	SF = 1
JNS	Salto se o sinal é não negativo (+)	SF = 0
JP ou JPE	Salto se a paridade é PAR ( <i>even</i> )	PF = 1
JNP ou JPO	Salto se a paridade é ÍMPAR ( <i>odd</i> )	PF = 0

## A Instrução De Comparação (CMP)

Além das instruções aritméticas, as condições de salto são frequentemente geradas por comparação entre dois operandos; para isso, temos uma instrução cujo mnemônico é CMP (*compare*). A sintaxe desta instrução é:

CMP destino, fonte

Essa instrução compara o destino com a fonte ao realizar uma subtração entre os dois; porém, o resultado não é armazenado; apenas os *flags* são alterados.

A [Tabela 16.4](#) nos mostra as combinações de operando considerando destino-

fonte.

**Tabela 16.4**

### Possibilidades de comparação de dados – instrução CMP

Operando fonte	Operando destino	
	Registrador de dados	Memória
Registrador de dados	SIM	SIM
Memória	SIM	NÃO
Constante	SIM	SIM

Vamos agora considerar AX=7FFFh, BX=0001h, e também o seguinte trecho de código:

```
CMP AX,BX  
JG ALVO
```

O resultado de CMP AX,BX é 7FFFh-0001h = 7FFEh. O comando JG ALVO é um comando de salto (salto de maior do que – *jump if greater than*). Para tanto, ele verifica os *flags* ZF, SF e OF. Se todos forem iguais a ZERO (0), sua condição é verdadeira e ele redireciona o ponteiro de instrução para ALVO. No caso deste exemplo, ZF = SF = OF = 0, portanto, a próxima instrução a ser executada deverá estar em ALVO, ou seja, o salto será feito.

Na verdade, o que temos é que se AX é maior que BX o salto será feito e o fluxo do programa será transferido para ALVO; caso contrário, não.

Neste outro exemplo, vamos supor que AX e BX contenham os mesmos números do exemplo anterior. O que o código a seguir fará?

MOV CX,AX	; coloca o valor de AX em CX
CMP BX,CX	; BX é maior?
JLE NEXT	; se não, vá para NEXT
MOV CX,BX	; se não, coloque BX em CX!!

NEXT:...

Evidentemente, como a instrução JLE considera AX e BX ambos números sinalizados, a execução desse trecho fará com que o maior dos números seja colocado em CX. No exemplo, CX conterá o valor 7FFFh.

## Instrução De Salto Incondicional (JMP)

Além dos saltos condicionais, também temos uma instrução que, quando executada, faz com que um salto aconteça. Chamamos este salto de salto incondicional, cujo mnemônico é **JMP** e a sintaxe é:

JMP destino

O termo destino é um rótulo no mesmo segmento da instrução JMP. Não existe limitação de faixa de endereçamento dentro do segmento, como no salto condicional.

### Exemplo:

TOP:

```
;corpo do loop
DEC CX
JZ EXIT
JMP TOP
```

EXIT:

```
MOV AX,BX
```

## Estruturas De Linguagens De Alto Nível

Utilizando as instruções vistas até agora, vamos traduzir algumas estruturas complexas de uma linguagem de alto nível.

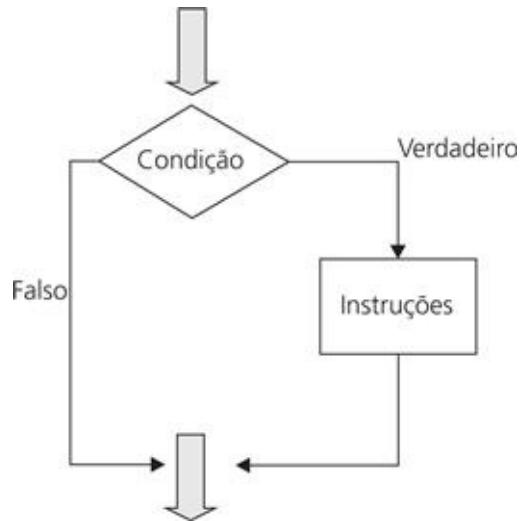
### ***Estrutura IF-THEN***

Em pseudocódigo (próximo a uma linguagem de alto nível) teríamos a seguinte estrutura para esse comando:

```
IF condição é verdadeira  
Then  
    Execute um conjunto de instruções  
END_IF
```

Essa estrutura nos diz que, se a condição for verdadeira, um conjunto de instruções será executado; caso contrário, não.

A [Figura 16.1](#) nos mostra o fluxo do comando IF-THEN.



**FIGURA 16.1** Comando IF-THEN.

Vamos fazer um trecho de programa, em linguagem de montagem do 8086, que substitua o número que está em AX pelo seu valor absoluto. Se fôssemos escrever um algoritmo em um pseudocódigo, teríamos:

```

IF AX < 0
THEN
    Replace AX by -AX
END_IF

```

Traduzindo para a linguagem de montagem:

; se AX < 0	
CMP AX,0	; AX < 0?
JNL END_IF	; não, saia do If
; then	
NEG AX	; sim, mude o sinal de AX
END_IF:	

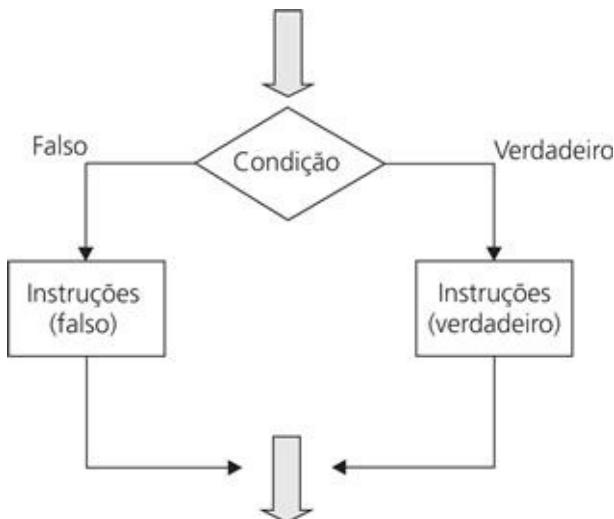
## Estrutura IF-THEN-ELSE

O trecho em pseudocódigo que representa esse comando é:

```
IF condição é verdadeira
THEN
    Execute um conjunto de instruções (verdade)
ELSE
    Execute outro conjunto de instruções (falso)
END_IF
```

Se a condição for verdadeira, um conjunto de instruções será executado, e se for falsa, outro conjunto de instruções será executado.

A [Figura 16.2](#) nos mostra o fluxo do comando IF-THEN-ELSE.



**FIGURA 16.2** Comando IF-THEN-ELSE.

Vamos supor, por exemplo, que AL e BL contenham caracteres ASCII. Queremos escrever um trecho de programa que mostre o menor caractere segundo a ordem alfabética.

O algoritmo em pseudocódigo ficaria:

```
IF AL <= BL
Then
    Mostre o caractere em AL
Else
    Mostre o caractere em BL
END_IF
```

E a tradução em linguagem de montagem seria:

```
MOV AH,2          ; preparando para exibir caractere
;se AL <= BL
    CMP AL,BL      ; AL <= BL?
    JNBE ELSE_
;then
    MOV DL,AL      ; move caractere a ser exibido
    JMP DISPLAY     ; vai para exibição
;else
    MOV DL,BL
ELSE_:
    MOV DL,BL
DISPLAY:
    INT 21h        ;exibe o caractere em DL
END_IF:
    ...
```

**Observação:** O rótulo ELSE\_ é usado, pois ELSE é uma palavra reservada.

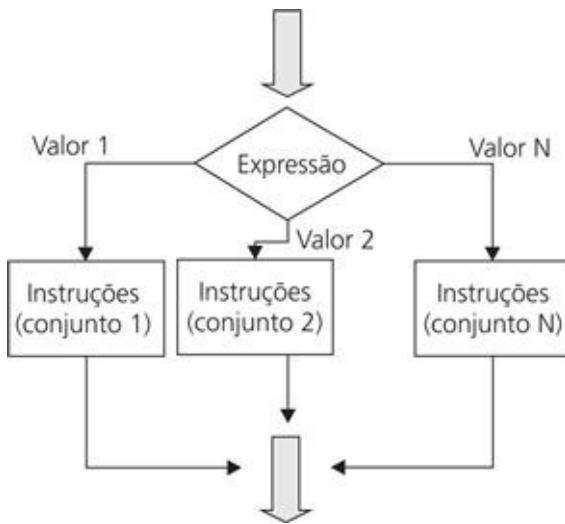
## Estrutura CASE

O comando CASE pode ser representado, em pseudocódigo, como:

```
CASE expressão
  Valor_1:conjunto1 de instruções
  Valor_2:conjunto2 de instruções
  Valor_3:conjunto3 de instruções
  ...
  Valor N: conjuntoN de instruções
END_CASE
```

Nessa estrutura, “expressão” é testada. Se o seu valor faz parte de um universo de valores (Valor\_1, Valor\_2, … , Valor\_N), então o respectivo conjunto de instruções será executado. Os outros conjuntos de instruções serão ignorados.

A [Figura 16.3](#) nos mostra o fluxo do comando CASE:



**FIGURA 16.3** Comando CASE.

Vamos supor que queremos colocar -1 em BX se AX tiver um valor negativo e colocar zero em BX, se AX tiver um valor igual a zero, e que também

desejamos colocar 1 em BX se AX tiver um valor positivo.

Podemos escrever o seguinte algoritmo para esse problema:

```
CASE AX  
<0:BX = -1  
=0:BX = 0  
>0:BX = 1  
END_CASE
```

O trecho de programa em linguagem de montagem seria:

; case AX	
CMP AX,0	; testa AX
JL NEGATIVO	; AX < 0
JE ZERO	; AX = 0
JG POSITIVO	; AX > 0
NEGATIVO:	
MOV BX,-1	; coloca -1 em BX pois AX negativo
JMP END_CASE	; e sai...
ZERO:	
MOV BX,0	; coloca 0 em BX pois AX = 0
JMP END_CASE	; e sai...
POSITIVO:	
MOV BX,1	; coloca 1 em BX pois AX positivo
END CASE:	
...	

**Observação:** Apenas um comando CMP é necessário, pois as instruções de salto (JXXX) não alteram os *flags*.

## **Estrutura de laço (LOOP)**

O laço em pseudocódigo pode ser escrito como:

```
FOR n vezes DO  
    Conjunto de instruções  
END_FOR
```

Traduzindo para a linguagem de montagem, vamos introduzir a instrução **LOOP**, cuja sintaxe é:

```
LOOP rótulo de destino
```

Essa instrução tem como contador implícito o registrador CX, que deve ser inicializado antes do laço. Desta forma, a instrução fará o salto para rótulo\_de\_destino, enquanto CX não for zero. Quando CX = 0, a próxima instrução após LOOP será executada.

**Observação:** O registrador CX é decrementado automaticamente quando a instrução LOOP é executada.

Vamos agora escrever um trecho de código em linguagem de montagem para mostrar uma linha com 80 asteriscos (“\*”).

Em algoritmo, utilizando pseudocódigo, ficaria:

```
FOR I=1 TO 80 DO  
    DISPLAY '*'  
END for
```

ou

```
FOR 80 TIMES DO  
    DISPLAY '*'  
END_FOR
```

A tradução para a linguagem de montagem é:

MOV CX,80	; número de asteriscos
MOV AH,2	; prepara para mostrar um caractere
MOV DL,'*'	; caractere a ser mostrado
TOPO:	
INT 21h	; mostra um asterisco
LOOP TOPO	; repete 80 vezes (vide CX)
...	

O valor de CX pode vir a sofrer alguma alteração. Caso o valor de CX mude para zero e depois disso o laço seja executado, o valor de CX será igual a FFFFh = 65535 após o primeiro comando, fazendo com que o laço seja executado por esta mesma quantidade de vezes.

Para prevenir tal fato, deve-se utilizar a instrução **JCXZ** (*jump if CX is zero*) antes do início do laço. Esta instrução testa o valor de CX e, se for zero, salta para o rótulo\_destino.

```
JCXZ SKIP  
TOPO:  
;corpo do laço
```

LOOP TOPO

SKIP:

...

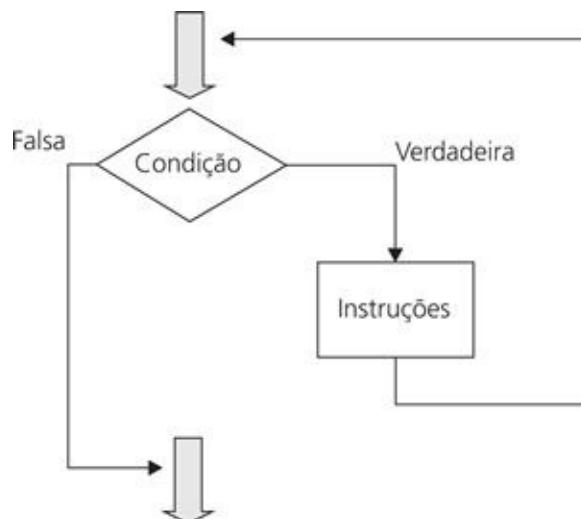
## Estrutura de laço WHILE

O comando WHILE pode ser representado, em pseudocódigo, como:

```
WHILE condição DO  
    Conjunto de instruções  
END_WHILE
```

A condição é verificada no início do laço. Se for verdadeira, o conjunto de instruções é executado. Se falsa, o programa vai para o que estiver abaixo do conjunto de instruções. O laço é executado enquanto a condição for verdadeira.

A [Figura 16.4](#) nos mostra o fluxo do comando WHILE:



**FIGURA 16.4** Comando WHILE.

Vamos agora escrever um código em linguagem de montagem para contar o número de caracteres digitados em uma linha. A condição de parada será a entrada da tecla <enter> (CR – *carriage return*).

O algoritmo desse programa pode ser representado assim:

```
Inicialize o contador com 0
Leia um caractere
WHILE (caractere <> CR) DO
    Armazenar caractere lido
    Contador = contador + 1
    Leia um caractere
END_WHILE
```

Em linguagem de montagem, o programa ficaria:

...	
MOV DX,0h	; inicialização
MOV AH,1h	
INT 21h	
; while	
LOOP_: CMP AL,0Dh	; é o caractere CR?
JE FIM	; salto quando caractere é igual a CR
MOV AL, (algum lugar)	; salvando o caractere lido
INC DX	; conta número de caracteres
INT 21h	; é outro caractere
JMP LOOP_	; fecha o loop WHILE
;end_while	
FIM:	

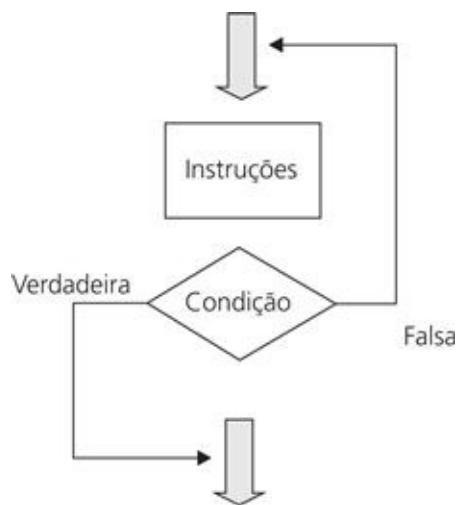
## Estrutura de laço REPEAT

O comando REPEAT pode ser escrito em pseudocódigo como:

```
REPEAT
    Conjunto de instruções
    UNTIL condição
```

A condição é verificada no final do laço. Se for falsa, o conjunto de instruções é executado novamente. Se verdadeira, o programa continua executando os comandos abaixo da verificação da instrução. O laço é executado enquanto a condição for falsa.

A [Figura 16.5](#) nos mostra o fluxo do comando REPEAT:



**FIGURA 16.5** Comando REPEAT.

Para exemplificar a tradução do comando REPEAT, vamos escrever um código em linguagem de montagem para contar o número de caracteres digitados em uma linha, até que a tecla <enter> seja digitada (CR – carriage return).

Em pseudocódigo, isso fica:

```
Initialize o contador com 0  
REPEAT  
    Leia um caractere  
    Armazenar caractere lido  
    Contador = contador + 1  
UNTIL (caractere = CR)
```

Em linguagem de montagem:

```
MOV DX, 0h          ; inicialização  
MOV AH, 1h
```

### ***Estrutura WHILE x Estrutura REPEAT***

Quando uma estrutura de repetição é necessária, a utilização de WHILE ou REPEAT é apenas uma questão de preferência pessoal.

A vantagem da estrutura WHILE é que o laço pode ser transposto se a condição de terminação for inicialmente falsa. Em contrapartida, se isso ocorrer na estrutura REPEAT o conjunto de instruções será executado ao menos uma vez. Cabe notar também que a estrutura REPEAT é geralmente menor que a estrutura WHILE.

### ***Estruturas com condições compostas***

#### **Utilizando O Operador AND**

É uma estrutura na qual um determinado conjunto de instruções será executado

se (e somente se) duas condições forem verdadeiras.

Condição1 AND condição2

Se quisermos escrever um caractere (como exemplo) que pertença somente ao alfabeto e esteja em CAIXA-ALTA (maiúscula), o algoritmo ficaria:

```
Leia um caractere (para AL)
IF ("A" <= caractere) AND (caractere <= "Z")
    THEN
        Mostre-o
    END_IF
```

O trecho de programa em linguagem de montagem ficaria assim:

```
;ler um caractere
    MOV AH,1      ;prepara para leitura
    INT 21h       ;lê e coloca em AL
;se ('A' <= char) E (char <= 'Z')
    CMP AL,'A'   ;char >= 'A'
    JNGE END_IF  ;não, saia!
    CMP AL,'Z'   ;char <= 'Z'
    JNLE END_IF  ;não, saia!
;then mostrar o caractere
    MOV DL,AL    ;pegue o caractere
    MOV AH,2      ;prepare para mostrar um caractere
    INT 21h      ;mostra o caractere
END_IF:
...
...
```

## Utilizando O Operador OR

É uma estrutura na qual um determinado conjunto de instruções será executado, se pelo menos uma das duas condições for verdadeira:

Condição1 OR condição2
------------------------

Vamos escrever um programa que leia um caractere, por exemplo, com as seguintes especificações: se ele for igual a “y” ou “Y”, mostre-o. Do contrário, saia do programa.

Em algoritmo ficaria:

```

Leia um caractere (para AL)
IF (caractere = "y") OR (caractere = "Y")
    THEN
        Mostre-o
    ELSE
        Termine o programa
END_IF

```

A tradução para a linguagem de montagem seria:

; ler um caractere	
MOV AH,1	; prepara para leitura
INT 21h	; lê e coloca em AL
; se (char = 'y'	OU (char = 'Y')
CMP AL,'y'	; char = 'y'
JE THEN	; sim, mostre-o!
CMP AL,'Y'	; char = 'Y'
JE THEN	; sim, mostre-o!
JMP ELSE_	; não, termine o programa
THEN:	
MOV DL,AL	; pegue o caractere
MOV AH,2	; prepare para mostrar um caractere
INT 21h	; mostra o caractere
JMP END_IF	; e saia do IF
ELSE_:	
MOV AH,4Ch	
INT 21h	; sair para o DOS
END_IF:	
...	



## O Que Vem Depois

Bem, agora que já sabemos como construir programas um pouco mais complexos, com controle de fluxo e domínio das estruturas de controle de linguagens de alto nível, chegou a hora de nos aprofundarmos nas estruturas de dados mais complexas como pilha, vetores e matrizes. Com isso vamos obter um controle mais efetivo dos dados que estamos manipulando ao longo da execução de nossos programas em linguagem de montagem. Além disto, com a utilização de procedimentos, aprenderemos a modular os programas.

---

## CAPÍTULO 17

---

# Pilha, procedimentos, macros, vetores, matrizes e modos de endereçamento

---

### Objetivos do capítulo

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Manipular a estrutura de pilha disponível na linguagem de montagem do processador 8086.
- Modular os programas de forma eficiente e elegante, utilizando os procedimentos.
- Definir estrutura de dados mais complexas, como os vetores e matrizes.



### Apresentação

O que acontecerá se você tentar pegar o último livro da base de uma pilha de livros? Certamente, a pilha perderá a sustentação e cairá. Para que isso não ocorra, é necessário desempilhar todos os livros até chegar ao desejado.



Em programação, os dados também podem ser organizados em pilhas para que o seu acesso seja facilitado por meio de instruções específicas. Neste capítulo aprenderemos a manipular pilhas, estruturar vetores e matrizes e utilizar a programação modulada.

## Fundamentos

Como vimos em outros capítulos, a pilha é uma estrutura de dados unidimensional, na qual as informações podem ser inseridas e retiradas segundo a ordem do “último que entrou é o primeiro a ser retirado” (*last in first-out*). O topo da pilha é a localização do último elemento inserido nela. A linguagem de montagem do processador 8086 propicia a utilização de uma pilha de dados de maneira fácil, pois esta linguagem permite que utilizemos as instruções específicas de armazenamento e a retirada de dados desta pilha. Durante a execução do programa, a pilha é utilizada para o armazenamento temporário de dados e endereço, principalmente na chamada e retorno de um procedimento.

Os procedimentos (sub-rotinas) são trechos de programas que executam uma determinada função e são definidos fora do programa principal. A chamada para que este procedimento seja executado ocorre no programa principal, em outro procedimento ou no próprio procedimento (chamadas recursivas). A utilização de procedimentos na programação é de extrema importância, não só na programação em alto nível como também na linguagem de montagem, pois ela permite a modulação de programas. A modulação, por sua vez, faz com que tenhamos uma programação elegante, um código de menor tamanho e ainda facilita a verificação e correção de erros.

## Organização Da Pilha

Antes de estudarmos as instruções de manipulação de pilha, vamos ver como podemos utilizá-la com o montador TASM. Para isso utilizaremos a diretiva **.STACK**, pela qual um segmento de memória pode ser alocado para a pilha. A declaração que o programa deve conter é:

```
.STACK 100h ; aloca um bloco de memória para a pilha
```

Essa declaração faz com que o SS (*stack segment*) tenha, após a montagem do programa, o endereço base do segmento de pilha. Já o registrador SP (*stack pointer*) será usado pelas instruções de manipulação da pilha para indicar o *offset* desse segmento, e corresponderá ao topo da pilha. Portanto, o endereço lógico do topo da pilha será dado pelo par SS:SP. É importante salientar que a pilha “cresce” do endereço maior para o endereço menor. Nos exemplos de utilização das instruções, isto ficará mais claro.

## **Instruções de manipulação da pilha**

Temos dois tipos de instruções específicas de manipulação da pilha. Uma permite que um item seja armazenado na pilha, o qual será representado pelo mnemônico PUSH, e a outra permite que um item seja retirado da pilha, o qual será representado pelo mnemônico POP.

## **Instruções de armazenamento na pilha**

As instruções de armazenamento de um item na pilha são as instruções **PUSH** e **PUSHF**.

As instruções do tipo **PUSH** têm a seguinte sintaxe:

```
PUSH fonte
```

Nessa instrução, a fonte é um operando de 16 bits, e pode ser apenas um registrador ou uma palavra de memória do tipo DW.

A execução do PUSH apresenta as seguintes ações:

- O registrador SP é decrementado de 2, uma vez que a pilha armazena palavras de 16 bits, ou seja, 2 bytes.
- Uma cópia do conteúdo do operando fonte é armazenado na pilha de forma que a posição SS:SP armazene o byte menos significativo do operando fonte e a posição SS:(SP + 1) armazene o byte mais significativo.
- O conteúdo da fonte não é alterado e não altera nenhum dos *flags*.

A outra instrução para o armazenamento de uma informação na pilha é a instrução **PUSHF**, que tem a seguinte sintaxe:

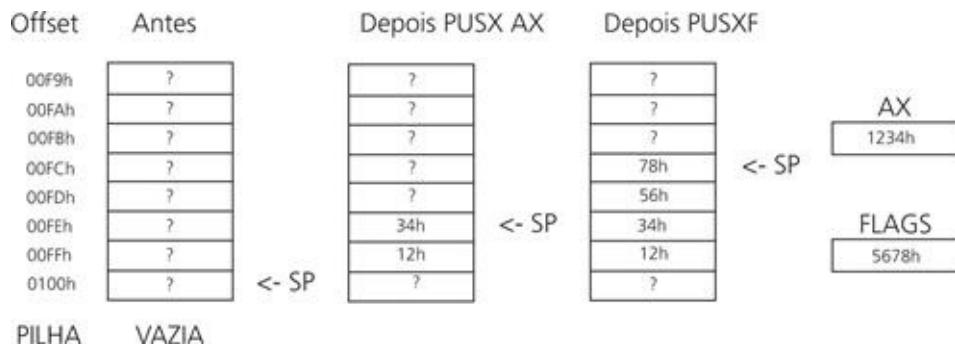
PUSHF

A instrução PUSHF não possui operando explícito. Ela é usada para armazenar, no topo da pilha, o conteúdo do registrador de *flags*. Sua operação é análoga à da instrução PUSH, exceto por aquilo que a instrução PUSHF armazena na pilha: o registrador de *flags*.

Suponha que AX contenha o valor 1234h e que o registrador de *flag* tenha armazenado 5678h. Quais valores serão armazenados na pilha após a execução de cada uma das instruções seguintes?

PUSH AX	; armazena AX no topo da pilha
PUSHF	; armazena o FLAG no topo da pilha

A [Figura 17.1](#) nos mostra a solução. O que podemos observar é que inicialmente a pilha está vazia e SP aponta para o offset 0100h. Após a execução da primeira instrução, o valor 1234h é armazenado e o valor de SP é atualizado para o novo topo. Como vimos no [Capítulo 12](#), o SP tem de ser atualizado ( $SP = SP - 2$ ) antes do armazenamento do dado. Como a organização da memória é em bytes, a palavra de 16 bits é armazenada da seguinte maneira: o byte menos significativo fica na posição de maior endereço e o mais significativo fica alocado na posição de menor endereço. Após a execução da segunda instrução, o conteúdo do *flag* é armazenado no novo topo da pilha.



**FIGURA 17.1** Passos da execução de instruções PUSH e PUSHF.

## Instruções de retirada na pilha

As instruções de retirada de um item da pilha são as instruções **POP** e **POPF**.

As instruções do tipo **POP** têm a seguinte sintaxe:

POP destino

Análogo à instrução PUSH, nesta instrução, destino é um operando de 16 bits, podendo apenas ser um registrador ou uma palavra de memória do tipo DW.

A execução de POP apresenta as seguintes ações:

- O conteúdo das posições SS:SP (byte menos significativo) e SS:(SP + 1) (byte mais significativo) é movido para o destino.
- O registrador SP (*stack pointer*) é incrementado de 2, determinando o novo topo da pilha.
- Nenhum dos bits de FLAG é alterado.

Outra instrução para retirada de um item da pilha é a instrução **POPF**, que tem a seguinte sintaxe:

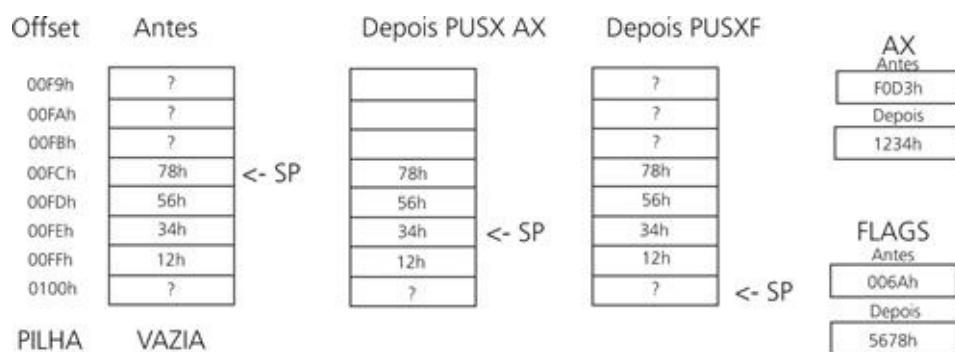
POPF

A instrução POPF, assim como a instrução PUSHF, não possui operando explícito. Ela é usada para retirar do topo da pilha a palavra armazenada e colocá-la no registrador de *flags*. Sua operação é análoga à da instrução POP, com a exceção de que aqui o destino será o registrador de *flag*.

Suponha que a pilha tenha armazenado duas palavras 1234H e 5678h; esta última, no seu topo. Suponha ainda que tenha em AX o número F0D3h e no *flag* o número 006Ah. Quais valores seriam armazenados nos registradores após a execução de cada uma das instruções seguintes? E como ficaria a pilha?

POPF	; retira do topo da pilha e armazena no FLAG
POP AX	; retira do topo da pilha e armazena no AX

A [Figura 17.2](#) nos mostra o que acontece. Inicialmente, o SP aponta para o offset 00FCh, que é o topo da pilha e tem armazenado o número 5678h. Após a execução da primeira instrução, esse valor fica armazenado no *flag* e o valor de SP é atualizado para o novo topo ( $SP = SP + 2$ ). Após a execução da segunda instrução, temos o conteúdo do novo topo armazenado em AX e SP apontando para a última posição da pilha, que é vazia. Portanto, apesar de os valores estarem na memória, até que algo seja escrito em cima da pilha ela é considerada vazia, uma vez que não temos como desempilhar os valores.



**FIGURA 17.2** Passos da execução de instruções POP e POPF.

Neste outro exemplo vemos um programa que utiliza a pilha para escrever

uma *string* invertendo a sua leitura:

```
TITLE ENTRADA INVERTIDA
.MODEL SMALL
.STACK 100h
.CODE

MOV AH,2      ; exibe o prompt para o usuário
MOV DL,'?'   ; caractere '?' para a tela
INT 21h       ; exibe
XOR CX,CX    ; inicializando contador de caracteres em zero
MOV AH,1      ; prepara para ler um caractere do teclado
INT 21h       ; caractere em AL

; while caractere não é <CR> do

INICIO: CMP AL,0DH ; é o caractere <CR>?
JE PT1        ; sim, então saindo do loop

; salvando o caractere na pilha e incrementando o contador

PUSH AX       ; AX vai para a pilha (interessa somente AL)
INC CX        ; contador = contador + 1 lendo um novo caractere
INT 21h       ; novo caractere em AL
JMP INICIO    ; retorna para o inicio do loop ; end while
PT1: MOV AH,2 ; prepara para exibir
MOV DL,0DH    ; <CR>
INT 21h       ; exibindo
MOV DL,0AH    ; <LF>
INT 21h       ; exibindo: mudança de linha
JCXZ FIM      ; saindo se nenhum caractere foi digitado for contador vezes do
TOPO: POP DX  ; retira o primeiro caractere da pilha
INT 21h       ; exibindo este caractere
LOOP TOPO     ; em loop até CX = 0 ; end for

FIM: MOV AH,4CH ;preparando para sair para o DOS
INT 21H
END
```

## Definição e utilização de procedimentos

O procedimento é definido pelas diretivas **PROC** e **ENDP**. A diretiva **PROC**

serves para definir o nome do procedimento e o seu início, enquanto a **ENDP** define o fim do procedimento.

A definição de um procedimento segue a sintaxe seguinte:

```
nome PROC tipo  
    ; instruções do procedimento  
    RET ; instrução de retorno, que transfere o controle de  
    volta para a rotina principal  
    nome ENDP
```

O *nome* do procedimento é qualquer símbolo válido que não seja palavra reservada. O *tipo* define se o procedimento está definido no mesmo segmento de código do local de chamada (tipo **FAR**) ou em segmentos distintos (tipo **NEAR**).

## Instruções De Chamada E Retorno De Procedimentos

### ***Instrução de chamada de procedimento***

Para que um procedimento seja executado, ele precisa ser invocado por uma instrução. A instrução CALL faz com que o controle de execução do programa passe para o procedimento.

A sintaxe dessa instrução é:

```
CALL nome
```

Como vimos no item anterior, nome é qualquer símbolo válido que não seja palavra reservada, que foi definido pela diretiva PROC. Quando da execução dessa instrução, o registrador IP, que contém o offset do endereço da próxima instrução à chamada (ou seja, o endereço da instrução depois do CALL), é armazenado na pilha. Depois disso, o IP recebe o offset do endereço da primeira instrução do procedimento chamado para que a rotina seja executada. Esta

instrução não afeta nenhum dos *flags*.

## Instrução de retorno de procedimentos

A última instrução de um procedimento deve ser a instrução de retorno de procedimento, pois o controle de execução do programa deve retornar a quem chamou o procedimento – justamente a função da instrução **RET**. A sintaxe dessa instrução é:

```
RET
```

O retorno para quem chamou significa que o IP deverá ter o endereço da instrução que vem depois do CALL. Lembremos que o *offset* é empilhado pela instrução CALL, portanto o RET deverá desempilhar o *offset* e carregá-lo em IP. Esta instrução não afeta nenhum dos *flags*.

A [Figura 17.3](#) mostra o mecanismo de chamada de um procedimento, e a [Figura 17.4](#) mostra o mecanismo de seu retorno. Podemos notar que durante a execução do CALL a pilha está vazia, e o IP aponta para a próxima instrução (*offset* 1012h) depois da instrução CALL. Após a execução do CALL, o endereço de retorno (*offset* 1200h) é empilhado no topo da pilha, e IP aponta para a primeira instrução do procedimento (*offset* 1200h). Isto fará com que a próxima instrução a ser executada seja a primeira instrução do procedimento.



**FIGURA 17.3** Mecanismo de chamada de um procedimento.



**FIGURA 17.4** Mecanismo de retorno de um procedimento.

A Figura 17.4 nos mostra o mecanismo de retorno de um procedimento. Antes da execução da instrução RET, temos no registrador IP o offset dessa instrução, e no topo da pilha o endereço de retorno (1012h – offset da primeira instrução depois da instrução CALL). Após a execução do RET, o conteúdo do topo da pilha é carregado em IP (offset 1012h), devolvendo o controle de execução “para quem chamou”. A pilha fica vazia.

## Passagem De Parâmetros Em Procedimentos

Como fazer a passagem de parâmetros? Ou melhor, como podemos fornecer os dados necessários para a execução do procedimento e enviar os dados calculados por este procedimento para “quem” o chamou? Na linguagem de montagem isso é feito utilizando os registradores de propósito geral ou até mesmo a pilha.

O trecho de programa a seguir exemplifica a utilização de um procedimento com o programa principal. Trata-se de um procedimento que calcula a multiplicação de dois números, utilizando operações de soma e deslocamento.

```

TITLE MULTIPLICACAO POR SOMA E DESLOCAMENTO
.MODEL SMALL
.STACK 100h
.CODE
PRINCIPAL PROC
... ; supondo a entrada de dados
CALL MULTIPLICA

```

```

... ; supondo a exibição do resultado
MOV AH,4Ch ; retorno ao Sistema Operacional
INT 21h
PRINCIPAL ENDP
MULTIPLICA PROC NEAR
;multiplica dois números A e B por soma e deslocamento
;entradas: AX = A, BX = B, números na faixa 00h - FFh
;saída: DX = A*B (produto)
PUSH AX

        PUSH BX      ; salva os conteúdos de AX e BX
        AND DX,0     ; inicializa DX em 0
; repeat
; if    B é' impar
TOPO: TEST BX,1   ; B é impar?
        JZ PT1       ; não, B é par (LSB = 0) then
        ADD DX,AX   ; sim, então produto = produto + A
                    end_if
PT1:  SHL AX,1   ; desloca A para a esquerda 1 bit
        SHR BX,1   ; desloca B para a direita 1 bit
                    until
        JNZ TOPO   ; fecha o loop repeat
        POP BX
        POP AX      ; restaura os conteúdos de BX e AX
        RET         ; retorno para o ponto de chamada

MULTIPLICA ENDP
END PRINCIPAL

```

## Macros

Outra técnica utilizada na modularização de programas são as *macros*. Como os procedimentos, as macros também são trechos de programa que realizam uma determinada função. Mais especificamente, falando em linguagem de montagem, podemos definir uma macro como um bloco de texto que recebe um nome especial e que contém instruções, diretivas, comentários ou referências a outras macros.

A macro é chamada no momento da montagem e é expandida, ou seja, o montador copia o bloco de texto na posição de cada chamada de macro. As expansões podem ser vistas no arquivo .LST, gerado pelo TASM.

As macros são utilizadas para “criar novas instruções” que operacionalizem tarefas frequentes e repetitivas. A [Tabela 17.1](#) mostra a comparação do uso da macro com o uso do procedimento. Utilizando macros, o tempo de montagem é maior, pois a cada chamada de macro o montador as substitui pelas respectivas instruções; porém, ao se utilizar a chamada de procedimentos, isto não é feito. A consequência disto é que o código gerado com macros é maior que aquele gerado com procedimentos, mas em compensação teremos mais acessos na pilha (memória) se utilizarmos os procedimentos, tornando a execução mais lenta.

---

**Tabela 17.1**  
**Comparação entre macros e procedimentos**

---

	<b>Macros</b>	<b>Procedimentos</b>
Tempo para montagem do programa	Maior	Menor
Tamanho de código de máquina (.EXE)*	Maior	Menor
Tempo de execução menor	Menor	Maior
Utilização	Pequenas funções	Grandes funções

**Observação (\*):** Na maioria das vezes para funções análogas.  
A sintaxe da definição de uma macro é:

```
nome_da_macro MACRO p1,p2,p3,...pn  
instruções  
ENDM
```

Os termos p1,p2,p3..pn **são argumentos da macro**, e são opcionais.

Considere agora a criação de uma macro que mova uma palavra de uma posição de memória para outra. Já sabemos que não há uma instrução que faça isso.

Assim, a macro seria MOVW, definida como:

```
MOVW    MACRO      WORD1, WORD2  
        PUSH       WORD1  
        POP        WORD2  
        ENDM
```

A chamada dessa macro seria:

```
MOVW  PALAVRA1, PALAVRA2
```

Onde PALAVRA1, PALAVRA2 são *offset* no segmento de dados do tipo DW.  
Note-se que poderíamos também utilizar essa macro para mover uma palavra  
de um registrador de 16 bits para outro.

Por exemplo:

```
MOVW  AX, BX
```

O que acontece é que WORD1 e WORD2 são substituídos pelos argumentos  
da chamada: no primeiro caso, por duas posições de memória, e no segundo, por  
dois registradores. Em ambas chamadas os argumentos precisam ser de 16 bits  
por causa das instruções da macro, PUSH e POP.

## Diretiva INCLUDE

Para incluir outro arquivo no arquivo do programa que será montado, usamos a diretiva:

```
INCLUDE caminho\nome_do_arquivo_texto.
```

## Vetores e matrizes (arrays)

### Vetores Unidimensionais

Podemos definir um vetor unidimensional como uma lista ordenada de elementos do mesmo tipo. Por ordem entendemos que existe um primeiro elemento, um segundo, um terceiro e assim por diante, até o último elemento. Nas aulas de matemática, por exemplo, você já deve ter se deparado com um vetor de n elementos, no qual os elementos são identificados por:

```
A[1], A[2], A[3], A[4], A[5].. A[n];
```

Na expressão anterior, A é o nome do vetor, e os números dentro dos colchetes são os índices. Portanto, se quisermos acessar o segundo elemento do vetor A, é só utilizarmos A[2]. Na maioria das linguagens de alto nível, em um vetor de n posições o primeiro elemento teria o índice 0, e o último, o índice (n – 1).

Na linguagem de montagem, os vetores têm de ser definidos na área de dados. A seguir, mostraremos as diversas definições desse tipo de estrutura:

### Definição De Uma Cadeia De Caracteres (String)

```
MSG DB 'abcde' ; vetor composto por uma string de 5 caracteres ASCII
```

MSG é o *offset* dentro do segmento de dados (considerando 00FFh), e o armazenamento dessa *string* na memória ficaria:

	Offset de endereço	Endereço simbólico	Conteúdo
MSG ->	00FFh	MSG	10h
	0100h	MSG+1	10h
	0101h	MSG+2	20h
	0102h	MSG+3	10h
	0103h	MSG+4	30h

**FIGURA 17.5** Conteúdo de memória referente à *string* MSG.

### *Definição De Um Vetor De Elementos Do Tipo Word*

```
W DW 1010h, 1020h, 1030h ; vetor de 3 valores de 16 bits
```

W é o *offset* (supondo 0200h) dentro do segmento de dados, e o armazenamento desse vetor na memória ficaria:

	Offset de endereço	Endereço simbólico	Conteúdo
W ->	0200h	W	10h
	0201h		10h
	0202h		20h
	0203h		10h
	0204h		30h
	0205h		10h

**FIGURA 17.6** Conteúdo de memória referente ao vetor W.

## Operador DUP

O operador DUP pode ser utilizado na definição de vetores, quando queremos repetir algumas subestruturas. Os exemplos a seguir mostram a correta utilização deste operador.

```
GAMA DB 100 DUP (0) ; cria um vetor de 100
bytes, cada byte
```

inicializado com o valor zero, a partir do offset GAMA

```
BETA W 200 DUP (?) ; cria um vetor de 200
words (16 bits), não
inicializados, a partir
do offset BETA
```

; operadores DUP encadeados

LINHA DB 5, 4, 3 DUP (2, 3 DUP (0), 1) ; equivalente a definição de LINHA dada abaixo: LINHA DB 5, 4, 2, 0, 0, 0, 1, 2, 0, 0, 0, 1, 2, 0, 0, 0, 1
---

## Matrizes ou vetores bidimensionais

Uma matriz ou um vetor bidimensional é uma estrutura logicamente organizada em linhas e colunas, apesar de ser fisicamente considerada um vetor unidimensional.

Para uma determinada matriz A, o acesso aos elementos organizados em linha e coluna é dado por **A[i,j]**; A é o nome da matriz, i representa a linha, e j corresponde à coluna do elemento a ser acessado.

Considere uma matriz 3 X 4. A [Figura 17.7](#) mostra como ela seria:

MATRIZ	A [1,1]	A [3,2]	A [1,3]	A [1,A]
A	A [2,1]	A [2,2]	A [2,3]	A [2,4]
	A [3,1]	A [3,2]	A [3,3]	A [3,4]

**FIGURA 17.7** Matriz A (representação).

A definição das matrizes em linguagem de montagem é feita no segmento de dados, conforme mostraremos no exemplo a seguir.

Considere agora a matriz A ( $3 \times 4$ ) seguinte, inicializada com valores mostrados na [Figura 17.8](#).

	1	2	3	4
1	10	20	30	40
2	50	60	70	80
3	90	100	110	120

**FIGURA 17.8** Matriz A.

A definição na linguagem de montagem do 8086, se a organização for por linha, é:

...		
.DATA		
A	DW	10, 20, 30, 40
	DW	50, 60, 70, 80
	DW	90, 100, 110, 120
...		

E se a organização for por coluna:

...		
.DATA		
A	DW	10,50,90
	DW	20,60,100
	DW	30,70,110
	DW	40,80,120
...		

## ***Modos de endereçamento***

A forma como um operando é especificado em uma instrução é conhecida como modo de endereçamento, por meio do qual se mostra aquilo que se deve fazer para acessar o operando. Temos nove modos de endereçamento, os quais estudaremos a seguir:

### **Modo De Endereçamento Por Registrador**

Neste modo, os operandos são um dos registradores da UCP. Por exemplo:

```
ADD AX, BX
```

Nesse caso, os dados estão armazenados nos registradores AX e BX. Para acessá-los, basta que o processador leia os dois registradores.

### Modo De Endereçamento Utilizando Um Imediato

Aqui o operando é uma constante que está explicitamente definida na instrução, ou seja, disponibilizada imediatamente. Por exemplo:

```
MOV AX, 5
```

O valor 5 é definido na própria instrução. Uma vez realizado o *fetch* da instrução, este valor estará disponível para a execução da instrução.

### Modo De Endereçamento Direto

Neste modo, um dos operandos é uma variável declarada, ou seja, uma posição de memória com endereço determinado.

```
DADO DB 5
.....
MOV AL, DADO
```

A variável DADO (*offset* dentro do segmento de dados) é inicializada como 5. Quando a instrução MOV é executada, o conteúdo do endereço físico dado para o DS:DADO é armazenado em AL, ou seja, o valor 5 é armazenado em AL.

## Modo De Endereçamento Indireto Por Registrador

Neste modo de endereçamento, o *offset* do operando está em um registrador e, portanto, atua como ponteiro para a posição de memória.

Os únicos registradores que podem ser utilizados para armazenar o *offset* nesse modo de endereçamento são:

- BX, SI, DI, juntamente com o registrador de segmento DS, formando o endereço físico com o par DS:[registrador]. Lembre-se que, quando utilizamos os colchetes, estamos querendo dizer que o conteúdo do operando dentro dos colchetes é um apontador (*offset*) para a memória.
- BP, juntamente com o registrador de segmento SS, cujo endereço físico é formado por SS:[BP].

Os exemplos a seguir nos mostram como utilizar esse modo de endereçamento:

### Exemplo 1:

Vamos supor que o conteúdo de SI seja 0100h e que a palavra contida na posição de memória de *offset* 0100h seja 1234h. Os comentários mostram o que acontece com o registrador SI após a execução das respectivas instruções.

```
MOV AX,SI ;AX recebe 0100h
```

Nesse caso, temos o endereçamento por registrador, portanto, o conteúdo de SI será armazenado em AX.

```
MOV AX,[SI] ;AX recebe 1234h
```

Aqui, como temos os colchetes, o modo é o indireto por registrador, ou seja, o conteúdo de SI é o *offset* do dado a ser armazenado em AX.

### **Exemplo 2:**

Escreva um trecho de programa que acumule em AX a soma dos 10 elementos do vetor LISTA, pré-definido.

```
LISTA    DW      10,20,30,40,50,60,70,80,90,100
        ...
        XOR AX,AX ; inicializa AX com zero
        LEA SI,LISTA ; SI recebe o offset de end.
                      de LISTA
        MOV CX, 10 ; contador inicializado nº
                      de elementos
SOMA:   ADD AX,[SI] ; acumula AX com o elemento
                      de LISTA apontado por SI
        ADD SI,2 ; movimenta o ponteiro para
                  o próximo elemento de
                  LISTA (que é do tipo DW)
        LOOP SOMA ; faz o laço até CX = 0
```

### **Modo De Endereçamento Por Base**

No modo de endereçamento por base, o *offset* do operando é obtido adicionando um deslocamento ao conteúdo de um registrador base, BX ou BP.

O deslocamento pode ser um *offset* definido por uma variável, uma constante (positiva ou negativa), ou um *offset* definido por uma variável mais ou menos uma constante. A seguir, apresentamos os formatos possíveis:

- [registrar + deslocamento] ou [deslocamento + registrar]
- [registrar] + deslocamento ou deslocamento + [registrar]
- deslocamento [registrar]

Os registradores que podem ser utilizados nesse modo de endereçamento são:

- BX (*base register*) juntamente com o registrador de segmento DS.
- BP (*base pointer*) juntamente com o registrador de segmento SS.

A seguir temos exemplos de utilização desse modo de endereçamento.

### **Exemplo 1:**

Suponha que BX tenha o valor 4d e que o vetor lista é definido como:

```
LISTA DW 10h, 20h, 30h, 40h, 50h, 60h, 70h, 80h, 90h, 100h
```

A execução da instrução:

```
MOV AX, [LISTA + BX] ;resulta que AX = 30h
```

Ela faz com que o conteúdo do endereço físico dado por DS e (LISTA + 4) seja armazenado em AX, ou seja, o número 30h. Note que, ao somarmos 4 à LISTA, estamos acessando o quinto e sexto bytes (word), visto que a memória é organizada em bytes, conforme mostra a [Figura 17.9](#).

	Endereço simbólico	Conteúdo
LISTA ->	LISTA+0	10h
		00h
	LISTA+2	20h
		00h
	LISTA+4	30h
		00h
	LISTA+6	40h
		00h
	LISTA+8	50h
		00h
	LISTA+10	60h
		00h
	LISTA+12	70h
		00h
	LISTA+14	80h
		00h
	LISTA+16	90h
		00h
	LISTA+18	00h
		01h

**FIGURA 17.9** Vetor LISTA na memória.

### **Exemplo 2:**

Escreva um trecho de programa que acumule em AX a soma dos 10 elementos do vetor LISTA definido previamente, usando endereçamento por base.

```

LISTA DW 10h,20h,30h,40h,50h,60h,70h,80h,90h,100h
...
XOR AX,AX           ; inicializa AX com zero
XOR BX,BX          ; limpa o registrador base
MOV CX, 10          ; contador inicializado
                     ; nº de elementos
SOMA: ADD AX,LISTA+[BX] ; acumula AX com o
                         ; elemento de LISTA
                         ; apontado por offset

```

```

                     ; de LISTA + BX
ADD BX,2            ; incrementa base
LOOP SOMA           ; faz o laço até CX = 0
...

```

### **Exemplo 3:**

O acesso à pilha sem alteração do valor do SP, ou seja, sem a necessidade de empilhar ou desempilhar qualquer elemento, pode ser feito utilizando o modo de endereçamento por base e o registrador BP. Vamos escrever um trecho de programa que carregue AX, BX e CX com as três palavras mais superiores da pilha, sem modificá-la.

...	
MOV BP,SP	; BP aponta para o topo da pilha
MOV AX, [BP]	; coloca o conteúdo do topo em AX
MOV BX, [BP+2]	; coloca a 2ª palavra (abaixo do topo) em BX

MOV CX, [BP+4]	; coloca o 3ª palavra em CX
...	

## Modo De Endereçamento Indexado

Analogamente ao endereçamento por base, nesse caso o *offset* do operando é obtido adicionando-se um deslocamento ao conteúdo de um registrador indexador, SI ou DI.

O deslocamento pode ser o *offset* definido por uma variável, uma constante (positiva ou negativa), ou o *offset* definido por uma variável acrescido ou diminuído de uma constante. A seguir, temos os formatos possíveis:

- [registraror + deslocamento] ou [deslocamento + registraror]
- [registraror] + deslocamento ou deslocamento + [registraror]
- deslocamento [registraror]

Os registradores utilizados só podem ser o SI e o DI, juntamente com o registrador de segmento DS.

A seguir temos alguns exemplos de utilização desse modo de endereçamento:

### Exemplo 1:

Suponha que SI contenha o *offset* de LISTA, definido a seguir:

```
LISTA DW 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
```

Para acessar um elemento de LISTA, podemos:

LEA SI,LISTA	;SI recebe o offset de LISTA
MOV AX, [SI + 12]	;resulta que AX = 70h

Nesse caso, na primeira instrução, SI recebe o *offset*, indicado por LISTA, dentro do segmento de dados (similar à instrução **MOV SI, offset LISTA**). A segunda instrução pegará a palavra armazenada em (LISTA+12) (0070h,c conforme [Figura 17.3](#)) e guardará em AX.

## Exemplo 2:

Fazer uma rotina que converta os caracteres da *string* definida em MSG para maiúsculos.

```

MSG DB      'isto e uma mensagem'
...
        MOV CX,19          ; inicializa nº de
                           ; caracteres de MSG
        XOR SI,SI          ; SI = 0
TOPO:   CMP MSG[SI], ' ' ; compara caractere com
                           ; branco
        JE PULA            ; igual, não converte
        AND MSG[SI], 0DFh   ; diferente, converte
                           ; para maiúscula
PULA:   INC SI            ; incrementa indexador
        LOOP TOPO          ; faz o laço até CX = 0
        ...

```

## Modo De Endereçamento Por Base Indexado

Com o modo de endereçamento por base indexado (*Based Indexed Mode*) podemos obter o *offset* do operando somando o conteúdo de um registrador de base (BX ou BP) com o conteúdo de um registrador índice (SI ou DI), e opcionalmente com o *offset* representado por uma variável ou uma constante (positiva ou negativa). Portanto, os possíveis formatos são:

- variável [reg\_de\_base] [reg\_índice]
- variável [reg\_de\_base + reg\_índice + constante]
- constante [reg\_de\_base + reg\_índice + variável]
- [reg\_de\_base + reg\_índice + variável + constante]

Nesse modo, podemos apenas utilizar os registradores: SI, DI e BX juntamente com o registrador de segmento DS e SI, DI e o registrador BP juntamente com o registrador de segmento SS.

A seguir, alguns exemplos de utilização:

### Exemplo 1:

Considerando a variável LISTA definida a seguir e admitindo que SI tenha 14 e BX tenha 2:

```
LISTA DW 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
```

```
MOV AX, LISTA [BX][SI] ; resulta que  
AX = LISTA + 2 + 14 = 90
```

A execução da instrução anterior faz com que o conteúdo do segmento de dados apontado pelo offset LISTA + 2 + 14 seja armazenado em AX.

## Exemplo 2

Suponha que A seja uma matriz 3X4 com elementos de tipo DW. Escreva um trecho de programa que zere os elementos da segunda linha de A.

```
...
MOV BX,8          ; BX aponta para o 1º
                  ; elemento da segunda
                  ; linha
MOV CX,4          ; CX contém o número de
                  ; colunas
XOR SI,SI          ; inicializa o indexador
                  ; de coluna

LIMPA: MOV A [BX][SI], 0 ; carrega zero no
                         ; operando calculado
    ADD SI,2          ; incrementa 2 em SI,
                      ; pois é word
    LOOP LIMPA        ; faz o laço até que
                      ; CX seja zero
...
...
```

## Observações

### 1. Acesso de dados em outro segmento

Para acessar dados de um segmento em outro (*segment override*), basta que utilizemos a notação completa de endereço, especificando o endereço base pelo registrador de segmento. Por exemplo:

```
MOV AX, ES:[SI]
```

Pode-se utilizar *segment override* nos modos de endereçamento indireto por registrador, por base e indexado.

## 2. Diretiva PTR

A instrução MOV [BX],1 é ilegal, pois o montador não pode determinar, por si só, se [BX] aponta para uma informação na memória do tipo byte ou do tipo *word*, uma vez que o imediato pode ser representado com 8 ou 16 bits. Para resolver isto, utilizamos a diretiva PTR, da seguinte maneira:

```
MOV BYTE PTR [BX],1 ;define o destino como byte  
MOV WORD PTR [BX],1 ;define o destino como word
```

## 4. Diretiva LABEL

LABEL é uma diretiva que serve para alterar o tipo de variáveis, por exemplo:

Tempo	Label	Word
Horas	DB	10
Minutos	DB	20

Essa estrutura, declarada no segmento de dados, permite que:

- TEMPO e HORAS recebam o mesmo endereço pelo montador.
- TEMPO (16 bits) englobe HORAS e MINUTOS (8 + 8 bits).

Portanto, são legais as seguintes instruções:

MOV AH, HORAS	
MOV AL, MINUTOS	
MOV AX, TEMPO	; produz o mesmo efeito das anteriores

## 5. A instrução XLAT

É uma instrução sem operando, que converte um valor (tipo *byte*) em outro valor (tipo *byte*), utilizando uma tabela de conversão. O valor a ser convertido tem de estar em AL, e o *offset* do endereço base da tabela de conversão tem de estar em BX. A execução da instrução XLAT fará com que o conteúdo de AL seja somado ao *offset* dado por BX, resultando em uma posição na tabela. O conteúdo desta posição será armazenado em AL.

Por exemplo, converter o conteúdo de AL, supondo um número binário, no seu correspondente hexadecimal, representado por um caractere.

.DATA	
Tabela 16. DB	30h, 31h, 32h, 33h, 34h, 35h, 36h, 37h, 38h, 39h
DB	41h, 42h, 43h, 44h, 45h, 46h
...	
.CODE	
...	
MOV AL, 0Ch	; exemplo, converter 0Ch para caractere ASCII 'C'
LEA BX, TABELA 16.	; BX recebe o offset de TABELA 16.
XLAT	; é feita a conversão e AL recebe 43h = 'C'
...	



## Que Vem Depois

Vimos aqui as operações com pilha e a utilização de procedimento, tópicos imprescindíveis na prática da boa programação em linguagem de montagem. Se

você praticar, irá se tornar um profundo conhecedor e um bom programador de linguagem de montagem. Nos próximos capítulos veremos outro tipo de processador: os microcontroladores.

---

## CAPÍTULO 18

---

# A família PIC®

---

## Objetivos do capítulo

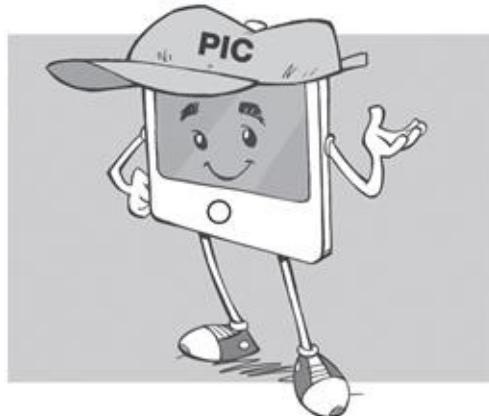
Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Entender a diferença entre microcontroladores e processadores de uso geral.
- Aprender os fundamentos da programação da família PIC®.



## Apresentação

Quando precisamos de um processador agregado a alguns dispositivos especiais, como os sensores, devemos utilizar os microcontroladores, que, justamente por terem esses dispositivos integrados junto ao processador, proporcionam uma maior facilidade nos projetos.

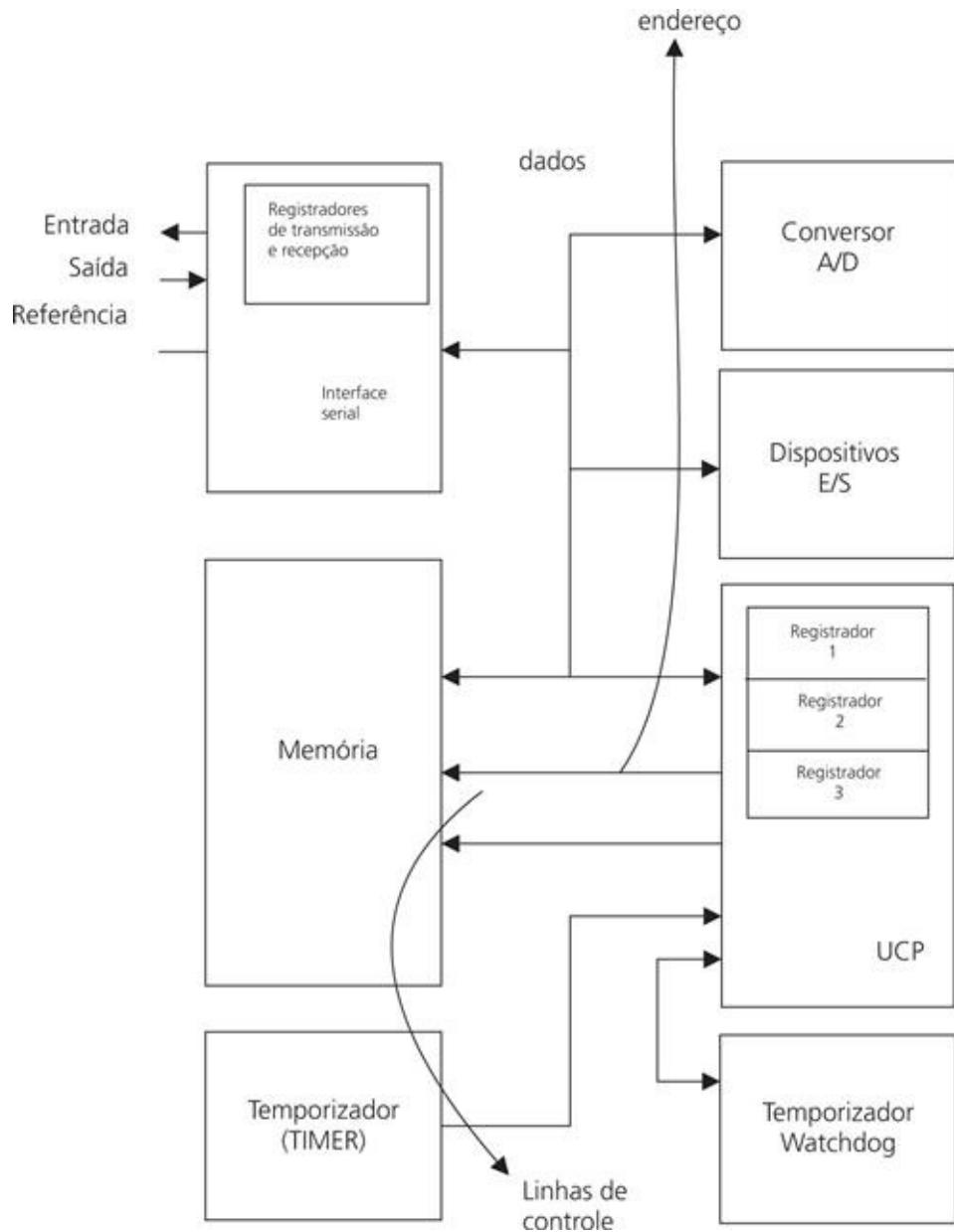


Neste capítulo estudaremos o microcontrolador PIC, um microcontrolador simples útil para a aprendizagem e um dos mais utilizados em projetos de sistemas de controle.

# Fundamentos

## **Microcontroladores**

O microcontrolador (MCU – *Micro Controller Unit*) é um circuito formado por uma unidade central de processamento (UCP), uma unidade de entrada e saída, uma unidade de memória e outras unidades específicas integradas em sua estrutura. Estas unidades específicas são circuitos que interagem com o meio externo, e alguns exemplos são os conversores analógicos/digitais (conversores A/D), os conversores digitais/analógicos (conversores D/A), as interfaces de entrada e saída de dados, os temporizadores e os sensores em geral. A unidade de memória é composta por três partes: uma específica para armazenamento de dados, uma para armazenamento de programas e uma para armazenamento permanente de dados. Os microcontroladores geralmente são utilizados em sistemas embarcados, ou seja, sistemas em que é necessário um controle inteligente e uma interface com sensores e atuadores para controle do sistema. Exemplos destes sistemas são os aparelhos da linha branca (refrigeradores, lavadoras de louça e de roupas etc.) com controle digital, que disponibilizam várias funções. A [Figura 18.1](#) nos mostra a estrutura básica de um microcontrolador.

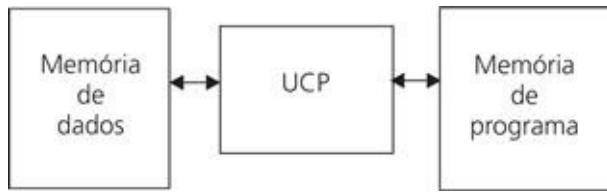


**FIGURA 18.1** Estrutura básica de um microcontrolador. Adaptação PIC 16F84 Data Sheet.

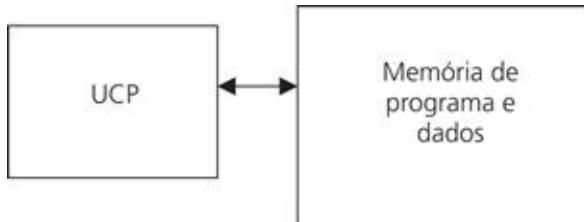
## A Família De Microcontroladores PIC®

Os microcontroladores da família PIC® (*Programmable Interface Controller – PIC®*), fabricados pela Microchip® Technology Inc., são micro-controladores que trabalham com palavras de dados de 8, 16 e 32 bits; eles utilizam a arquitetura de Harvard ([Figura 18.2](#)). A arquitetura de Harvard difere da arquitetura de von Neumann ([Figura 18.3](#)) quanto ao armazenamento de dados e programas: a

primeira tem uma memória de dados separada da memória de instruções, e a segunda tem apenas uma memória para instruções e dados. Além disso, os PICs são processadores RISC (*Reduced Instruction Set Computer*), ou seja, são processadores que têm um conjunto reduzido de instruções simples e utilizam a técnica de *pipeline* para a execução paralelizada de instruções. Um processador com *pipeline* de instruções é formado basicamente por unidades independentes que têm a função de executar (separadamente) cada etapa do ciclo de execução da instrução, permitindo assim a paralelização da execução de instruções.



**FIGURA 18.2** Arquitetura de Harvard.



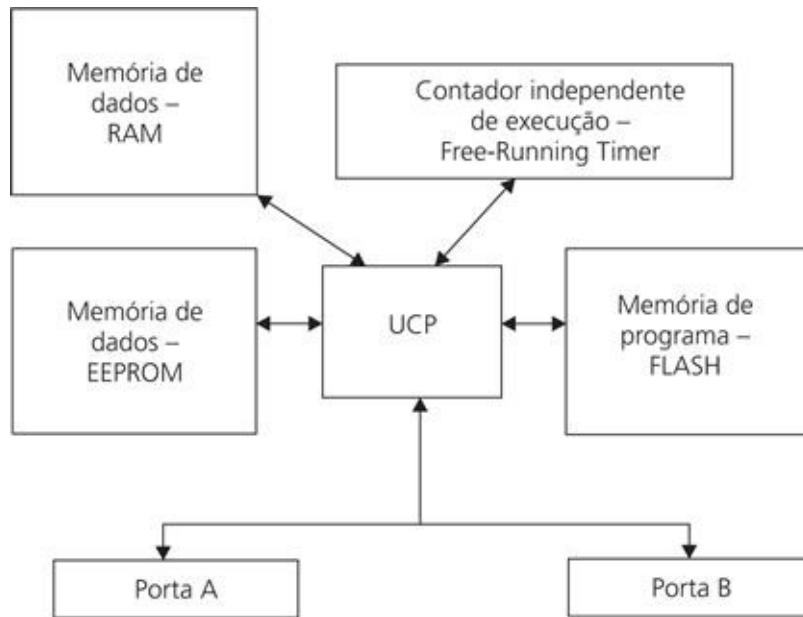
**FIGURA 18.3** Arquitetura de von Neumann.

Alguns PICs podem ser programados mais de uma vez, utilizando a memória *flash* ou a memória EEPROM, e outros apenas uma vez, que são os chamados PICs OTP (*One Time Programmable*). Os microcontroladores desta família trabalham com núcleos de processamento de 12, 14 e 16 bits.

## Microcontrolador PIC 16F84

Como estudo de caso, consideraremos o PIC 16F84, um dos microcontroladores mais utilizados, pela sua simplicidade e funcionalidade. Trata-se de um microcontrolador de 8 bits que estruturalmente é composto de uma UCP RISC, uma memória de programa tipo *flash*, uma memória de dados tipo RAM, outra

memória de dados tipo EEPROM e duas portas de entrada e saída ([Figura 18.4](#)).



**FIGURA 18.4** Estrutura do microcontrolador PIC 16F84.

Para entendermos melhor seu funcionamento, vamos voltar à [Figura 18.1](#), para explicar cada bloco que o compõe:

- **Unidade Central de Processamento (UCP):** como em qualquer sistema, a UCP é responsável pela execução das instruções e controle das demais unidades.
- **Sistema de memória:** é utilizado para o armazenamento de instruções e dados. O PIC 16F84 tem uma memória de programa tipo FLASH utilizada para o armazenamento dos programas, uma memória tipo EEPROM para o armazenamento dos dados que necessitam ser preservados quando a tensão de alimentação é desligada e uma memória RAM utilizada para armazenamento temporário de dados necessários à execução dos programas.
- **Unidades de E/S:** são utilizadas para a comunicação com o mundo externo, a Porta A (PORTA) e a Porta B (PORTB).
- **Interface serial:** permite a comunicação serial com qualquer dispositivo que adote esse protocolo de comunicação.
- **Conversor A/D:** unidade responsável pela conversão de um sinal analógico para digital.
- **Temporizador (Timer):** contador cujo conteúdo é incrementado em uma

unidade, com um intervalo de tempo pré-fixado. Serve, por exemplo, para fornecer o tempo decorrido entre dois eventos.

- **Watchdog:** É também um contador que está continuamente incrementado à medida que um programa está sendo executado. O programa zera este contador para indicar que está sendo executado normalmente; se por acaso o programa parar de ser executado por algum motivo, o contador não vai ser zerado, e assim o microcontrolador será reinicializado automaticamente, fazendo com que o programa seja executado novamente, sem a necessidade da intervenção humana.

A [Figura 18.4](#) mostra essa estrutura.

Como vimos anteriormente, esse microcontrolador é composto por: uma memória tipo *flash*, utilizada para o armazenamento dos programas que serão executados; uma memória EEPROM para armazenamento de dados, que devem ser mantidos mesmo após o desligamento do microcontrolador, e finalmente por uma outra memória de dados, tipo RAM, utilizada pelo programa durante a sua execução. Além disto, existe um contador independente de execução (*Free-Running Timer*), um registrador de 8 bits que trabalha independente do programa e é incrementado a cada quatro ciclos de *clock*, uma unidade central de processamento que executa os programas e duas portas de entrada e saída: portas A e B.

Esses microcontroladores são utilizados em sistemas embarcados para aplicações automobilísticas, controle de equipamentos industriais, dispositivos de segurança e aparelhos da linha branca, entre outros.

Na [Figura 18.5](#) podemos ver a pinagem do circuito integrado PIC 16F84.

1	RA2	RA1	18
2	RA3	RA0	17
3	RA4 / TOCK1	OSC1	16
4	MCLR	OSC2	15
5	V <sub>ss</sub>	V <sub>dd</sub>	14
6	RB0 / INT	RB7	13
7	RB1	RB6	12
8	RB2	RB5	11
9	RB3	RB5	10

**FIGURA 18.5** Pinagem do microcontrolador PIC 16F84.

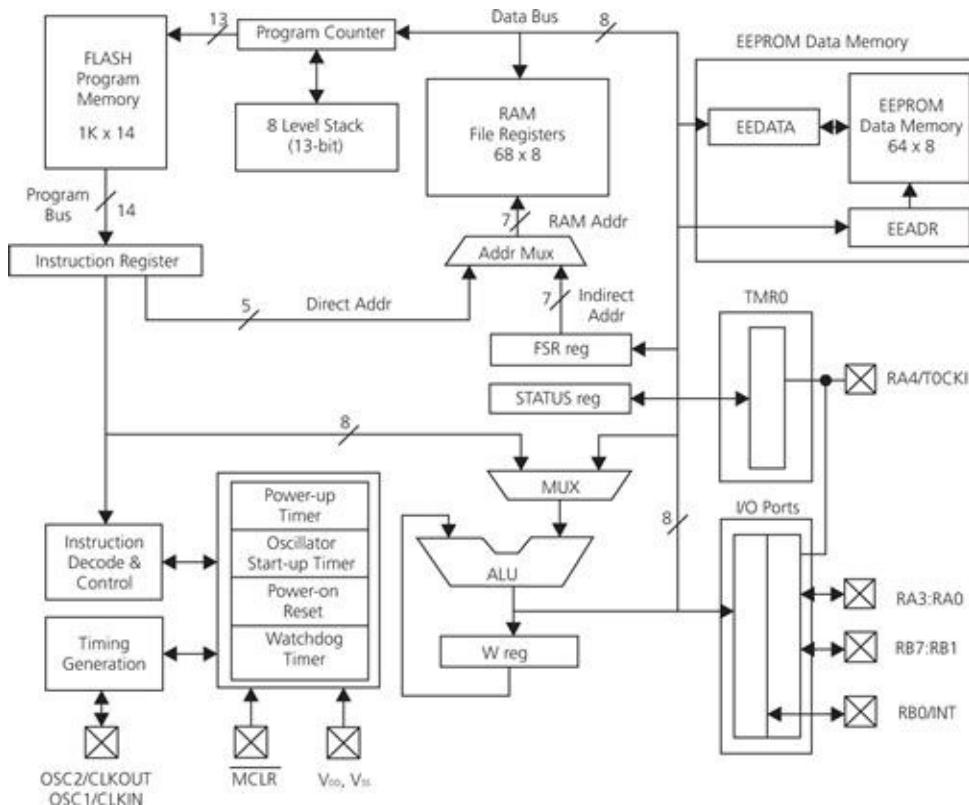
A seguir, temos a função de cada pino:

- **Pino 1 (RA2):** terceiro pino da porta de E/S A.
- **Pino 2 (RA3):** quarto pino da porta de E/S A.
- **Pino 3 (RA4):** quarto pino da porta de E/S A. Tem como função adicional TOCK1, que funciona como um temporizador.
- **Pino 4 (MCLR):** entrada de reset; tem como função adicional a tensão de programação V<sub>pp</sub> do microcontrolador.
- **Pino 5 (conexão de terra):** V<sub>ss</sub> da fonte de alimentação.
- **Pino 6 (RB0), pino Zero da porta B:** entrada para sinal de interrupção.
- **Pino 7 (RB1):** primeiro pino da porta de E/S B.
- **Pino 8 (RB2):** segundo pino da porta de E/S B.
- **Pino 9 (RB3):** terceiro pino da porta de E/S B.
- **Pino 10 (RB4):** quarto pino da porta de E/S B.
- **Pino 11 (RB5):** quinto pino da Porta de E/S B.
- **Pino 12 (RB6):** sexto pino da Porta de E/S B. Funciona também como a linha de *clock* no modo de programação.
- **Pino 13 (RB7):** sétimo pino da porta de E/S B. Funciona também como a linha de dado no modo de programação.
- **Pino 14:** tensão positiva V<sub>dd</sub> da fonte de alimentação.
- **Pino 15 (OSC2):** pino para a conexão de um oscilador.
- **Pino 16 (OSC1):** pino para a conexão de um oscilador.
- **Pino 17 (RA0):** primeiro pino da Porta A.

■ Pino 18 (RA1): segundo pino da Porta A.

## Arquitetura Do Microcontrolador PIC 16F84

Na [Figura 18.6](#) podemos ver o diagrama de blocos do PIC 16F84 e sua ligação com os pinos de entrada e saída. Nesta seção iremos detalhar as funções de alguns componentes desta estrutura, necessárias para a sua programação.



**FIGURA 18.6** Diagrama de blocos do PIC 16F84. Microchip Technology Inc., PIC16F84A Data Sheet 18-pin Enhanced FLASH/EEPROM 8-bit Microcontroller

## Organização De Memória Do PIC 16F84

No PIC 16F84 temos duas memórias, a memória para o programa e a memória para os dados. Estas memórias são distintas e cada uma tem um barramento próprio, permitindo, assim, acessos simultâneos às duas.

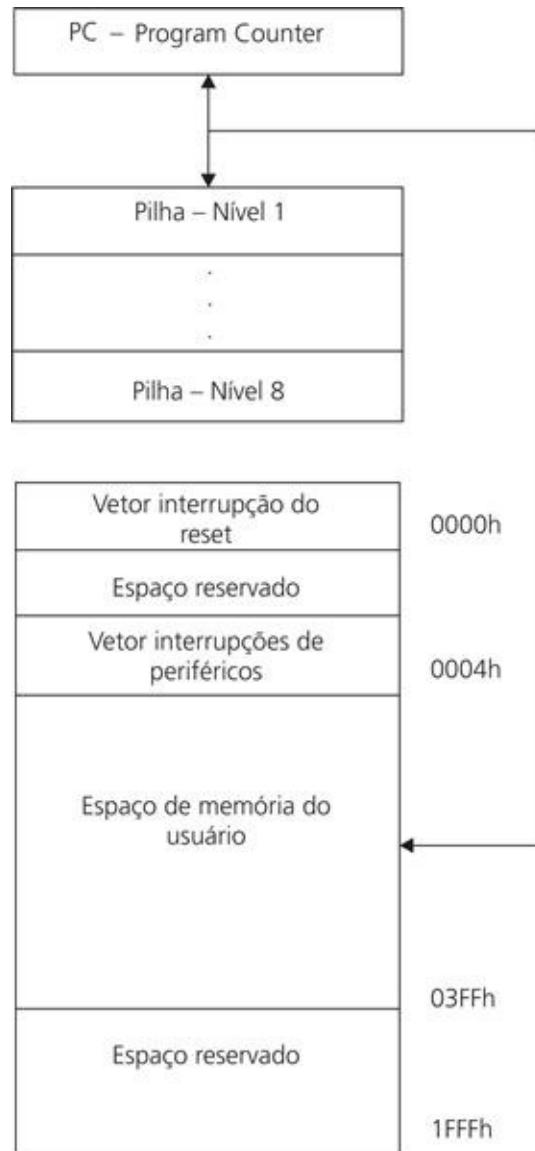
A memória de dados tipo RAM pode ser dividida em duas partes, a área de registradores de propósito geral (**GPR** - *General Purpose Registers*) e os

registradores de função específica (**SFR** - *Special Function Register*). Os registradores específicos são usados no controle de periféricos. Cabe aqui lembrar que para os dados temos ainda a memória de dados EEPROM, com 64 bytes, a qual não pode ser mapeada diretamente nesta área de dados, só no modo indireto.

Para isso, é necessário um registrador apontador que especifica o endereço a ser acessado na memória EEPROM.

### ***Organização da memória de programa***

O registrador PC desse microcontrolador tem 13 bits e acessa um espaço de memória de 1K X 14 bits. Podemos observar esta organização na [Figura 18.7](#).



**FIGURA 18.7** Organização da memória de programa do PIC 16F84.

A localização 0000h é reservada para o endereço da rotina que trata a interrupção de RESET, e a 0004h é usada para os endereços das rotinas que tratam as interrupções de periféricos.

Na [Figura 18.7](#) a seguir podemos ainda ver uma pilha de oito posições que podem apenas ser acessadas pelas instruções de chamada e retorno de procedimentos.

### ***Organização da memória de dados***

Como vimos anteriormente, a memória de dados é dividida em registradores

específicos (SFR) e registradores de propósito geral (GPR).

Ambas as áreas são divididas em bancos, e por isso necessitam de mais um bit para endereçar o banco. Este bit de controle está localizado no registrador **STATUS**.

O bit 5 desse registrador (RP0) seleciona o banco. Se o bit for zero, mostra que o banco 0 está selecionado, e se for 1, trata-se então do banco 1. Cada banco tem 128 bytes. As 12 primeiras localizações de cada banco são reservadas para os registradores de função específica (SFR) e as demais para os registradores de propósito geral (GPR). Instruções específicas (MOVW e MOVF) são utilizadas para mover dados do registrador W para qualquer localização do banco de registradores, e vice-versa.

A memória de dados pode ser acessada usando o endereçamento absoluto de cada banco de registradores ou o endereçamento indireto pelo registrador FSR (*File Select Register*).

A Figura 18.8 mostra a disposição da memória de dados SFR e os respectivos bancos.

00h	INDF	INDF	80h
01h	TMR0	OPTION	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	83h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h
07h	-	-	87h
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2	89h
0Ah	PCLATH	PCLATH	8Ah
0Bh	INTCON	INTCON	8Bh
0Ch	68 BYTES	Mapeado no	8Ch
...			...
4Fh	GPR	Banco 0	Cfh

**FIGURA 18.8** Organização da memória de dados do PIC 16F84.

## Registradores De Função Específica - SFR

A seguir iremos conhecer cada um dos registradores de função específica (SFR) e sua respectiva função:

- INDF: Registrador para endereçamento indireto.

Na verdade, esse registrador não é implementado fisicamente. Quando acessamos o INDF, estamos realmente acessando a posição indicada pelo registrador FSR (***File Selection Register***), que atua como ponteiro para outra posição da memória.

■ **TMR0:** Registrador de contagem do temporizador 0.

É o registrador incrementador de 8 bits do temporizador 0. É incrementado a cada pulso na entrada RA4/T0CKI ou a cada ciclo de instrução no modo *timer*.

■ PCL: Parte menos significativa do PC.

■ **STATUS:** Registrador de estado da UCP.

O registrador STATUS armazena o estado da UCO após a ocorrência de um evento como uma operação lógica ou aritmética, ou ainda uma seleção de bancos, entre outros.

BIT 7	6	5	4	3	2	1	BIT 0
IRP	RP1	RPO	TO\	PDI\	Z	DC	C
Endereço – 03h, 83h							valor inicial: 00011XXX

**FIGURA 18.9** Registrador STATUS.

A função de cada um dos bits é:

■ Bit 7 – IRP: Seleciona os bancos de memória para endereçamento indireto

Para o PIC 16F84 deve ser mantido em zero. Disponível para leitura e escrita (R/W)

■ **Bit 6 (RP1) e bit 5 (RP0):** Seleciona os bancos de memória para

endereçamento direto. Para o PIC 16F84, RP1 deve ser mantido em zero.

Disponíveis para R/W.

RP1 RP0

00 – Banco 0 – 00h a 7Fh

0 1 – Banco 1 – 80h a FFh

■ **Bit 4 – TO\:** Sinaliza a ocorrência de *time-out*. É um sinal ativo-baixo.

Disponível para R.

É igual a 1 após a ligação (*power up*), ou quando da execução das instruções

CLRWDT ou SLEEP.

É igual a quando ocorre *time-out* no temporizador *watchdog*.

■ **Bit 3 – PD\:** Bit *power down*.

Disponível para R.

É igual a 1 após a ligação (*power up*) ou quando da execução da instrução CLRWDT.

É igual a 0 se foi executada a instrução SLEEP.

■ **Bit 2 – Z:** Sinalizador de zero.

Disponíveis para R/W.

É igual a 1 se ocorreu resultado zero gerado pela ULA.

É igual a 0 se ocorreu resultado diferente de zero gerado pela ULA.

■ **Bit 1 – DC:** Digit Carry/Borrow (vai-um/vem-um).

Disponível para R/W.

É igual a 1 se ocorreu vai-um do 3º para o 4º bit em uma operação de adição ou subtração.

É igual a 0 se ocorreu vai-um do 3º para o 4º bit.

■ **Bit 0 – Carry/Borrow:** (vai-um/vem-um).

Disponível para R/W.

É igual a 1 se ocorreu vai-um do 7º para o 8º bit em uma operação de adição ou subtração.

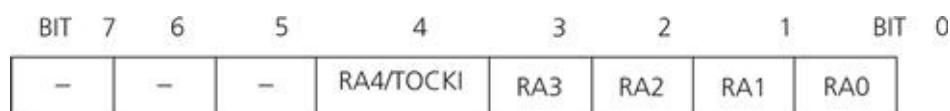
É igual a 0 se ocorreu vai-um do 7º para o 8º bit.

■ **FSR – File Selection Register**

Em um endereçamento indireto, atua como ponteiro para outra posição da memória.

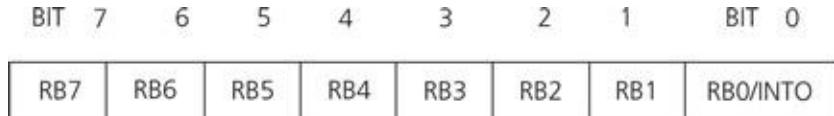
■ **PORTA:** Registrador referente à PORTA A.

Apenas os cinco bits menos significativos são usados e se referem às entradas/saídas da PORTA A, como mostra a [Figura 18.10](#).



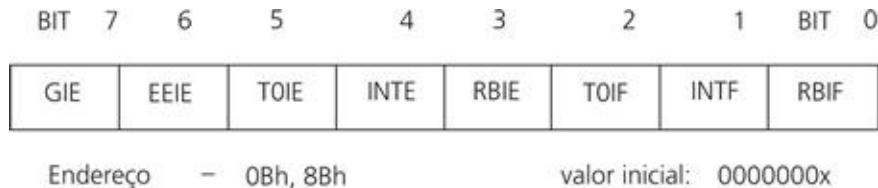
**FIGURA 18.10** Registrador PORTA.

- **PORTB:** Registrador referente à PORTA B.
    - Referem-se às entradas/saídas da PORTB, como mostra a [Figura 18.11](#).



### **FIGURA 18.11** Registrador PORTB.

- **EEDATA:** Armazena o dado lido/gravado na EEPROM.  
O dado lido ou a ser escrito na EEPROM é armazenado neste registrador.
  - **EEADR:** Endereço do dado a ser lido/gravado na EEPROM.  
O endereço da posição a ser acessada na EEPROM é armazenado neste registrador.
  - **PCLATH:** Parte mais significativa do PC.
  - **INTCON:** Habilita e desabilita as interrupções.



**FIGURA 18.12** Registrador INTCON.

A função de cada um dos bits é:

- **Bit 7 – GIE:** Desabilita todas as interrupções e permite habilitar as interrupções selecionadas. Disponível para R/W.
    - Se 1, habilita as interrupções selecionadas.
    - 0 – Desabilita todas as interrupções.
  - **Bit 6 – EEIE:** Interrupção de fim de escrita na EEPROM. Disponível para R/W.
    - Se 1, habilita a interrupção.
    - Se 0, desabilita a interrupção.
  - **Bit 5 – TOIE:** Habilita e desabilita a interrupção que indica OVERFLOW no TMR0. Disponível para R/W.

Se 1, habilita a interrupção.

Se 0, desabilita a interrupção.

- **Bit 4 – INTE:** Habilita e desabilita a interrupção externa através do pino RB0/INT. Disponível para R/W.

Se 1, habilita a interrupção.

Se 0, desabilita a interrupção.

- **Bit 3 – RBIE:** Habilita e desabilita a interrupção que indica alterações na PORTB. Disponível para R/W

Se 1, habilita a interrupção.

Se 0, desabilita a interrupção.

- **Bit 2 – T0IE:** Indica a ocorrência de overflow no TMR0. Disponível para R/W. Este bit tem de ser zerado pelo programa.

Se 1, ocorreu overflow.

Se 0, não ocorreu overflow.

- **Bit 1 – INTF:** Indica a ocorrência de uma interrupção externa através do pino RB0/INT. Disponível para R/W.

Se 1, ocorreu um pedido de interrupção no pino RB0/INT.

Se 0, não ocorreu um pedido de interrupção no pino RB0/INT.

- **Bit 0 – RBIF:** Indica a ocorrência de uma interrupção decorrente de alterações na PORTB. Disponível para R/W.

Se 1, houve mudança em um ou mais bits (RB7:RB0) da PORTB.

Se 0, não houve mudança nos bits (RB7:RB0) da PORTB.

- **OPTION:** Usado para configuração de temporizadores, escalas e outros.

BIT	7	6	5	4	3	2	1	BIT	0
RPBUA	INTEDG	TOCS	TOSE	PSA	PS2	PS1	PS0		

**FIGURA 18.13** Registrador OPTION.

Se 1, os *pull-ups* estão desabilitados.

Se 0, os *pull-ups* estão habilitados.

■ **Bit 6 – INTEDG:** Modo de reconhecimento dos sinais de interrupção.

Disponível para R/W.

1 – Reconhecimento na subida da transição do sinal no pino RBO/INT.

0 – Reconhecimento na descida da transição do sinal no pino RBO/INT.

■ **Bit 5 – T0CS:** Define a fonte do *clock* do temporizador 0. Disponível para R/W.

1 – Transição no pino RA4/T0CKI.

0 – Clock interno.

■ **Bit 4 – TOSE:** Define como o *clock* externo irá incrementar o temporizador 0. Disponível para R/W.

1 – Incremento na descida da transição do sinal no pino RA4/T0CKI.

0 – Incremento na subida da transição do sinal no pino RA4/T0CKI.

■ **Bit 3 – PSA:** Atribuição da escala. Disponível para R/W.

1 – Para o *watchdog*.

0 – Para o TMR0.

■ **Bit 2 (PS2), bit 1 (PS1) e bit 0 (PS0):** Ajuste da escala. Disponível para R/W.

A [Tabela 18.1](#) mostra os valores das escalas para as combinações dos bits PS2, PS1 e PS0.

■ **TRISA:** Permite a definição dos pinos da PORTA como entrada ou saída.

Inicialmente todos os bits são 1.

1 – entrada.

0 – saída

■ **TRISB:** Permite a definição dos pinos da PORTB como entrada ou saída.

Inicialmente todos os bits são 1.

1 – entrada.

0 – saída

■ **ECON1:** Registro que controla as operações da EEPROM.



**FIGURA 18.14** Registrador EECON1.

- **Bit 7, bit 6 e bit 5:** Não são utilizados; seus valores sempre são zero.
  - **Bit 4 - EEIE:** Sinalizador (*flag*) de fim de escrita na EEPROM. Disponível para R/W.
    - 1 – Terminou a escrita.
    - 0 – Não terminou a escrita.
  - **Bit 3 – WRERR:** Sinalizador (*flag*) de erro de escrita na EEPROM. Disponível para R/W.
    - 1 – Parou pelo RESET ou pelo *watchdog*.
    - 0 – Sem erro de escrita.
  - **Bit 2 – WREN:** Bit de habilitação de escrita na EEPROM. Disponível para R/W.
    - 1 – Escrita permitida.
    - 0 – Escrita não permitida.
  - **Bit 1 – WR:** Bit de início de escrita na EEPROM. Disponível para R/W O programa pode ler, mas apenas seta o bit.
    - 1 – Inicia um ciclo de escrita. A memória zera o bit quando termina uma escrita.
    - 0 – Fim de escrita.
  - **Bit 0 RD:** Bit de início de leitura na EEPROM. Disponível para R/W. O programa pode ler, mas apenas setar o bit.
    - 1 – Inicia um ciclo de leitura. A memória zera o bit quando termina uma leitura.
    - 0 – Não inicia uma leitura.
  - **EECON2:** Registro que controla as operações da EEPROM.  
Não é um registrador implementado fisicamente. Para se iniciar um processo de escrita, deve-se mover 55h e depois AAh para o EECON2, para evitar problemas no processo de escrita.

---

**Tabela 18.1**

Valores das escalas do temporizador 0 e do *watchdog* para as combinações dos bits PS2, PS1 e PS0

---

BIT 2 PS2	BIT 1 PS1	BIT 0 PS0	Escala Temporizador	Escala Watchdog
0	0	0	1:2	1:1
0	0	1	1:4	1:2
0	1	0	1:8	1:4

0	1	1	1:16	1:8
1	0	0	1:32	1:16
1	0	1	1:64	1:32
1	1	0	1:128	1:64
1	1	1	1:256	1:128

## **Formato das instruções do PIC 16F84**

Antes de estudarmos as instruções do PIC 16F84, vamos mostrar os quatro formatos de instruções que ele reconhece. As descrições destes formatos estão a seguir, e utilizam a nomenclatura mostrada na [Tabela 18.2](#).

---

**Tabela 18.2**

Nomenclatura usada na descrição dos formatos de instrução do PIC 16F84

---

Campo	Descrição
f	Endereço de registradores (0x00 a 0x7F)
W	Registrador Work (acumulador)
b	Endereço do bit dentro do registrador de 8 bits
k	Literal, constante ou <i>label</i>

d | Seleção de destino; d = 0 armazena resultado em W; d = 1 no registrador f; default d = 1.

### **Formato de instruções que operam com byte (byte oriented) e com o file register**

A Figura 18.15 mostra o formato, onde os 6 bits mais significativos correspondem ao *opcode*, **d** é o bit de destino, quando for 0 o destino é o registrador **W**, quando for 1 é um dos registradores do *file register* (SPR ou GPR). Os últimos 7 bits correspondem ao **f**, que é o endereço do registrador do *file register* (SPR ou GPR) a ser usado.



**FIGURA 18.15** Formato de operações, com byte, com o *file register*.

### **Formato de instruções que operam com bit (bit oriented) e com o file register**

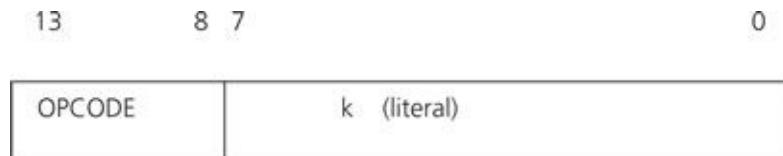
A Figura 18.16 nos mostra esse formato: os quatro bits mais significativos correspondem ao *opcode*, e os três seguintes, ao **b** que é a posição do bit. O bit de destino é **d**: quando for 0, o destino é o registrador **W**, quando for 1, é um dos registradores do *file register* (SPR ou GPR). Os últimos sete bits correspondem ao **f**, que é o endereço do registrador do *file register* (SPR ou GPR) a ser usado.



**FIGURA 18.16** Formato de operações, com bit, com o *file register*.

### **Formato de instruções que operam com literais e instruções de controle**

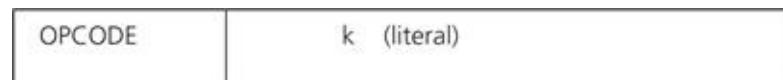
A Figura 18.17 nos mostra o formato dessas instruções: os seis bits mais significativos correspondem ao *opcode*, e os oito seguintes, ao k, que é o literal.



**FIGURA 18.17** Formato de operações com literais e instruções de controle.

### **Formato de instruções de controle call e goto**

A Figura 18.18 nos mostra o formato dessas instruções: os três bits mais significativos correspondem ao *opcode*, e os 11 seguintes, ao k, que é o literal.



**FIGURA 18.18** Formato de instruções de controle call e goto.

## **Conjunto De Instruções Do PIC 16F84**

O microcontrolador PIC 16F84 tem 35 instruções de 14 bits dos seguintes tipos:

- **Transferência de dados:** São instruções que fazem a transferência de conteúdos entre o registrador Work (W) e o registrador f (*register file*). O registrador f que representa qualquer um dos registradores de função específica (SPR) ou de propósito geral (GPR).
- **Aritmética e lógica:** São instruções que possibilitam a execução de operações como a soma, a subtração, a multiplicação e a divisão.
- **Operações com bit:** São instruções que permitem manipular um bit de uma palavra.
- **Controle do fluxo de execução de um programa:** São as instruções que possibilitam o desvio dentro de um programa.

A seguir classificamos as instruções por tipo, lembrando que a Tabela 18.2 tem a descrição das nomenclaturas utilizadas na explicação das instruções.

## ***Instruções de movimentação de dados***

A Tabela 18.3 mostra as instruções de movimentação de dados.

**Tabela 18.3**

### **Instruções de movimentação de dados**

Mnemônico	Descrição	Operação	Flag
MOVLW k	Copia para W, uma constante k.	W ← k	—
MOVWF f	Copia para f, o conteúdo de W.	f ← W	—
MOVF f,d	Copia o conteúdo de f para W se d = 0 e para f se d = 1.	f ← d	Z
clrW	Zera W.	W ← 0	Z
CLRF f	Zera f.	f ← 0	Z
SWAPF f,d	Troca os <i>nibbles</i> em f, e o resultado irá para W, se d = 0, e para f se d = 1.	f(3:0)(7:4) ← f(7:4)(3:0)	—

A seguir, alguns exemplos de uso dessas instruções:

```
MOVLW AAh ; após a execução da instrução, o  
            registrador W = AAh.
```

```
MOVWF 19h ; supondo que antes da execução  
            W = 66h e f(19h) = BCh, após a  
            execução teremos W = 66h e f(19h) = 66h.
```

```
movf 19h,0 ; supondo que antes da execução  
            f(19h) = 66h e flag Z = 0, após a  
            execução teremos W = 66h e flag Z = 1.
```

```
movf 19h,1 ; supondo que antes da execução f(19h)  
            = 66h e flag Z = 0, após a execução  
            teremos f(19h) = 66h e flag Z = 1.
```

```
clrw ; supondo que antes da execução  
            W = 66h, após a execução teremos  
            W = 0 e flag Z = 1.
```

```
CLRF 19h ; supondo que antes da execução f(19h)  
            = 66h, após a execução teremos f(19h)  
            = 0 e flag Z = 1.
```

```
SWAPF 19h,0 ; supondo que antes da execução  
            f(19h) = 34h, após a execução  
            teremos W = 43h.
```

```
SWAPF 19h,1 ; supondo que antes da execução f(19h)  
            = 34h, após a execução teremos  
            f(19h)=43h
```

## Instruções aritméticas e lógicas

A Tabela 18.4 mostra as instruções aritméticas e lógicas.

**Tabela 18.4**

### Instruções aritméticas e lógicas

Mnemônico	Descrição	Operação	Flag
ADDLW k	Soma a constante k a W e armazena o resultado em W.	$W \leftarrow k + W$	C,DC,Z
ADDWF f,d	Soma o conteúdo de f com o conteúdo de W; se d = 0 armazena em W, e se d = 1 armazena em f.	$d \leftarrow f + W$	C,DC,Z
SUBLW k	Subtrai da constante k o conteúdo de W e armazena o resultado em W.	$W \leftarrow k - W$	C,DC,Z
SUBWF f,d	Subtrai do conteúdo de f o conteúdo de W; se d = 0 armazena em W, e se d = 1 armazena em f.	$d \leftarrow f - W$	C,DC,Z
ANDLW k	Faz um AND entre W e a constante k e armazena o resultado em W.	$W \leftarrow W \& k$	Z
ANDWF f,d	Faz um AND entre o conteúdo de f com o conteúdo de W; se d = 0 armazena em W, e se d = 1 armazena em f.	$d \leftarrow W \& f$	Z
IORLW k	Faz um OR entre W e a constante k e armazena o resultado em W.	$W \leftarrow W   k$	Z
IORWF f,d	Faz um OR entre o conteúdo de f com o conteúdo de W; se d = 0, armazena em W, e se d = 1 armazena em f.	$d \leftarrow W   f$	Z
XORLW k	Faz um XOR entre W e a constante k, e armazena o resultado em W.	$W \leftarrow W \text{ xor } k$	Z
XORWF f,d	Faz um XOR entre o conteúdo de f com o conteúdo de W; se d = 0 armazena em W, e se d = 1 armazena em f.	$d \leftarrow W \text{ xor } f$	Z
INCF f,d	Incrementa o conteúdo de f; se d = 0 armazena em W, e se d = 1 armazena em f.	$d \leftarrow f + 1$	Z
DECWF f,d	Decrementa o conteúdo de f; se d = 0 armazena em W, e se d = 1 armazena em f.	$d \leftarrow f - 1$	Z

A seguir, alguns exemplos de uso dessas instruções:

ADDLW 10h ; se W = 25h antes da execução da instrução, depois da execução teremos W = 35h.

ADDWF 15h,0 ; se W = 25h e f(15h) = 10h, antes da execução da instrução, depois da execução teremos W = 35h e f(15h) = 10h.

ADDWF 15h,1 ; se W = 25h e f(15h) = 10h, antes da execução da instrução, depois da execução teremos f(15h) = 35h e W = 25h.

SUBLW 10h ; se W = 25h antes da execução da instrução, depois da execução teremos W = 15h.

SUBWF 15h,0 ; se W = 10h e f(15h) = 25h, antes da execução da instrução, depois da execução teremos W = 15h e f(15h) = 25h.

SUBWF 15h,1 ; se W = 10h e f(15h) = 25h, antes da execução da instrução, depois da execução teremos f(15h) = 15h e W = 25h.

ANDLW F0h ; se W = 25h antes da execução da instrução, depois da execução teremos W = 20h

ANDWF 13h,0 ; se W = 25h e f(13h) = F0h, antes da execução da instrução, depois



da execução teremos  $W = 20h$  e  
 $f(13h) = F0h$ .

ANDWF 13h,1 ; se  $W = 25h$  e  $f(13h) = F0h$ , antes  
da execução da instrução, depois  
da execução teremos  $f(15h) = 20h$  e  
 $W = 25h$ .

IORLW F0h ; se  $W = 25h$  antes da execução da  
instrução, depois da execução teremos  
 $W = F5h$ .

IORWF 13h,0 ; se  $W = 25h$  e  $f(13h) = F0h$ , antes  
da execução da instrução, depois  
da execução teremos  $W = F5h$  e  
 $f(13h) = F0h$ .

IORWF 13h,1 ; se  $W = 25h$  e  $f(13h) = F0h$ , antes  
da execução da instrução, depois  
da execução teremos  $f(15h) = F5h$  e  
 $W = 25h$ .

XORLW F0h ; se  $W = 0Fh$  antes da execução da  
instrução, depois da execução teremos  
 $W = 00h$ .

XORWF 13h,0 ; se  $W = 0Fh$  e  $f(13h) = F0h$ , antes  
da execução da instrução, depois  
da execução teremos  $W = 00h$  e  
 $f(13h) = F0h$ .

XORWF 13h,1 ; se  $W = 0Fh$  e  $f(13h) = F0h$ , antes  
da execução da instrução, depois da  
execução teremos  $f(15h) = 00h$  e  
 $W = 0fh$ .

```
INCF 13h,0 ; se W = 15h e f(13h) = 24h, antes  
da execução da instrução, depois  
da execução teremos W = 25h e  
f(13h) = 24h.
```

```
DECF 13h,1 ; se W = 15h e f(13h) = 24h, antes  
da execução da instrução, depois  
da execução teremos f(13h) = 25h e  
W = 15h.
```

## Instruções de rotação e complemento

A Tabela 18.5 nos mostra as instruções de rotação e complemento.

---

**Tabela 18.5**

**Instruções de rotação e complemento**

---

Mnemônico	Descrição	Flag
RLF f,d	Rotação à esquerda de f, através do carry, se d = 0 armazena em W, e se d = 1 armazena em f.	C
RRF f,d	Rotação à direita de f, através do carry, se d = 0 armazena em W, e se d = 1 armazena em f.	C
COMF f,d	Complemento de 1 de f, se d = 0 armazena em W, e se d = 1 armazena em f.	Z

A seguir, alguns exemplos de uso dessas instruções:

```
RLF 13h,0 ; se W = 25h, f(13h) = F0h e C = 1,  
antes da execução da instrução,  
depois da execução teremos W = E1h,  
f(13h) = F0h e C = 1.
```

```
RLF 13h,1 ; se f(13h) = F0h e C = 1, antes da  
execução da instrução, depois da  
execução teremos f(13h) = E1h e C = 1.
```

```
RRF 13h,0 ; se W = 25h, f(13h) = F0h e C = 1,  
antes da execução da instrução,  
depois da execução teremos W = F8h,
```

f(13h) = F0h e C = 0.

```
RRF 13h,1 ; Se f(13h) = F0h e C=1, antes da  
execução da instrução, depois da  
execução teremos f(13h) = F8h e  
C = 0.
```

```
COMF 13h,0 ; Se W = 25h, f(13h) = F0h, antes  
da execução da instrução, depois  
da execução teremos W = 0Fh,  
f(13h) = F0h.
```

```
COMF 13h,0 ; Se f(13h) = F0h, antes da execução  
da instrução, depois da execução  
teremos f(13h) = 0Fh.
```

## ***Instruções de operações com bit***

A Tabela 18.6 nos mostra as instruções de operações com bit.

---

**Tabela 18.6**  
**Instruções de operações com bit**

---

Mnemônico	Descrição	Operação	Flag
BCF f,b	Zera bit b de f.	$f(b) \leftarrow 0$	—
BSF f,b	Seta bit b de f.	$f(b) \leftarrow 1$	—

A seguir, alguns exemplos de uso dessas instruções:

BCF 13h,1 ; se $f(13h) = FFh$ , antes da execução da instrução, depois da execução teremos $f(13h) = FDh$ .
BSF 13h,1 ; se $f(13h) = F0h$ , antes da execução da instrução, depois da execução teremos $f(13h) = F2h$ .

### **Instruções de controle de fluxo de execução de um programa**

A Tabela 18.7 nos mostra as instruções de controle de fluxo de execução de um programa.

---

**Tabela 18.7**  
**Instruções de controle de fluxo de execução de um programa**

---

Mnemônico	Descrição	Flag
BTFSC f,b	Se o bit b de f for 0, pula uma instrução.	—
BTFSS f,b	Se o bit b de f for 1, pula uma instrução.	—
DECFSZ f,d	Decrementa f, e se o resultado for zero, pula uma instrução. Se d = 0, o novo valor de f é armazenado em W, se d = 1, é armazenado em f.	—
INCFSZ f,d	Incrementa f e se o resultado for zero, pula uma instrução. Se d = 0, o novo valor de f é armazenado em W, se d = 1, é armazenado em f.	—
GOTO k	Desvia para o endereço k.	—
CALL k	Chama procedimento no endereço k.	—
return	Retorno da procedimento.	—
RETLW k	Retorno com constante k armazenada em W.	—
retfe	Retorna da rotina interrupção.	—

A seguir, temos alguns exemplos de uso dessas instruções:

BTFSC 13h,1 ; se f(13h) = F0h, ele não executará a próxima instrução.

BTFSS 13h,1 ; se f(13h) = F2h, ele não executará a próxima instrução.

DECFSZ 13h,0; se W = 12h e f(13h) = 01h, ele não executará a próxima instrução e teremos W = 00h e f(13) = 01h.

DECFSZ 13h,1; se W = 12h e f(13h) = 01h, ele não executará a próxima instrução e teremos W = 12h e f(13) = 00h.

INCFSZ 13h,0; se W = 12h e f(13h) = FFh (-1 em complemento de 2), ele não executará a próxima instrução teremos W = 00h e f(13) = FFh.

DECFSZ 13h,1; se W = 12h e f(13h) = FFh (-1 em complemento de 2), ele não executará a próxima instrução e teremos W = 12h e f(13) = 00h.

GOTO 23h ; a próxima instrução a ser executada é aquela no endereço 23h (PC recebe 23h).

CALL 23h ; a próxima instrução a ser executada será a primeira instrução da procedimento que está no endereço 23h (PC recebe 23h). O endereço de retorno (endereço da instrução depois do call) será armazenado no topo da pilha. Instrução de chamada de procedimento.

RETURN ; a próxima instrução a ser executada será a instrução cujo endereço está armazenado no topo da pilha (PC recebe o conteúdo do topo da pilha). Instrução de retorno de procedimento.

RETLW 10h ; a próxima instrução a ser executada será a instrução cujo endereço está armazenado no topo da pilha (PC recebe o conteúdo do topo da pilha) e W receberá o valor 10h. Instrução de retorno de procedimento com alteração de W.

**RETFIE** ; a próxima instrução a ser executada será a instrução cujo endereço está armazenado no topo da pilha (PC recebe o conteúdo do topo da pilha) e o bit GIE (bit 7 do registrador INTCON) é setado. Instrução de retorno de rotina de tratamento de interrupção.

## Outras instruções

A Tabela 18.8 nos mostra algumas instruções especiais que controlam o hardware.

---

**Tabela 18.8**

**Outras instruções**

---

Mnemônico	Descrição	Operação
nop	Nenhuma operação.	
clrwdt	Zera o temporizador <i>watchdog</i> .	WDT ← 0, TO\ ← 1, PD\ ← 1
sleep	Inicia para o modo <i>standby</i> .	WDT ← 0, TO\ ← 1, PD\ ← 0

A seguir alguns exemplos de uso destas instruções:

NOP	; nenhuma operação é feita, consome 1 ciclo de máquina.
clrwdt	; reinicializa o temporizador <i>watchdog</i> , setando os bits TO\ e PD\ (bits 4 e 3 do registrador de STATUS). Se tiver escala, o registrador OPTION também será zerado.
sleep	; a UCP entra no modo SLEEP, o gerador de clock para e o <i>watchdog</i> e a escala são zerados.



## Que Vem Depois

Apresentamos neste capítulo uma introdução aos microcontroladores e à família PIC®, da Microchip® Technology Inc. Os microcontroladores desta família são

simples, ideais para o aprendizado da programação em linguagem de montagem com este tipo de dispositivo. No próximo capítulo, veremos como programar o PIC® 16F84.

---

## CAPÍTULO 19

---

# Programação do microcontrolador PIC® 16F84

---

### Objetivos do capítulo

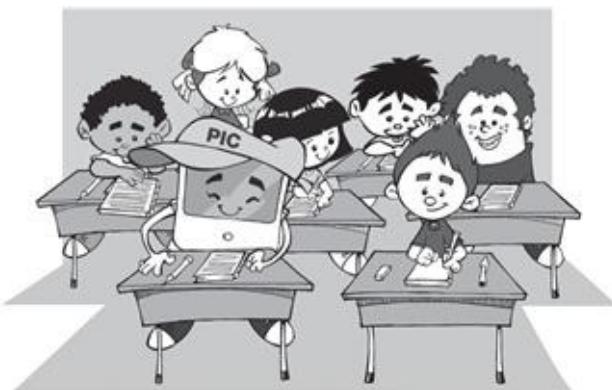
Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- Aprender como transformar o código em linguagem de montagem em código de linguagem de máquina.
- Programar o microcontrolador PIC 16F84.



### Apresentação

Assim como no estudo de caso no qual o processador INTEL 8086 foi analisado, vamos também aprender como utilizar um programa de maneira a gerar o código de máquina que será gravado no microcontrolador PIC 16F84.



Apresentaremos as diretivas para a linguagem de montagem do PIC 16F84, assim como todo o processo para gravação do programa em linguagem de máquina no PIC 16F84.

# Fundamentos

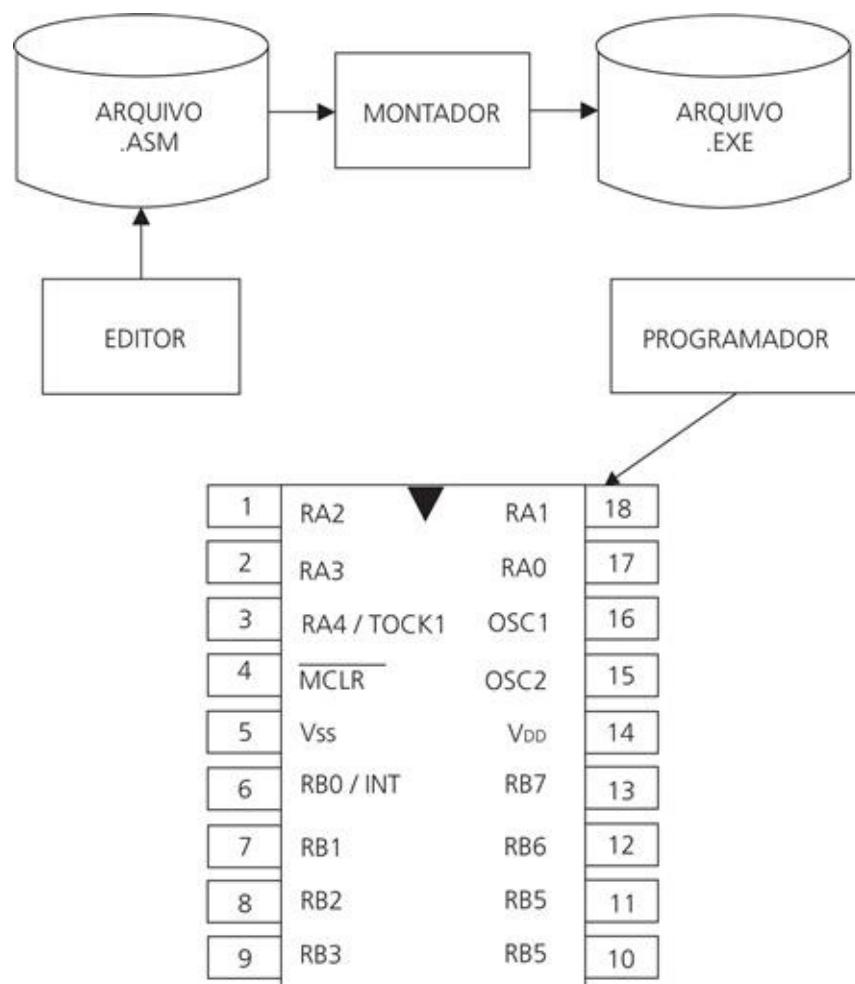
## ***Edição, montagem e gravação de um programa no PIC 16F84***

Para programar um microcontrolador é necessário executar os seguintes passos:

- Editar o programa em linguagem de montagem, gerando um arquivo com extensão .asm.
- Passar por um montador e gerar arquivo .exe.
- Passar por um programador de microcontroladores e gravar o programa no PIC 16F84.

Vocês notaram que os dois primeiros passos, como era de se esperar, são válidos também para um processador de uso geral. A única diferença é que aqui iremos gravar o código executável na memória de programa.

A [Figura 19.1](#) mostra esses passos.



**FIGURA 19.1** Programação PIC 16F84.

## **Elementos da linguagem da montagem do PIC 16F84**

Os elementos da linguagem de montagem têm o objetivo de permitir que o usuário escreva um programa em *assembly* para que o montador possa traduzi-lo em código executável.

Para o PIC 16F84, temos os seguintes elementos:

■ **Labels:** também chamados de rótulos, são símbolos que indicam um endereço de um dado ou uma instrução. Devem começar na primeira coluna e não podem começar por números. São *labels* corretos:

- start
- \_end
- P123
- Isto\_eh\_grande?

São *labels* errados:

- Start (não está na primeira coluna)
- 2\_fim

■ **Instruções:** o conjunto de instruções foi visto no capítulo anterior.

■ **Operandos:** os tipos de operandos podem ser um número (hexadecimal, decimal ou binário) ou uma variável representada por um símbolo. Por exemplo:

- Level ; é uma variável
- H'FF' ; corresponde ao número hexadecimal FF.

■ **Diretivas:** são comandos para o montador, os quais podemos classificar de acordo com os tipos seguintes.

## **Diretivas de controle**

Na [Tabela 19.1](#), podemos observar as diretivas de controle utilizadas. Elas geralmente são introduzidas para incluir arquivos, definir textos, constantes e variáveis, além de especificar o fim de um programa.

---

### **Tabela 19.1**

#### **Diretivas de controle**

---

Diretivas	Descrição

#define	Dá um significado para um texto
#include	Inclui arquivo no programa
Constant	Define uma constante
Variable	Define uma variável
Set	Define uma variável
Que	Define uma constante
Org	Define um endereço p/ carregar o programa
End	Fim do programa

## ***Diretivas condicionais***

Na [Tabela 19.2](#), temos as diretivas condicionais utilizadas. Elas permitem a escolha das instruções a serem executadas, dependendo de uma condição.

---

**Tabela 19.2**

### **Diretivas condicionais**

---

Diretivas	Descrição
if	Salto condicional
else	Alternativa para o if
endif	Fim de um salto condicional
while	Executa trecho enquanto condição verdadeira
endw	Fim de um while
ifdef	Execução de parte do programa, se símbolo foi definido
ifndef	Execução de parte do programa, se símbolo não foi definido

## ***Diretivas de dados***

Na [Tabela 19.3](#) temos as diretivas de dados utilizadas. Elas permitem a definição de tipo e bloco de dados.

---

**Tabela 19.3**

### **Diretivas de dados**

---

Diretivas	Descrição
Cblock	Define um bloco de constantes
Endc	Fim da definição de um bloco de constantes

DB	Define um dado do tipo byte
De	Define um byte da EEPROM
Dt	Define um conjunto (tabela) de dados

## **Diretivas de configuração**

Na [Tabela 19.4](#) temos as diretivas de configuração utilizadas. Elas permitem a definição do microcontrolador para o qual o código será gerado e a configuração inicial de bits.

---

**Tabela 19.4**

### **Diretivas de configuração**

---

Diretivas	Descrição
_config	Seta configuração de bits
Processor	Define o tipo do microcontrolador

Exemplos mais comuns de utilização de diretivas:

```
PROCESSOR 16F84
; define o microcontrolador para o qual o código será
gerado.
#include #p16f84, inc
; inclui um outro arquivo no arquivo corrente.
```

```
_CONFIG _CP_OFF&WDT_OFF&_PWRTE_ON&XT_OSC
; define valores iniciais para os bits CP, PWRTE, T_OSC.
```

■ **Comentários:** delimitado por um “;”.

A seguir, apresentamos um trecho de programa que calcula o número de voltas de um motor, para exemplificar a utilização de comentários.

```

;
;parte do programa que calcula o número de voltas do motor
;



|                    |                                                                              |
|--------------------|------------------------------------------------------------------------------|
| clrf TMRO          | ;TRMO = 0                                                                    |
| clrf INTCON        | ;interrupções e T0IF=0, desabilitados                                        |
| bsf STATUS, RP0    | ;banco 1 pelo OPTION_REG                                                     |
| movlw B'00110001'  | ;escala 1:4, borda de descida para clock externo, pull up na porta B ativada |
| movwf OPTION_REG   | OPTION_REG <- W                                                              |
| TO_OVFL:           |                                                                              |
| btfss INTCON, T0IF | testa bit overflow                                                           |
| goto TO_OVFL       | interrupção não ocorreu, espere                                              |


```

■ **Macros:** são trechos de código que realizam uma determinada função e recebem um nome. Toda vez que este nome é encontrado no código, o montador substitui pelo trecho de programa.

A macro a seguir, por exemplo, permite a saída de um valor passado por argumento (ARG1) pela PORTB. As diretivas: **macro** e **endm** são os delimitadores da macro, e **NA\_PORTB** é o nome da macro.

```

NA_PORTB macro ARG1
    ; macro que permite a saída de um valor, passado por ARG1,
    na

    ; PORTB
    BANK0
    movlw ARG1
    movwf PORTB

```

;	PORTB
BANK0	seleciona o banco 0 da memória
movlw ARG1	argumento ARG1 é armazenado em W
movwf PORTB	w (valor ARG1) é colocado na PORTB endm

■ **Operadores:** são caracteres que indicam uma determinada operação. Na [Tabela 19.5](#) temos a relação de operadores, seu significado e um exemplo de utilização.

**Tabela 19.5**  
**Operadores**

Operador	Descrição	Exemplo
\$	Status corrente do Program Counter	goto \$ +6
)	Parêntese direito	5 + ( x - 3 )
(	Parêntese esquerdo	5 + ( x - 3 )
!	Complemento lógico	if !( a - b )
-	Complemento	x = -x
-	Negação	-3 * x
High	Retorna byte mais significativo	movlw high x
Low	Retorna byte menos significativo	movlw low x
*	Multiplicação	a = b * c
/	Divisão	a = b / c
%	Divisão pelo módulo	a = b % 16
+	Adição	a = b + c
-	Subtração	a = b - c
<<	Mover para a esquerda	x = flags << 1
>>	Mover para a direita	x = flags >> 1
>=	Maior que ou igual	if entr_x >= num_entr
>	Maior	if entr_x > num_entr
<	Menor	if entr_x < num_entr
<=	Menor que ou igual	if entr_x <= num_entr
==	Igual	if entr_x == num_entr
!=	Diferente	if entr_x != num_entr
&	Operação AND com bits	flags = flags & BIT_ERRO

<code>^</code>	Operação XOR com bits	<code>flags = flags ^ BIT_ERRO</code>
<code> </code>	Operação OR com bits	<code>flags = flags   BIT_ERRO</code>
<code>&amp;&amp;</code>	Operação AND	<code>if (len == 256) &amp;&amp; (x==y)</code>
<code>  </code>	Operação OR	<code>if (len == 256)    (x==y)</code>
<code>=</code>	Igual	<code>indice = 0</code>
<code>+=</code>	Somar e atribuir	<code>indice += 1</code>
<code>-=</code>	Subtrair e atribuir	<code>indice -= 1</code>
<code>*=</code>	Multiplicar e atribuir	<code>indice *= 1</code>
<code>/=</code>	Dividir e atribuir	<code>indice /= 1</code>
<code>%=</code>	Dividir pelo módulo e atribuir	<code>indice %= 1</code>
<code>&lt;=</code>	Mover para a esquerda e atribuir	<code>indice &lt;= 3</code>
<code>&gt;=</code>	Mover para a direita e atribuir	<code>indice &gt;= 3</code>
<code>&amp;=</code>	Operação AND e atribuir	<code>indice &amp;= masc_indice</code>
<code> =</code>	Operação OR e atribuir	<code>indice  = masc_indice</code>
<code>^=</code>	Operação XOR e atribuir	<code>indice ^= masc_indice</code>
<code>++</code>	Incremento	<code>x++</code>
<code>--</code>	Decremento	<code>x--</code>



## Que Vem depositis

Agora estamos prontos para iniciar a programação do PIC 16F84. Nos exercícios desta parte você encontrará não só o código do programa, mas também a ligação do microcontrolador com os dispositivos, de acordo com a necessidade do exercício.

---

## Referências Bibliográficas

1. Bonatti I, Madureira M. *Introdução à Análise e Síntese de Circuitos Lógicos*. 2. ed. Campinas: Editora da Unicamp; 1995.
2. Brey BBarry. *The Intel Microprocessors 8086/8088, 80186, 80286, 80386, and 80486 Architecture, Programming, and Interfacing*. 3. ed. Nova Jersey: Prentice Hall; 2004.
3. Brown S, Vranesic Z. *Fundamentals of Digital Logic with VHDL Design*. Nova York: McGraw-Hill; 2004.
4. Capuano FG, Idoeta IV. *Elementos de Eletrônica Digital*. 40. ed. São Paulo: Érica; 2008.
5. Floyd TL. *Sistemas Digitais – Fundamentos e Aplicações*. 9. ed. São Paulo: Artmed Bookman; 2007.
6. Hamacher VCarl, Vranesic ZG, Zaky G. *Computer Organization*. Nova York: McGraw-Hill; 1990.
7. Intel Corporation. *8086 family Users Manual*, 1979.
8. ——. *Intel Microprocessors Documentation. 80386 High Performance 32-bit Microprocessor with Integrated Memory Management*, 1986.
9. Lipovski GJack. *Introduction to Microcontrollers*. Massachussets: Academic Press; 1999.
10. Liu Yu-Cheng, Gibson Glenn A. *Microcomputer Systems: The 8086/8088–Architecture, Programming, and Design*. 2.ed. Prentice-Hall International Editions 1986.
11. MICROCHIP Technology Inc. *Matlab – Ambiente de Desenvolvimento Integrado*. Acessado em: [www.microchip.com](http://www.microchip.com).
12. ——. *PIC16F84A Data Sheet 18-pin Enhanced FLASH/EEPROM 8-bit Microcontroller*, 2001.
13. Patterson David A, Hennessy John L. *Organização e Projeto de Computadores – A Interface Hardware/Sofware*. 2. ed. São Paulo: LTC; 2000.
14. Pedroni VA. *Eletrônica Digital Moderna e VHDL*. São Paulo: Campus Elsevier; 2010.
15. Sedra SA, Smith KC. *Microeletrônica*. 3. ed. São Paulo: Makron Books; 2000;

16. Stallings W. *Arquitetura e Organização de Computadores*. 8. ed. São Paulo: Pearson Prentice-Hall; 2010.
17. Tanenbaum A. *Organização Estruturada de Computadores*. 4. ed. São Paulo: LTC; 2001.
18. Taub H. *Circuitos Digitais e Microprocessadores*. São Paulo: Makron Books do Brasil; 1984.
19. Tocci RJ. *Sistemas Digitais - Princípios e Aplicações*. 5. ed. São Paulo: Prentice-Hall; 1994.
20. Vahid F. *Sistemas Digitais, Projeto, Otimização e HDLs*. São Paulo: Bookman; 2008.
21. Yu Y, Marut C. *Assembly Language Programming and Organization of IBM PC*. Nova York: McGraw-Hill; 1992.