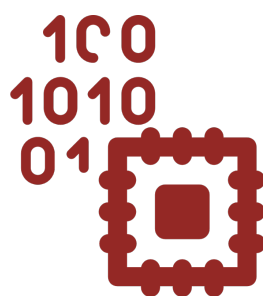




Universidade Federal do Amazonas
Instituto de Computação



APOSTILA DE ICCo6o SISTEMAS LÓGICOS

LEANDRO SILVA GALVÃO DE CARVALHO

Agosto 2018 – versão 0.8

© Leandro Silva Galvão de Carvalho

Versão: Agosto 2018

Apostila da disciplina ICCo60 Sistemas Lógicos, ministrada pelo Instituto de Computação da Universidade Federal do Amazonas (IComp / UFAM).

Endereço: Av. General Rodrigo Octávio, 6200, Coroado I – Manaus-AM – CEP 69080-900



Você tem a liberdade de:

Compartilhar: copiar, distribuir e transmitir esta obra.

Remixar: criar obras derivadas.

Sob as seguintes condições:

Atribuição: você deve creditar a obra da forma especificada pelo autor ou licenciante (mas não de maneira que sugira que estes concedem qualquer aval a você ou ao seu uso da obra).

Uso não comercial: você não pode usar esta obra para fins comerciais.

Compartilhamento pela mesma licença: se você alterar, transformar ou criar em cima desta obra, poderá distribuir a obra resultante apenas sob a mesma licença, ou sob uma licença similar à presente.

SUMÁRIO

1	INTRODUÇÃO AOS SISTEMAS LÓGICOS	1
1.1	A Necessidade de Computar	3
1.2	O Modelo de von Neumann	3
1.3	Organização da Apostila	3
i	SISTEMAS DE REPRESENTAÇÃO	5
2	SISTEMAS DE NUMERAÇÃO	7
2.1	O que é um Sistema de Numeração?	7
2.1.1	Esgotamento de Símbolos	9
2.1.2	Base Numérica	10
2.1.3	O Sistema Decimal de Numeração	11
2.1.4	O Sistema Binário	15
2.1.5	O Sistema Dozenal	17
2.1.6	O Sistema Hexadecimal	19
2.2	Conversão de uma Base B para a Base Decimal	20
2.2.1	Conversão Binário-Decimal	22
2.2.2	Conversão Dozenal-Decimal	23
2.2.3	Conversão Hexadecimal-Decimal	24
2.3	Conversão da Base Decimal para uma Base B	25
2.4	Generalizando: Conversão de uma Base Z para Outra Base B	27
2.5	Instanciando: Quando Z é uma Potência de B	28
2.5.1	Conversão da Base Binária para Hexadecimal	29
2.5.2	Conversão da Base Hexadecimal para a Binária	30
2.6	Operações Aritméticas com Números Binários	31
2.6.1	Adição de Números Binários	31
2.6.2	Subtração de Números Binários	32
2.7	Representação de Números em Espaços Restritos	33
2.8	Resumo do Capítulo	38
	Exercícios do Capítulo 2	39
3	REPRESENTAÇÃO DE NÚMEROS INTEIROS	43
3.1	Notação Sinal e Magnitude	43
3.1.1	Conversão da Base 10 para a Notação Sinal e Magnitude.	44
3.1.2	Conversão da Notação Sinal e Magnitude para a Base 10.	45
3.1.3	Limites de Representação da Notação Sinal e Magnitude	46
3.1.4	Desvantagens da Notação Sinal e Magnitude	47
3.2	Complemento de 2	47
3.2.1	Complemento de Horas	48

3.2.2	Complemento de Dez	48	
3.2.3	A Notação Complemento de 2 (Finalmente)	51	
3.3	Operações Aritméticas em Complemento de 2	58	
3.3.1	Adição em Complemento de 2	58	
3.3.2	Subtração em Complemento de 2	59	
3.3.3	Overflow em Complemento de 2	60	
3.4	Notação com Excesso	62	
3.4.1	Propriedades da Notação com Excesso	62	
3.4.2	Conversão da Base Decimal para a Notação com Excesso	64	
3.4.3	Conversão da Notação com Excesso para a Base Decimal	66	
3.5	Resumo do Capítulo	67	
	Exercícios do Capítulo 3	68	
4	REPRESENTAÇÃO DE NÚMEROS REAIS	71	
4.1	Representação de Números Reais em Base Binária	71	
4.1.1	Conversão de Números Reais da Base Binária para Decimal	72	
4.1.2	Conversão de Números Reais da Base Decimal para a Binária	74	
4.1.3	Erros de Arredondamento	77	
4.1.4	Limitações da Notação Fixa	78	
4.2	Representação em Ponto Flutuante	78	
4.2.1	O Padrão IEEE 754	79	
4.3	Casos Especiais	86	
4.3.1	Como Representar o Zero na Notação IEEE 754?	86	
4.3.2	Infinito	86	
4.3.3	Situações Inválidas (NaN)	87	
4.3.4	Números Desnormalizados	87	
4.4	Propriedades	89	
4.5	Overflow e Underflow	91	
4.6	Resumo do Capítulo	92	
	Exercícios do Capítulo 4	93	
5	NÚMEROS BINÁRIOS NA LINGUAGEM C	97	
5.1	Tipos numéricos em C	97	
5.1.1	Tipos Inteiros	98	
5.1.2	Tipos Ponto Flutuante	99	
5.2	Entrada e Saída de Dados Numéricos em C	99	
5.3	A função printf()	101	
5.3.1	Especificadores de formato	102	
5.3.2	Modificadores de extensão	107	
5.3.3	Flags	108	
5.3.4	Tamanho mínimo	108	
5.3.5	Precisão	109	
5.4	Operações Lógicas Bit a Bit	111	
5.5	Deslocamento de Bits	112	

5.5.1	Deslocamento à Esquerda	112
5.5.2	Deslocamento à Direita	115
5.5.3	Resto da Divisão Inteira	117
5.6	Impressão de Valores em Formato Binário	117
5.6.1	Impressão de Inteiros em Binário	118
5.6.2	Impressão da Representação Hexadecimal de Ponto Flutuante	120
5.7	Overflow e Underflow	123
5.7.1	Overflow de Inteiros em Linguagem C	123
5.7.2	Overflow e Underflow de Ponto Flutuante em Linguagem C	123
5.8	Resumo do Capítulo	125
	Exercícios do Capítulo 5	126
6	NÚMEROS BINÁRIOS NA LINGUAGEM PYTHON	127
6.1	Números Inteiros em Python	127
6.2	Números Ponto Flutuante em Python	127
6.3	Resumo do Capítulo	127
	Exercícios do Capítulo 6	128
7	OUTRAS FORMAS DE REPRESENTAÇÃO	129
7.1	Binary Coded Decimal (BCD)	129
7.2	Código Gray	129
7.3	Representação de caracteres	129
7.3.1	ASCII	129
7.3.2	Unicode	129
7.4	Códigos de Detecção e Correção de Erros	129
7.4.1	Código de Barras	129
7.4.2	QR Code	129
7.5	Resumo do Capítulo	129
	Exercícios do Capítulo 7	130
ii	CIRCUITOS COMBINATÓRIOS	133
8	ÁLGEBRA BOOLEANA	135
8.1	Relação entre Álgebra Booleana e Conjuntos	135
8.2	Propriedades	135
8.3	Resumo do Capítulo	135
	Exercícios do Capítulo 8	136
9	PORTAS LÓGICAS	137
9.1	Porta NOT	137
9.2	Porta AND	137
9.3	Porta OR	137
9.4	Porta XOR	137
9.5	Porta NAND	137
9.6	Resumo do Capítulo	137
	Exercícios do Capítulo 9	138
10	MAPAS DE KARNAUGH	139
10.1	Resumo do Capítulo	139

Exercícios do Capítulo 10	140
11 CIRCUITOS COMBINATÓRIOS	141
11.1 Resumo do Capítulo	141
Exercícios do Capítulo 11	142
iii CIRCUITOS SEQUENCIAIS	143
12 CIRCUITOS SEQUENCIAIS	145
12.1 Resumo do Capítulo	145
Exercícios do Capítulo 12	146
13 CIRCUITOS DE MEMÓRIA	147
13.1 Resumo do Capítulo	147
Exercícios do Capítulo 13	148
14 JUNTANDO TUDO: O PROCESSADOR	149
14.1 Resumo do Capítulo	149
Exercícios do Capítulo 14	150
REFERÊNCIAS BIBLIOGRÁFICAS	151

LISTA DE FIGURAS

Figura 1	Níveis de abstração de um sistema computacional. Fonte: [10].	2
Figura 2	Nas sociedades menos complexas, não havia necessidade de se expressar grandes quantidades numéricas. Em termos práticos, pouco importava a diferença entre três ou dez elementos; o importante era correr. Fonte: [2].	4
Figura 3	A invenção do número zero facilitou a realização de operações aritméticas, de modo que o sistema de numeração romano foi substituído pelo sistema indo-arábico. Fonte: [2].	15
Figura 4	Algoritmo de conversão de um número expresso na base decimal para uma base B qualquer.	26
Figura 5	Algoritmo geral para conversão de bases. Fonte: [1].	28
Figura 6	Analogia de um espaço restrito de numeração a um tabuleiro circular.	35
Figura 7	Analogia da soma $(110 + 001)_2$ em um tabuleiro circular.	37
Figura 8	Analogia da soma $(110 + 011)_2$ em um tabuleiro circular.	37
Figura 9	Analogia da soma $(0110 + 0011)_2$ em um tabuleiro circular maior, agora com 16 casas.	37
Figura 10	Algoritmo de conversão de um número inteiro expresso na base decimal para a notação sinal e magnitude.	45
Figura 11	Faixa de valores inteiros possíveis de serem representados por uma palavra de n bits na notação sinal e magnitude.	47
Figura 12	Horas complementares no relógio são simétricas em relação ao eixo vertical.	48
Figura 13	Algoritmo de conversão de um número inteiro expresso na base decimal para a notação complemento de 2.	52
Figura 14	Faixa de valores inteiros possíveis de serem representados por uma palavra de n bits na notação complemento de 2.	55
Figura 15	Faixa de valores inteiros possíveis de serem representados por uma palavra de n bits na notação com excesso.	63

Figura 16	Algoritmo de conversão de um número inteiro expresso na base decimal para a notação com excesso. 64	
Figura 17	Algoritmo de conversão de um número real fracionário expresso na base decimal para uma base B qualquer. 75	
Figura 18	Codificação dos campos de um número ponto flutuante segundo o padrão IEEE 754. A extensão dos campos “expoente” e “fração” diferencia os formatos de precisão entre si. 81	
Figura 19	Formato de 32 bits considerando somente números normalizados. Fonte: [26]. 87	
Figura 20	Formato de 32 bits com números desnormalizados. Fonte: [26]. 89	
Figura 21	Densidade de distribuição de números ponto flutuante ao longo da reta numérica. Fonte: [26]. 91	
Figura 22	Subintervalos e regiões especiais na representação ponto flutuante. Fonte: [21]. 91	
Figura 23	As funções scanf() e printf() acessam a memória do computador identificada pela variável contida em seus argumentos. 100	
Figura 24	A função printf() lê dados da memória sem modificá-los, apresentando-os na tela sob algum formato especificado. 101	
Figura 25	Sintaxe de uma diretiva de formatação. 102	

LISTA DE TABELAS

Tabela 1	Representações dos valores de 0 a 32 nas bases binária, decimal, dozenal e hexadecimal. 21
Tabela 2	Faixa de números inteiros representáveis na notação complemento de 2 em uma palavra de n bits. 54
Tabela 3	Comparação das notações complemento de 2 e com excesso para uma palavra de n bits. 63
Tabela 4	Números binários podem ser convertidos em frações decimais. Fonte: [4]. 73
Tabela 5	Quantidade de bits ocupadas pelos formatos de número ponto flutuante definidos pelo padrão IEEE 754. 81
Tabela 6	Resumo dos principais parâmetros associados aos formatos IEEE 754. 93
Tabela 7	Tamanho e faixa de valores dos tipos inteiros. 98
Tabela 8	Tamanho e faixa de valores dos tipos ponto flutuante. 99
Tabela 9	Especificadores de formato para números inteiros. Fontes: [23, 16]. 103
Tabela 10	Exemplos de diretivas de formatação para números inteiros e seus respectivos resultados. 104
Tabela 11	Especificadores de formato para ponto flutuante. Fontes: [23, 16]. 105
Tabela 12	Exemplos de diretivas de formatação para números ponto flutuante e seus respectivos resultados. 106
Tabela 13	Modificadores de extensão. Fontes: [23, 16]. 108
Tabela 14	Exemplos de diretivas de modificação de extensão de extensão. 108
Tabela 15	Flags da função printf. Fontes: [23, 16]. 109
Tabela 16	Exemplos de diretivas de formatação para valores short int. 110
Tabela 17	Exemplos de diretivas de formatação para valores long double. 111
Tabela 18	Operadores bit a bit. Fontes: [23]. 111
Tabela 19	Correspondência entre algumas convenções BCD e seu valor em base decimal. 130
Tabela 20	Código Gray de 4 bits. 131
Tabela 21	Exemplos de códigos de correção de erro na transmissão de alguns valores numéricos. 131

Tabela 22	Código de barras UPC (<i>Universal Product Code</i>).	132
-----------	---	-----

LISTINGS

Código 1	Exemplo de declaração de variáveis em linguagem C. 97
Código 2	Modelo de código para encontrar o tamanho (em bytes) do espaço em memória ocupado pelos tipos da Linguagem C. 99
Código 3	Exemplo básico de entrada e saída de dados em linguagem C. 100
Código 4	Modelo de código C sobre os principais especificadores de formato de tipos inteiros, abordados no Exemplo 5.1. 103
Código 5	Modelo de código C sobre os principais especificadores de formato de tipos ponto flutuante, abordados no Exemplo 5.2. 105
Código 6	Modelo de código C sobre modificadores de extensão de tipos inteiros, abordados no Exemplo 5.3. 107
Código 7	Código C para exemplificar o uso de flags aliadas a tipos inteiros, conforme abordado no Exemplo 5.4. 109
Código 8	Código C para exemplificar o uso de flags aliadas a tipos de precisão dupla, conforme abordado no Exemplo 5.5. 110
Código 9	Modelo de código para encontrar o tamanho (em bytes) do espaço em memória ocupado pelos tipos da Linguagem C. 112
Código 10	Deslocamento à esquerda de uma variável unsigned char. 113
Código 11	Deslocamento à esquerda de uma variável char (sinalizada). 114
Código 12	Deslocamento lógico à direita de uma variável unsigned char. 115
Código 13	Deslocamento aritmético à direita de uma variável char. 116
Código 14	Função que imprime a sequência de bits de uma variável char. 119
Código 15	Impressão da sequência de bits de uma variável short usando mascaramento. 119
Código 16	Impressão da representação hexadecimal de uma variável float usando union. 120
Código 17	Impressão da representação hexadecimal de uma variável float usando ponteiro. 121

- Código 18 Dissecar a representação de bits de uma variável float usando campos de bits e union. [121](#)
- Código 19 Comparação entre a impressão de uma variável double e de outra float que armazenam valores iguais. [123](#)

ACRÔNIMOS

LSB Least Significant Bit

LSD Least Significant Digit

MSB Most Significant Bit

MSD Most Significant Digit

INTRODUÇÃO AOS SISTEMAS LÓGICOS

SISTEMAS COMPUTACIONAIS SÃO CONSTITUÍDOS por dois componentes básicos: *hardware* e *software*. Na verdade, ambos os termos são empregados de forma coletiva, de modo a designar um conjunto de componentes de mesma natureza. O *hardware* engloba os aspectos físicos dos sistemas computacionais: processador, memória, cabos, interfaces, fonte de alimentação, entre outros. Já o *software* designa os vários tipos de programas que operam sobre os sistemas computacionais.

Ambos os componentes são essenciais para o funcionamento de um dispositivo computacional. No entanto, conforme descrito por John von Neumann [6], há um importante sentido em que o *hardware* se torna prioritário em relação ao *software*. Trata-se de uma analogia com a Biologia. Um dispositivo composto somente de *hardware* pode existir e manter seu próprio metabolismo. Pode viver indefinidamente enquanto houver energia para engolir e números para mastigar. Já um dispositivo composto somente de *software* deverá ser obrigatoriamente um parasita. Só poderá funcionar em um mundo onde haja outro dispositivo cujo *hardware* possa tomar emprestado.

Estendendo o paralelo entre Biologia e Computação, podemos dizer que, da mesma forma que a separação entre o núcleo e citoplasma ao nível celular proporcionou a evolução de seres mais complexos, os multicelulares, a separação entre *hardware* e *software* proporcionou a criação de máquinas mais complexas, os computadores. As primeiras máquinas da humanidade serviam apenas para um único propósito: ver as horas, fazer uma soma, medir um ângulo. Nas máquinas modernas, não é necessário trocar de *hardware* toda vez que trocarmos de necessidade. Basta apenas trocar o *software*.

Nosso corpo pode ser dividido em diversas camadas de abstração: mente, cérebro, tecido cerebral, células nervosas, moléculas, átomos. Da mesma forma, sistemas computacionais são constituídos por diversas camadas. A abstração é uma técnica fundamental para gerenciar a complexidade, pois esconde detalhes quando não são importantes.

A Figura 1 ilustra os níveis de abstração de um sistema computacional, juntamente com blocos de construção típicos em cada nível. No nível mais baixo de abstração está a Física, o movimento de elétrons. O comportamento dos elétrons é descrito por mecânica quântica. Nosso sistema é construído a partir de dispositivos eletrônicos, como transistores (ou válvulas, nos primórdios). Esses dispositivos possuem pontos de conexão chamados terminais, que podem ser mo-

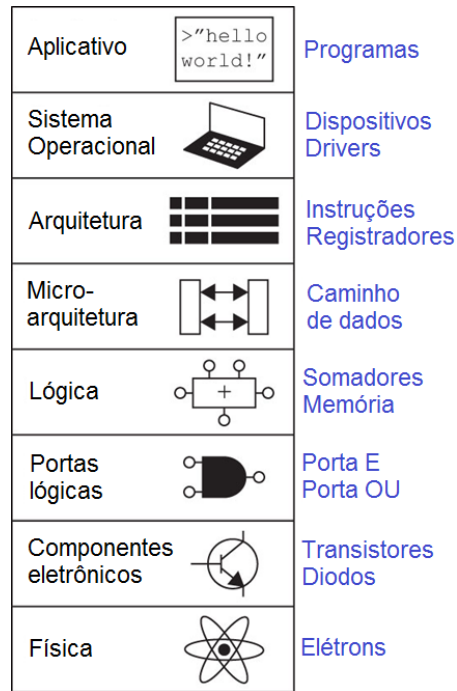


Figura 1: Níveis de abstração de um sistema computacional. Fonte: [10].

deloados pela relação entre a tensão e a corrente medida em cada terminal. Ao abstrair a esse nível dos transistores, podemos ignorar os elétrons individuais.

O próximo nível de abstração são os circuitos digitais, tais como portas lógicas, nos quais as tensões são restritas a intervalos discretos, indicados por 0 e 1. No projeto lógico, podemos construir estruturas mais complexas, como somadores ou memórias, a partir de circuitos digitais.

O nível *microarquitetura* interliga o nível de *lógica* ao de *arquitetura*. O nível de arquitetura descreve um computador do ponto de vista do programador. Por exemplo, a arquitetura Intel x86 usado por computadores pessoais (PCs) é definida por um conjunto de instruções e registradores (memória para armazenamento temporário de variáveis) que o programador está autorizado a usar. A microarquitetura envolve a combinação de elementos lógicos para executar as instruções definidas pela arquitetura.

Já no mundo do *software*, o sistema operacional lida com detalhes de baixo nível, tais como o acesso a um disco rígido ou gerenciamento de memória. Finalmente, o aplicativo usa essas facilidades oferecidas pelo sistema operacional para resolver um problema para o usuário. Graças ao poder de abstração, sua avó pode navegar na web sem se preocupar com as vibrações quânticas dos elétrons ou com a organização da memória no *tablet* dela.

Esta apostila enfoca os níveis dos circuitos digitais e da lógica. Após estudá-la, você compreenderá como os circuitos lógicos básicos podem ser combinados um processador simples. Depois dessa leitura, você está condenado a deixar de ser um usuário ingênuo. Entenderá que as façanhas e os cálculos realizados por um computador não são frutos de uma mágica inalcançável. Pelo contrário, são resultado do processamento de expressões lógicas encadeadas de maneira engenhosa.

1.1 A NECESSIDADE DE COMPUTAR

A origem da computação – o ato de calcular – é indicada pela origem da própria palavra *calcular*, que deriva da palavra latina *calculus*, cuja origem é a palavra grega *chalix*. Ambas significam pedra, ou seixo. Lembre-se que quando dizemos que uma pessoa tem *cálculo renal*, significa que ela tem “pedras nos rins”.

Mas o que pedras e seixos têm a ver com cálculo de quantidades? O historiador grego Herótoto, que viveu no século V a.C., reporta que “os gregos escrevem e calculam com seixo, movendo a mão da esquerda para a direita; os egípcios fazem-no ao contrário”. Ou seja, simplificada, podemos afirmar que pedras e seixos foram nossas primeiras máquinas de calcular.

Entre os povos primitivos, a computação atendia a duas necessidades primárias [5]:

1. Enumerar quantidades básicas relacionadas à agricultura e ao comércio, tais como a contagem de rebanho, troca de moedas e partilha de terras.
2. Manter um registro do tempo, das estações do ano, através de um calendário.

Embora dez e um milhão sejam quantidades muito diferentes para nós que vivemos no século XXI, para a mente de pessoas de sociedades menos complexas, ambas são “muitos”. Resquícios disso pode ser encontrado no ditado popular “um é pouco, dois é bom, três é demais” [9] [5].

1.2 O MODELO DE VON NEUMANN

1.3 ORGANIZAÇÃO DA APOSTILA

Esta apostila está organizada em três partes, conforme mostra a Figura...:

1. SISTEMAS DE REPRESENTAÇÃO. Nesta parte, abordamos como sistemas computacionais representam valores numéricos. No Capítulo 2, convidamos você a olhar de outra maneira o sistema de



Figura 2: Nas sociedades menos complexas, não havia necessidade de se expressar grandes quantidades numéricas. Em termos práticos, pouco importava a diferença entre três ou dez elementos; o importante era correr. Fonte: [2].

numeração decimal, tão presente no nosso dia-a-dia, preparando-o para compreender como outras alternativas de representação de números. Nos Capítulos 3 e 4, vemos como representar números *inteiros* e números *reais* em máquinas com memória limitada, como o seu *smartphone*, por exemplo. No Capítulo 5, você aprenderá a usar Linguagem C para manipular representações de números inteiros e ponto flutuante. No Capítulo 6, você aprenderá a fazer o mesmo, usando a Linguagem Python. Por fim, no Capítulo 7, você verá como representar outros tipos de dados em memória, tais como caracteres.

2. **CIRCUITOS COMBINATÓRIOS.** Nesta parte, cobrimos os fundamentos de projeto de circuitos lógicos e apresentamos os principais circuitos combinatórios, cujas saídas dependem unicamente da entrada. No Capítulo 8, recapitulamos os fundamentos básicos da Álgebra Booleana, em uma abordagem informal. O Capítulo 9 ... O Capítulo 10 ... O Capítulo 11 ...
3. **CIRCUITOS SEQUENCIAIS.** Nesta última parte, apresentamos os circuitos sequenciais mais importantes para sistemas computacionais e finalmente mostramos como todos os circuitos vistos são reunidos para compor um processador básico. O Capítulo 12 ... O Capítulo 13 ... Finalmente, no Capítulo 14, você compreenderá como os circuitos lógicos básicos podem ser combinados um processador simples.

Parte I

SISTEMAS DE REPRESENTAÇÃO

Sistemas computacionais utilizam uma linguagem para se comunicar. Essa linguagem é comumente chamada de *código binário*, pois utiliza apenas dois símbolos, zero e um, para representar as mais diversas mensagens.

EM TODO MOMENTO, LIDAMOS COM NÚMEROS. Qual a sua idade? E a idade do universo? Que hora é essa? Quantas calorias tem este alimento? Qual a resolução da câmera do *smartphone* na vitrine? Quantos créditos tenho no celular? Qual as dimensões de uma casa a ser reformada?

Empregamos números em uma variedade tão grande de aspectos do cotidiano que temos de registrá-los, a fim de não esquecê-los. Ou a fim de expressá-los de forma clara a alguém com quem desejamos informar essas quantidades. Ou a fim de realizar operações aritméticas com eles.

Conjuntos de símbolos tais como 153 e 42 são tão naturais para nós que não nos damos conta em que momento da história passamos a associá-los às quantidades “cento e cinquenta e três” e “quarenta e dois”. Neste capítulo, abordaremos as regras implícitas ao sistema de numeração decimal, que utilizamos no nosso dia-a-dia, dissociando a *ideia de valor* da *concretude da representação*. Em seguida, estudaremos outras possibilidades de representação de números, como o sistema binário, utilizado por microprocessadores e memórias. Por fim, veremos algumas regras de conversão entre bases numéricas.

2.1 O QUE É UM SISTEMA DE NUMERAÇÃO?

Antes de responder à pergunta título desta seção, precisamos deixar bem claras as definições de número e de numeral.



Definição 1 (Número)

Um número é *seqüência* de um ou mais símbolos que representa uma *quantidade*.




Definição 2 (Numeral)

Um numeral – também chamado *algarismo* – é um *símbolo* usado para denotar um número.

Portanto, podemos dizer que o conceito de *número* envolve a *associação* entre uma ideia *abstrata* (quantidade) e uma representação *concreta* (numeral). Contudo, estamos tão acostumados desde pequenos

a associar quantidades a símbolos, que hoje já não mais o fazemos conscientemente. Por exemplo, se você vai ao supermercado e vê sobre um produto uns rabiscos semelhantes a “15”, você logo os *associa* a quinze moedas de um real. Porém, um marciano que chegasse à Terra e visse esses mesmos rabiscos poderia achar que eles representam os anéis de Saturno, por exemplo, e não um número.

Da mesma maneira, você pode achar que o símbolo  representa um olho. Contudo, para qualquer membro alfabetizado da civilização maia, esse símbolo “obviamente” significava zero, quantidade que hoje você costuma associar ao símbolo 0.

Ao longo da história, várias associações entre valores e símbolos foram utilizadas pelas sociedades. À medida que as necessidades humanas foram se tornando mais sofisticadas (e complexas), algumas associações foram se tornando pouco práticas e por isso substituídas por outras. Por exemplo, é fácil associar o símbolo **XXI** ao século em que vivemos atualmente. Porém, determinar quantos dias se passaram nesses vinte um séculos (XXI vezes C vezes CCCLXV) já não é uma tarefa tão trivial.

Após esta discussão sobre associação de valores a símbolos escritos, estamos prontos para responder à pergunta do título desta seção:



Definição 3 (Sistema de Numeração)

Um *sistema de numeração* é um conjunto de *símbolos* e *regras*, utilizado para representar números de uma forma consistente.

Portanto, quando adotamos um sistema de numeração, arranjamos os símbolos (numerais ou algarismos) em uma sequência seguindo um conjunto de regras, de tal forma que nenhuma pessoa tenha dúvida do valor representado. Existem duas grandes classes de sistemas numéricos, divididas segundo a interpretação que se faz sobre o posicionamento dos símbolos ao longo do número escrito (concretude) para determinar o seu valor (abstração).

SISTEMAS NÃO POSICIONAIS Nestes, o valor de um símbolo é sempre o mesmo e *independe* da posição em que ocupa no número.

Exemplo: Sistema de numeração romano. O símbolo **V** expressa sempre cinco objetos, seja em IV, XV ou LXVIII.

SISTEMAS POSICIONAIS Nestes, o valor atribuído a um algarismo *depende da posição* em que ele ocupa no número.

Exemplo: Sistema decimal. O símbolo 5 expressa cinco objetos em 15 ou 1235, mas expressa cinquenta objetos em 458.

Quando você aprendeu o sistema de numeração romano, deve ter se perguntado o seguinte: *como representar números muito maiores que*

mil? De uma forma mais geral, poderíamos questionar: *O que deve ser feito quando uma sequência de símbolos se esgota, mas ainda restam objetos a ser contados?* A resposta para essa pergunta é o que veremos a seguir.

2.1.1 Esgotamento de Símbolos

Sistemas de numeração não posicionais têm o inconveniente de que apresentam um número limitado de símbolos para se representar as infinitas quantidades numéricas que nossa mente pode imaginar. Por mais que criemos mais um símbolo para representar uma quantidade presumida suficientemente grande, logo surgirá a necessidade de se representar uma quantidade ainda maior. Além do problema da *representação* de valores, ainda temos o problema da *manipulação* dos símbolos criados, por meio das operações aritméticas (ex.: adição, subtração, multiplicação, divisão, exponenciação, entre outras).

Por outro lado, os sistemas posicionais resolvem esse inconveniente estendendo a sequência de símbolos. Ou seja, quando uma sequência de símbolos se esgota, registramos esse fato em uma posição e passamos a utilizar outra posição da sequência de símbolos para contar os objetos disponíveis.

Vejamus um exemplo: um pastor conta cada uma das suas ovelhas à medida em que elas passam pelo portão da cerca. Ele usa uma pedra para representar cada ovelha. As pedras são depositadas em uma caixa dividida em dois compartimentos. Ao ver a primeira ovelha, ele põe uma pedra dentro do compartimento mais à direita da caixa.

1ª ovelha:

	1
--	---

Após ver a segunda ovelha passar, ele coloca outra pedra. E assim continua procedendo até a nona ovelha.

9ª ovelha:

	9
--	---

Porém, não há espaço para colocar uma décima pedra no mesmo compartimento da caixa, caso contrário elas transbordam e ele perde a contagem. Para contornar isso, o pastor põe uma pedra no compartimento mais à esquerda e deixa vazio o compartimento da direita:

10ª ovelha:

1	
---	--

Note a grande utilidade do zero para diferenciar os registros da primeira e da décima ovelha que atravessaram o portão da cerca. Sem o símbolo zero, tanto a primeira quanto a décima ovelhas seriam registradas apenas como 1. Se usássemos um espaço em branco, isso poderia gerar uma ambiguidade quanto à presença ou não de espaços

em branco. Se usássemos um traço ou ponto, estes símbolos teriam o mesmo papel que o zero; só convencionaríamos em usar outro signo gráfico. No número dez, o zero é usado para segurar a posição mais à direita, informando que os nove outros símbolos disponíveis foram esgotados.

Prosseguindo com sua tarefa, nosso pastor registra o seguinte, após a passagem da undécima ovelha:

11^a ovelha:

1	1
---	---

E assim ele prossegue até a décima nona ovelha. Como ele deve registrar a passagem da vigésima ovelha? Na posição da direita, ele retira as pedras, para marcar que foi esgotada a capacidade de armazenamento de pedras naquele compartimento da caixa. Em seguida, ele põe uma segunda pedra no compartimento da esquerda, indicando que dois esgotamentos foram registrados:

20^a ovelha:

2	
---	--

E assim o pastor vai procedendo até a nonagésima nona ovelha. E como ele registra a passagem da centésima ovelha pelo portão da cerca? Bem, primeiro ele passa a usar uma caixa com três compartimentos. Em seguida, ele retira as pedras da posição mais à direita e tenta acrescentar uma nova pedra ao compartimento do meio. Contudo, como lá já existem nove pedras, uma décima causaria transbordamento. Consequentemente, ele também esvazia essa posição e acrescenta uma pedra à posição mais à esquerda para marcar o transbordamento:

100^a ovelha:

1		
---	--	--

2.1.2 Base Numérica

Durante a contagem das ovelhas, o pastor passou a utilizar os compartimentos mais à esquerda da caixa porque só cabiam nove pedras em cada compartimento. Consequentemente, cada compartimento da caixa tinha *dez* estados possíveis: nenhuma pedra, uma pedra, duas pedras, ..., nove pedras. Ou seja, o número dez serviu de *base* para a contagem das ovelhas.

Se a capacidade da caixa fosse de apenas oito pedras, o registro seria diferente. Após a passagem da oitava ovelha, não caberiam mais pedras no compartimento da direita, o qual deveria ser esvaziado, exigindo que uma pedra fosse colocada no compartimento da esquerda:

8ª ovelha:

1	
---	--

De modo semelhante ao pastor ante a passagem das ovelhas, um computador também conta a passagem de bits. A diferença é que cada posição comporta apenas uma única “pedra”, ou seja, o computador utiliza apenas dois símbolos para representar valores. Tal analogia nos ajuda a entender a definição de base numérica:

**Definição 4 (Base numérica)**

A *base* de um sistema de numeração posicional define o número de algarismos distintos utilizados para representar números.

A partir dessa definição, tiramos as seguintes conclusões sobre um sistema numérico de base B qualquer:

1. Um sistema numérico de base B utiliza B símbolos distintos para representar qualquer grandeza.
2. O *menor* valor representável com um único símbolo em um sistema numérico de base B é *zero*.
3. O *maior* valor representável com um único símbolo em um sistema numérico de base B é $B - 1$.
4. Números que contêm o algarismo zero (0) na posição mais à direita são múltiplos da base B , pois marcam o momento do esgotamento de símbolos que ocorre durante a contagem de cada B objetos.

Nas subseções a seguir, trataremos algumas bases especiais como exemplo. Começaremos pela base decimal, da qual destacaremos algumas características que nos passam despercebidas no dia-a-dia. As bases binária e hexadecimal são muito utilizadas em computação. Já a base dozenal é apresentada a título de curiosidade.

2.1.3 O Sistema Decimal de Numeração

Na maior parte das aplicações do cotidiano, utilizamos o sistema decimal de numeração para representar quantidades: número de bombons em uma embalagem, quantidade de bytes em um *pen-drive*, área de uma superfície, entre outras.

Para nós que vivemos no século XXI, o sistema decimal nos parece tão simples e trivial que o senso comum nos leva a acreditar que a humanidade sempre pensou assim desde que começou a contar. Entretanto, o sistema decimal foi inventado pelos hindus por volta

do século IV. Ainda assim, ele só foi difundido na Europa no século XII, através dos árabes [13].

Provavelmente, o fato de termos dez dedos nas duas mãos deve ter influenciado nossos antepassados a contar grandezas com base em dez símbolos, em uma correspondência biunívoca para cada um dos dez dedos das mãos [13]. Cada objeto contado era associado a um dedo das mãos. Quando o número de objetos ultrapassava dez, registrava-se o esgotamento de dedos e recomeçava-se a contagem, de modo semelhante ao que discutimos no exemplo sobre contagens de ovelhas, na Seção 2.1.1.

Outros sistemas de numeração foram utilizados por diversas civilizações ao longo da história. Os maias, por exemplo, contavam objetos associando-os aos dedos das mãos e dos pés, de modo que seu sistema de numeração tinha base vinte. Acredita-se que os babilônicos utilizavam as três falanges dos quatro dedos de apenas uma das mãos para contar, em um sistema de numeração de base $3 \times 4 = 12$. Assim, o polegar ficava livre para marcar a contagem e a outra mão também ficava livre, podendo ser usada para manusear os objetos contados. O fato é que, entre essas e outras alternativas de contagem, a base dez prevaleceu.

A base decimal utiliza dez símbolos para representar grandezas numéricas:

0 1 2 3 4 5 6 7 8 9

Note que cada símbolo (algarismo) utilizado é uma unidade maior que o seu predecessor: por exemplo, 3 é uma unidade maior que 2. Embora pareça óbvio, isso não acontece no sistema de numeração romano. Note que o símbolo V é quatro unidades superior ao seu predecessor, I.

Para representar grandezas numéricas, utilizamos uma sequência de um ou mais dos dez símbolos disponíveis. Eles são posicionados um ao lado do outro, como no exemplo abaixo.

5	3	9	7
---	---	---	---

Como vimos anteriormente, o sistema de numeração decimal é um sistema *posicional*, pois o valor de um algarismo é determinado pela sua posição relativa no número. Por exemplo, sabemos que 7 é maior que 3, se tomados individualmente. Entretanto, no número 5397 representado acima, o algarismo 3 tem valor de **trezentos**, e não de apenas **três**, como na comparação inicial.

O algarismo localizado na posição mais à *esquerda* do número é conhecido como *algarismo mais significativo* (MSD – *most significant digit*). Ele representa a maior contribuição de um algarismo individual para o valor total do número representado. Por outro lado, o algarismo

localizado na posição mais à *direita* do número recebe o nome de *algarismo menos significativo* (LSD – *least significant digit*). Ele representa a menor contribuição de um algarismo individual para o valor total do número representado. No exemplo anterior, o algarismo 5 é o mais significativo e o algarismo 7 é o menos significativo.

5	3	9	7
↓			↓
MSD			LSD



Atenção: Último algarismo?

É muito comum alunos iniciantes fazerem referências ao “primeiro algarismo” ou ao “último algarismo” de um número. Porém, esses termos não têm expressividade alguma, pois os ordinais “primeiro” e “último” carecem de um ponto de referência. Portanto, quando for se comunicar com um colega, utilize apenas as expressões *algarismo mais significativo* e *algarismo menos significativo*.

No caso da base decimal, as posições são chamadas de *casas decimais*. O valor com que cada algarismo contribui individualmente para o valor numérico total representado depende de duas coisas:

1. O valor do algarismo em si.
2. A posição que o algarismo ocupa no número.

Assim, para encontrar o valor de um algarismo que ocupa a casa das *unidades* (U), multiplicamo-lo por **um**. Na casa das *dezenas* (D), multiplicamos o algarismo por **dez**. Na casa das *centenas* (C), multiplicamos por **cem**. Na casa dos *milhares* (M), multiplicamos por **mil**, e assim por diante.

M	C	D	U
5	3	9	7

No exemplo acima, para encontrar o valor N representado pelo conjunto de símbolos **5397**, procedemos assim:

$$N = 5 \cdot 1000 + 3 \cdot 100 + 9 \cdot 10 + 7 \cdot 1 = 5397 \quad (1)$$

Parece um tanto óbvio à primeira vista. Porém, a eq. 1 nos diz que, para encontrar o *valor* representado por um conjunto de símbolos na

base decimal, devemos multiplicar o valor de *cada símbolo* pelo *peso da posição* que ele ocupa no número. No exemplo acima, o peso da posição das centenas é **cem**, enquanto que o peso da posição das unidades é apenas **um**.

Quando identificamos as casas decimais por letras (M, C, D, U), começamos a ter dificuldades para referenciar posições mais significativas como dezenas de milhares, centenas de milhões, etc. Já que um número natural tem infinitas casas decimais, é mais adequado identificá-las usando números inteiros positivos:

3	2	1	0
5	3	9	7

Observe os índices numéricos que identificam as casas decimais. Note que eles correspondem ao expoente ao qual a base dez é elevada quando determinamos o peso com que cada casa contribui para o valor total representado pela sequência de símbolos:

$$N = 5 \cdot 10^3 + 3 \cdot 10^2 + 9 \cdot 10^1 + 7 \cdot 10^0 = 5397 \quad (2)$$

Pela eq. 2, note que cada casa decimal representa dez vezes mais aquilo que representa a casa decimal à sua direita, e um décimo do valor representado pelo casa decimal à sua esquerda.

O zero

O numeral 0 (zero) apresenta uma peculiaridade adicional em relação aos demais símbolos usados na base decimal. O zero representa o nada, mas isso não significa que ele não represente coisa alguma. Outra forma de entender essa colocação é remetendo ao conceito de cardinalidade. Na matemática, a cardinalidade indica uma medida do número de elementos de um conjunto. Dessa forma, o zero, no papel de *número*, representa a cardinalidade do conjunto vazio.

Além do papel de número, o símbolo 0 (zero) desempenha o papel de *porta-lugar* na numeração posicional. Nesse sentido, o zero tem a função de marcar que a posição onde ele se encontra não contribui para o valor do número, ou seja, que aquela posição está vazia. Veja que se o zero fosse omitido, poderia haver ambiguidade na interpretação do valor representado. Por exemplo, usando espaços em branco no lugar do zero, como entenderíamos se 1 3 representa 13, 103 ou 1003? Ainda que você sugerisse o uso de um traço (—), estaria apenas propondo um novo design para o número zero, mas não estaria retirando seu papel de porta-lugar.

Para entender melhor o papel do zero, tomemos um exemplo de John Gregg [8]. Considere o número 608. O que o 0 quer dizer na casa das dezenas? Alguém poderia dizer que não há dezenas no número 608, mas isso não é correto. Há na verdade 60 dezenas em 608. Então,

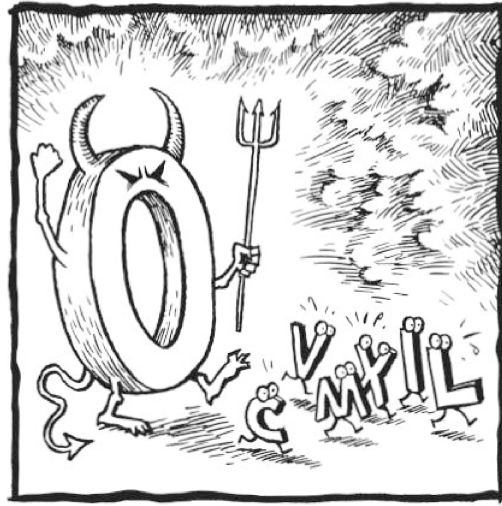


Figura 3: A invenção do número zero facilitou a realização de operações aritméticas, de modo que o sistema de numeração romano foi substituído pelo sistema indo-arábico. Fonte: [2].

o que o 0 na casa das dezenas realmente diz é que não sobraram grupos de dezenas depois que as agrupamos em centenas. Por outro lado, o 2 na casa das dezenas em 628 indica que sobraram dois grupos de dezenas depois que elas foram agrupadas em centenas. Similarmente, o 6 na posição das centenas indica que houve 6 grupos de cem que não puderam ser agrupados em uma coleção de dez centenas, ou seja, em milhares.

Outra propriedade do zero é sua *invisibilidade* nas infinitas posições à esquerda de um número. Ou seja, 4 é a mesma coisa que 0000004. Por conveniência, não precisamos escrever os infinitos zeros à esquerda do dígito mais significativo diferente de zero. Todos esses zeros indicam que a quantidade representada não se pode ser agrupada em coleções de 10, de 100, de 1000, etc.

No entanto, em certas aplicações, precisamos escrever esses zeros à esquerda, para evitar ambiguidades. É quando precisamos, por exemplo, preencher formulário ou visualizar o conteúdo da memória de computador.

2.1.4 O Sistema Binário

Na base dois – também conhecida como base *binária* – os números são expressos por uma sequência formada por algarismos 0 e 1. Como vimos na discussão sobre bases (Seção 2.1.2, p. 10), uma base B possui B símbolos disponíveis para representar valores. Logo, a base 2 dispõe de apenas dois símbolos para representar números.

A base binária é amplamente utilizada por sistemas computacionais. Um dos principais motivos é que seu uso facilita a *detecção e a*

correção de erros. Como? Durante a transmissão e processamento de dados, estes são representados fisicamente por sinais eletromagnéticos. Estes, por sua vez, estão suscetíveis a diversas formas de ruído. No receptor, o sinal recebido com distorção pode não corresponder a nenhum dos sinais previamente convencionados pelo sistema de comunicação. Se você tem dez símbolos na sua convenção, seu trabalho é decidir com qual dos dez símbolos o sinal distorcido mais se parece. Já se você dispõe de apenas dois símbolos, tal comparação é simplificada e estará menos propensa ao erro. Eis a grande vantagem do uso da base binária.

E como contar na base binária? Voltemos ao exemplo do pastor e suas ovelhas. Desta vez, ele conta suas ovelhas no sistema binário. Consequentemente, somente é possível colocar uma única pedra em cada compartimento da caixa. Quando a primeira ovelha passar pelo portão da cerca, é fácil registrar:

1ª ovelha:

			1
--	--	--	---

E quando a segunda ovelha passar? Ora, teremos um esgotamento de símbolos no compartimento menos significativo. Ou seja, não há espaço para uma segunda pedra no compartimento. Logo, devemos zerar o compartimento menos significativo após a passagem de *duas* ovelhas. Não esperaremos mais dez ovelhas, como no exemplo original. Esse transbordamento deverá ser registrado no compartimento logo à esquerda:

2ª ovelha:

		1	0
--	--	---	---

A terceira ovelha é fácil de registrar, pois basta colocar uma pedra no compartimento vazio da direita:

3ª ovelha:

		1	1
--	--	---	---

Para registrar a passagem da quarta ovelha, note que teremos dois transbordamentos. No compartimento mais à direita, retiramos a pedra e acrescentamos outra logo à sua esquerda para indicar o transbordamento:

4ª ovelha:

		1 + 1	0
--	--	-------	---

No entanto, não há espaço para mais uma pedra no segundo compartimento localizado à esquerda. Assim, retiramos a pedra dele e acrescentamos outra pedra logo à sua esquerda para indicar o transbordamento:

4ª ovelha:

	1	0	0
--	---	---	---

E assim nosso pastor vai contando até a sétima ovelha:

7ª ovelha:

	1	1	1
--	---	---	---

Nesse ponto, aparece uma oitava ovelha, que causa um triplo transbordamento entre os compartimentos da caixa:

8ª ovelha:

1	0	0	0
---	---	---	---

A esta altura, você já deve ter percebido que a caixa usada pelo pastor permite contar até quinze ovelhas:

15ª ovelha:

1	1	1	1
---	---	---	---

Como fazer para contar um milhão de ovelhas? Bom, vamos imaginar que o pastor do exemplo também é marceneiro nas horas vagas e pode construir tantos compartimentos na caixa quanto permite toda a madeira existente no planeta:

zilionésima ovelha:

0	1	...	0	1	1	0	1
---	---	-----	---	---	---	---	---



Números pares e ímpares na base dois

A partir da conclusão 4 na Seção 2.1.2 (p. 11), observe que números pares expressos na base dois sempre terminam em zero (algarismo menos significativo). Por outro lado, números ímpares sempre terminam em um. Generalizando, números divisíveis por 2^n , quando expressos em binário, devem terminar em n zeros.

2.1.5 O Sistema Dozenal

Alguns pensadores e matemáticos têm agurmentado ao longo da história que 12 seria uma base melhor para a construção de um sistema numérico, pois é um número mais versátil que o dez [2]. O motivo

dessa vantagem está na divisibilidade. O número doze pode ser dividido por 2, 3, 4 e 6, ao passo que 10 pode ser dividido por apenas 2 e 5. Os defensores da base 12 argumentam que somos muito mais propensos a querer dividir por 3 ou 4 do que dividir por 5 em nosso cotidiano.

Alex Bellos cita o seguinte exemplo [2]. “Imagine um comerciante. Se você tiver 12 maçãs, então você pode dividi-las em duas sacolas de 6, três sacolas de 4, quatro sacolas de 3, ou seis sacolas de 2. É muito mais fácil do que 10, que só pode ser dividido de forma exata em dois sacolas de 5, ou cinco sacolas de 2”.

Não é à toa que a dúzia ainda seja muito utilizada no comércio, pois elimina algumas frações, facilitando o cálculo mental de quantidades a pagar. Por exemplo, um terço de 10 é $3,33\dots$, uma dízima, ao passo que um terço de 12 é somente 4. Embora menos conhecida da maioria das pessoas, a unidade *grosa* é também utilizada no comércio atacadista, correspondendo a um conjunto de doze dúzias, ou seja, 144 unidades.

Por conta das vantagens citadas acima, alguns entusiastas ao redor do mundo costumam adotar o sistema dozenal para uso particular. Alguns chegam até a se organizarem em sociedades, como a *The Dozenal Society of America*.

www.dozenal.org

O sistema dozenal tem base 12. Portanto, este sistema possui doze dígitos: 0 a 9, emprestados do sistema decimal comum, e dois símbolos extras, chamados de *transdecimais*. O valor dez, chamado *dek*, é representado por χ . O valor onze, chamado *el*, é representado por Σ .

Assim, na base dozenal, cabem doze pedras em cada compartimento da caixa. Dessa forma, para um pastor deve registrar a passagem das overlhas pela cerca da seguinte maneira:

9 ^a ovelha:	<table border="1"><tr><td></td><td>9</td></tr></table>		9
	9		
10 ^a ovelha:	<table border="1"><tr><td></td><td>χ</td></tr></table>		χ
	χ		
11 ^a ovelha:	<table border="1"><tr><td></td><td>Σ</td></tr></table>		Σ
	Σ		
12 ^a ovelha:	<table border="1"><tr><td>1</td><td>0</td></tr></table>	1	0
1	0		
13 ^a ovelha:	<table border="1"><tr><td>1</td><td>1</td></tr></table>	1	1
1	1		
\vdots	\vdots		
21 ^a ovelha:	<table border="1"><tr><td>1</td><td>9</td></tr></table>	1	9
1	9		
22 ^a ovelha:	<table border="1"><tr><td>1</td><td>χ</td></tr></table>	1	χ
1	χ		

23ª ovelha:

1	Σ
---	---

24ª ovelha:

2	0
---	---

2.1.6 O Sistema Hexadecimal

O sistema hexadecimal, assim como o sistema binário, também é bastante utilizado em sistemas computacionais. Sua base de contagem é o 16. Assim, dezesseis símbolos são usados para representar os números. Para representar os valores de zero a nove, usamos os mesmos símbolos da base decimal. Já para representar os valores de dez a quinze, utilizamos as seis primeiras letras do alfabeto latino: A, B, C, D, E, F, de modo que $A = 10$, $B = 11$, $C = 12$, $D = 13$, $E = 14$, e $F = 15$.

Diferentemente do sistema dozenal, o sistema hexadecimal não dispõe de símbolos e nomes especiais para os valores de dez a quinze. Provavelmente isso se deve à sua origem ser de ordem mais pragmática, enquanto que a do sistema dozenal ser mais ideológica.



Qual a aplicação prática do sistema hexadecimal?

Números binários podem ser longos. Por exemplo, o número 58 na base decimal é expresso como 111010 na base binária. Por outro lado, o sistema decimal – embora seja mais compacto – não exprime diretamente valores armazenados em sistemas computacionais. Já o sistema hexadecimal permite que cada quatro dígitos binários sejam expressos por um só dígito hexa, permitindo expressar de forma concisa números representados em máquina. Veja a Seção 2.5 (p. 28) para mais detalhes.

Desse modo, um pastor que adote o sistema hexadecimal, poderá guardar apenas quinze pedras em cada compartimento de sua caixa de contagem, registrando assim a passagem das suas ovelhas:

9ª ovelha:

	9
--	---

10ª ovelha:

	A
--	---

11ª ovelha:

	B
--	---

⋮

15ª ovelha:

	F
--	---

16ª ovelha:	<table border="1"><tr><td>1</td><td>0</td></tr></table>	1	0
1	0		
17ª ovelha:	<table border="1"><tr><td>1</td><td>1</td></tr></table>	1	1
1	1		
⋮	⋮		
25ª ovelha:	<table border="1"><tr><td>1</td><td>9</td></tr></table>	1	9
1	9		
26ª ovelha:	<table border="1"><tr><td>1</td><td>A</td></tr></table>	1	A
1	A		
⋮	⋮		
159ª ovelha:	<table border="1"><tr><td>9</td><td>F</td></tr></table>	9	F
9	F		
160ª ovelha:	<table border="1"><tr><td>A</td><td>0</td></tr></table>	A	0
A	0		
⋮	⋮		
255ª ovelha:	<table border="1"><tr><td>F</td><td>F</td></tr></table>	F	F
F	F		

Resumo

A Tabela 1 apresenta uma contagem de 0 a 32 em suas representações nas bases decimal, binária, dozenal e hexadecimal. Note que outras bases também são possíveis: vinte, utilizada pelos maias; sessenta, utilizada pelos babilônicos; mil duzentos e trinta e quatro, sem utilidade conhecida.

Bases não pertencentes ao conjunto dos números naturais também são possíveis: bases negativas, fracionárias e irracionais. Infelizmente, elas estão fora do escopo desta apostila. O leitor curioso pode buscar mais informações em Koren [17] ou Parhami [21].

2.2 CONVERSÃO DE UMA BASE b PARA A BASE DECIMAL

Até o momento, vimos como contar em diferentes bases numéricas, partindo do zero em incrementos de um em um. Por outro lado, existem situações em que temos um número expresso em uma base B qualquer e precisamos conhecer o valor dele na base decimal. Dessa forma, como podemos converter um número de uma base B para a conhecer seu valor na base decimal?

Para responder a essa pergunta, considere um número N qualquer na base 10, contendo n casas decimais. Logo, sua representação é dada por uma sequência de n algarismos, indicados por a_i , com $0 \leq i < n$:

a_{n-1}	a_{n-2}	\cdots	a_1	a_0
-----------	-----------	----------	-------	-------

Tabela 1: Representações dos valores de 0 a 32 nas bases binária, decimal, dozenal e hexadecimal.

Decimal	Binário	Dozenal	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	8	8
9	1001	9	9
10	1010	ℵ	A
11	1011	£	B
12	1100	10	C
13	1101	11	D
14	1110	12	E
15	1111	13	F
16	10000	14	10
17	10001	15	11
18	10010	16	12
19	10011	17	13
20	10100	18	14
21	10101	19	15
22	10110	1ℵ	16
23	10111	1£	17
24	11000	20	18
25	11001	21	19
26	11010	22	1A
27	11011	23	1B
28	11100	24	1C
29	11101	25	1D
30	11110	26	1E
31	11111	27	1F
32	100000	28	20

Da discussão anterior sobre o sistema decimal (Seção 2.1.3), sabemos que o *valor* do número N representado acima será dado pela soma do produto de cada algarismo a_i pelo peso de sua posição i . Ora, quando trabalhamos com a base decimal em específico, esse peso é dado por 10^i .

Generalizando esse procedimento, temos que, se N está expresso em uma base B qualquer, então seu valor na base decimal, denotado por $(N)_{10}$, é expresso pela seguinte fórmula:

$$(N)_{10} = a_{n-1} \cdot B^{n-1} + a_{n-2} \cdot B^{n-2} + \dots + a_1 \cdot B^1 + a_0 \cdot B^0 \quad (3)$$

Note que *todos os cálculos devem ser realizados na aritmética da base de destino*. Ou seja, cada algarismo a_i da base B e o próprio valor da base B devem ser expressos na base 10. Essa exigência se explica pelo fato de que *só podemos operar números se todos eles estiverem em uma mesma base numérica*. Afinal, um mesmo símbolo (numeral) pode ter significados diferentes (valores) quando operamos em bases distintas.

A eq. 3 pode ser interpretada como um polinômio, onde cada algarismo a_i é visto como um coeficiente e a base B de origem, como a variável do polinômio. Por conta disso, esse método de conversão de bases é conhecido como *polinomial*.

2.2.1 Conversão Binário-Decimal

A eq. 3 vale para qualquer base B , inteira ou não. Aqui vamos particularizá-la para $B = 2$, ou seja, vamos abordar especificamente a conversão da base binária – amplamente utilizada em sistemas computacionais – para a base decimal.

Para converter um número expresso em base binária para a base decimal, primeiro você deve numerar as posições de cada algarismo do número em binário, para identificar os pesos. Você deve numerar do algarismo menos significativo para o mais significativo, ou seja, da direita para a esquerda. A numeração começa em zero, pois a contribuição da posição mais à direita é sempre $B^0 = 1$. Por fim, basta aplicar a equação 3, onde cada $a_i \in \{0,1\}$ e $B = 2$. Vejamos alguns exemplos a seguir.

Exemplo 2.1. Converter $(1011)_2$ para a base dez.

3	2	1	0
1	0	1	1

$$(N)_{10} = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

$$(N)_{10} = 8 + 0 + 2 + 1 = 11$$

□

Exemplo 2.2. Converter $(10100)_2$ para a base dez.

4	3	2	1	0
1	0	1	0	0

$$(N)_{10} = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$$

$$(N)_{10} = 16 + 0 + 4 + 0 + 0 = 20$$

□

Note que apenas as posições do número binário que contém 1s são as que realmente contribuem para o cálculo da conversão de base. As posições que contém o algarismo 0 têm peso nulo. Portanto, uma maneira mais prática de converter um número binário para a base decimal é somar as potências de 2 correspondentes apenas às posições que contém o algarismo 1. Veja o exemplo a seguir.

Exemplo 2.3. Converter $(110001)_2$ para a base dez.

5	4	3	2	1	0
1	1	0	0	0	1

$$(N)_{10} = 2^5 + 2^4 + 2^0$$

$$(N)_{10} = 32 + 16 + 1 = 49$$

□

2.2.2 Conversão Dozenal-Decimal

Para converter um número expresso em base dozenal para a base decimal, seguimos os mesmos procedimentos anteriores:

1. Numerar as posições de cada algarismo do número na base dozenal, da direita para a esquerda, identificando os pesos.
2. Aplicar a equação 3, onde desta vez cada $a_i \in \{0, 1, 2, \dots, 8, 9, \mathbb{X}, \mathbb{E}\}$ e $B = 12$.

Vejamos alguns exemplos:

Exemplo 2.4. Converter $(123)_{12}$ para a base dez.

2	1	0
1	2	3

$$(N)_{10} = 1 \cdot 12^2 + 2 \cdot 12^1 + 3 \cdot 12^0$$

$$(N)_{10} = 144 + 24 + 3 = 171$$

□

Exemplo 2.5. Converter $(1\chi\xi)_{12}$ para a base dez.

2	1	0
1	$\chi (= 10_{10})$	$\xi (= 11_{10})$

$$(N)_{10} = 1 \cdot 12^2 + 10 \cdot 12^1 + 11 \cdot 12^0$$

$$(N)_{10} = 144 + 120 + 11 = 275$$

□

2.2.3 Conversão Hexadecimal-Decimal

Para converter um número da base hexadecimal para a base decimal, adotamos os mesmos procedimentos anteriores, tomando $B = 16$. Vejamos alguns exemplos:

Exemplo 2.6. Converter $(123)_{16}$ para a base dez.

2	1	0
1	2	3

$$(N)_{10} = 1 \cdot 16^2 + 2 \cdot 16^1 + 3 \cdot 16^0$$

$$(N)_{10} = 256 + 32 + 3 = 291$$

□

Exemplo 2.7. Converter $(ABC)_{16}$ para a base dez.

2	1	0
$A (= 10_{10})$	$B (= 11_{10})$	$C (= 12_{10})$

$$(N)_{10} = 10 \cdot 16^2 + 11 \cdot 16^1 + 12 \cdot 16^0$$

$$(N)_{10} = 2560 + 176 + 12 = 2748$$

□

2.3 CONVERSÃO DA BASE DECIMAL PARA UMA BASE b

Agora, nosso problema é inverso ao da Seção 2.2 anterior: converter um número da base decimal para uma base B . Para resolvê-lo, um caminho seria usar o método polinomial da eq. 3. Contudo, primeiro teríamos que converter o valor dos algarismos a_i e o valor da base B para a notação da base B , e em seguida realizar adições, multiplicações e potenciações nessa base, o que não é muito intuitivo para nós que estamos acostumados a fazer contas na base decimal. Veja o exemplo a seguir:

Exemplo 2.8. Converter $(13)_{10}$ para a base hexadecimal.

1	0
1	3

Na base hexadecimal, o valor $(10)_{10}$ é expresso como A . Aplicando a eq. 3, temos:

$$\begin{aligned}(N)_{16} &= 1 \cdot A^1 + 3 \cdot A^0 \\ (N)_{16} &= A + 3 = D\end{aligned}$$

□

Note que, embora correto, o método polinomial não é muito prático para converter números da base 10 para outra base B . Se você ainda não encontrou a dificuldade, imagine então como seria converter o valor $(123)_{10}$ para hexadecimal. Teríamos que resolver a expressão $A^2 + 2 \cdot A + 3$ na base hexadecimal, sem recorrer à base 10.

Um método mais prático para converter números expressos na base decimal para outra base B qualquer é o das *divisões sucessivas*, descrito na Figura 4. Por esse método, o número N_{10} a ser convertido é dividido pela nova base B na aritmética da base decimal, à qual estamos familiarizados. O resto dessa divisão forma o algarismo mais à direita (menos significativo) do número convertido à nova base. O quociente é novamente dividido por B , e assim sucessivamente, até o quociente ser zero. A sequência de todos os restos forma o número na nova base B , do algarismo mais significativo para o menos significativo.

Para entender como o método funciona, vamos dividir o número N_{10} da eq. 3 pela base B :

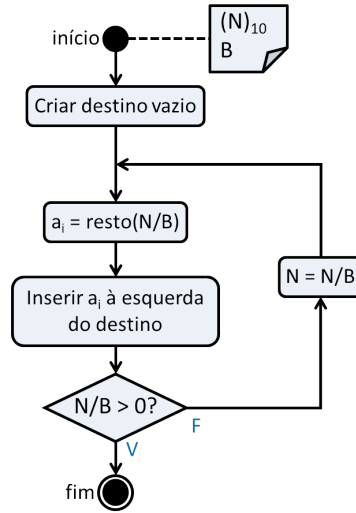


Figura 4: Algoritmo de conversão de um número expresso na base decimal para uma base B qualquer.

$$\begin{aligned}
 N_{10} &= a_{n-1} \cdot B^{n-1} + a_{n-2} \cdot B^{n-2} + \dots + a_1 \cdot B^1 + a_0 \cdot B^0 \\
 N_{10} &= (a_{n-1} \cdot B^{n-2} + a_{n-1} \cdot B^{n-3} + \dots + a_2 \cdot B + a_1) \cdot B + a_0 \\
 \frac{N_{10}}{B} &= \underbrace{(a_{n-1} \cdot B^{n-2} + a_{n-2} \cdot B^{n-3} + \dots + a_2 \cdot B + a_1)}_{Q_1}, \text{ com resto } a_0 \\
 Q_1 &= (a_{n-1} \cdot B^{n-3} + a_{n-2} \cdot B^{n-4} + \dots + a_2) \cdot B + a_1 \\
 \frac{Q_1}{B} &= \underbrace{(a_{n-1} \cdot B^{n-3} + a_{n-2} \cdot B^{n-4} + \dots + a_2)}_{Q_2}, \text{ com resto } a_1 \\
 &\vdots
 \end{aligned}$$

Dividindo-se sucessivamente os quocientes, obteremos como resto os valores $a_0, a_1, \dots, a_{n-2}, a_{n-1}, 0$, nessa ordem. Portanto, o número em base B será composto pelos valores a_i convertidos à base B de destino, arranjados na ordem inversa com que foram obtidos.

Uma vez que o resto da divisão é menor que o divisor, temos que $a_i < B, \forall i \in [0, n-1]$. Ainda que alguns valores a_i sejam expressos com mais de um algarismo na base 10, eles serão expressos com apenas um único símbolo na base B de destino.

Vejamos alguns exemplos:

Exemplo 2.9. Converter $(53)_{10}$ para a base binária.

Embora pareçam diferentes, ambos os métodos derivam da eq. 3. Observe na Figura 5 como eles se integram. A opção por um ou por outro vai depender da sua familiaridade com a aritmética da base Z de origem (divisões sucessivas) ou com a base B de destino (polinomial).

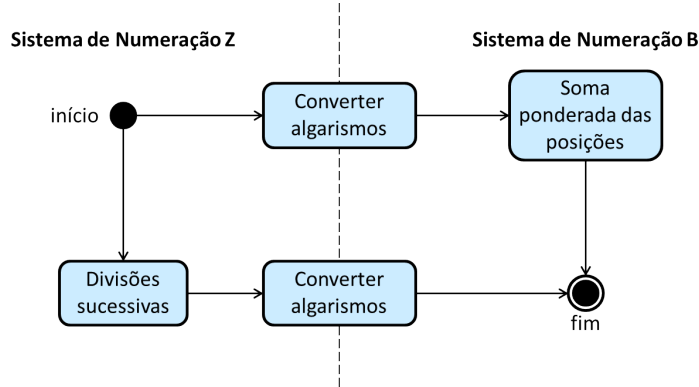


Figura 5: Algoritmo geral para conversão de bases. Fonte: [1].

2.5 INSTANCIANDO: QUANDO z É UMA POTÊNCIA DE b

Vimos que, para transformar um número N da base decimal para a binária, devemos dividi-lo sucessivamente por 2, até que o quociente seja zero. O resto de cada operação de divisão representa um algarismo do número resultante na base dois. Portanto, o número de divisões realizadas até o fim do procedimento determinará a quantidade de bits do número resultante.

Uma vez que sistemas computacionais têm espaço limitado de memória, é importante conhecermos qual a relação entre a quantidade de algarismos de um número expresso em base decimal e a quantidade de algarismos desse mesmo número quando expresso em base binária. Ou seja, temos de responder à seguinte pergunta. Qual a quantidade d de bits necessária para representar um número N na base binária, dado que N contém m algarismos na base decimal?

Para começar a responder, consideremos o seguinte exemplo. Se um número natural N tem *três* algarismos quando expresso em base dez, então ele é maior ou igual a 100 e menor do que 1000. Logo, para um número N qualquer de m algarismos, temos:

$$10^{m-1} \leq N < 10^m$$

Uma vez que $x = y^{\log_y x}$, $\forall y > 0$, temos que:

$$\begin{aligned} 2^{\log_2 10^{m-1}} &\leq N < 2^{\log_2 10^m} \\ 2^{(m-1) \cdot \log_2 10} &\leq N < 2^{m \cdot \log_2 10} \end{aligned}$$

Logo, a quantidade d de bits do número N convertido em binário será um valor pertencente à seguinte faixa de valores:

$$\lceil (m-1) \cdot \log_2 10 \rceil \leq d < \lceil m \cdot \log_2 10 \rceil \quad (4)$$

A função teto é utilizada na eq. 4 porque d indica quantidade de bits, ou seja, tem de ser um número natural, mas o resultado da operação \log pode fornecer um número real não-inteiro.

Exemplo 2.12. Quantos bits tem o número 53 quando convertido à base binária?

RESPOSTA. Já que 53 tem dois dígitos, então $m = 2$. Da eq. 4, temos:

$$\begin{aligned} \lceil (2-1) \cdot \log_2 10 \rceil &\leq d < \lceil 2 \cdot \log_2 10 \rceil \\ \lceil 3,322 \rceil &\leq d < \lceil 6,644 \rceil \\ 4 &\leq d < 7 \end{aligned}$$

Logo, 53 em binário tem de 4 a 7 bits. Ora, do Exemplo 2.9, sabemos que são exatamente seis bits, o que corrobora com nossa expectativa. \square

Exemplo 2.13. Qual a ordem de grandeza do valor de $(111010101101)_2$?

RESPOSTA. O número acima possui 12 bits. Adaptando a eq. 4, temos:

$$\begin{aligned} \lceil (12-1) \cdot \log_{10} 2 \rceil &\leq d < \lceil 12 \cdot \log_{10} 2 \rceil \\ \lceil 11 \cdot 0,301 \rceil &\leq d < \lceil 12 \cdot 0,301 \rceil \\ \lceil 3,311 \rceil &\leq d < \lceil 3,612 \rceil \\ 4 &\leq d < 4 \end{aligned}$$

Logo, o número binário acima possui 4 casas decimais quando convertido em base dez. Ou seja, é da ordem de grandeza de milhares. \square

2.5.1 Conversão da Base Binária para Hexadecimal

A partir da eq. 4, podemos dizer que cada casa decimal equivale a $\log_2 10 \approx 3,322$ dígitos binários. Ou seja, a cada três algarismos decimais que representam um valor numérico, usamos aproximadamente dez bits na notação binária equivalente (relação de 3,333...). É por isso que os fabricantes de memórias de armazenamento afirmam que 1024 Bytes ($= 2^{10}$) equivalem a aproximadamente 1000 Bytes ($= 10^3$).

No entanto, a relação não é exata e somente é aplicada em estimativas. Se, na eq. 4, em vez de $\log_2 10$ tivéssemos $\log_2 2^n$, a relação entre os dígitos da base 2 e da base 2^n seria *exatamente* de n para 1. Dessa

A função teto, denotada por $\lceil x \rceil$, converte um número real x no menor número inteiro maior ou igual a x .

A função chão, denotada por $\lfloor x \rfloor$, converte um número real x no maior número inteiro menor ou igual a x . Para números positivos, a função chão é mais conhecida como truncamento.

forma, para converter valores de uma base B para outra B^n , $B, n \in \mathbb{N}$, não precisamos converter da base B para a base decimal, e depois desta para a base B^n .

Para converter números expressos na base B para a base B^n , basta tomarmos cada n dígitos de N_B e convertê-los em um dígito na base B^n . Similarmente, na conversão de B^n para B , basta tomarmos cada dígito de N_{B^n} e convertê-lo em n dígitos na base B .

Exemplo 2.14. Converter $X = (101010110010000001101100)_2$ para a base hexadecimal.

Primeiro devemos arrumar os dígitos de X em grupos de $\log_2 16 = 4$ bits, do bit menos significativo (direita) para o mais significativo (esquerda). Em seguida, convertemos cada grupo de 4 bits em um único dígito hexadecimal:

1010	1011	0010	0000	0110	1100
↓	↓	↓	↓	↓	↓
A	B	2	0	6	C

Portanto, o resultado é $X = (AB206C)_{16}$.

□

Exemplo 2.15. Converter $X = (1011011001)_2$ para a base hexadecimal.

Mais uma vez, arrumamos os dígitos de X em grupos de 4 bits, da direita para a esquerda. Se o grupo mais significativo tiver menos de 4 bits, completamos o restante com zeros à esquerda:

0010	1101	1001
↓	↓	↓
2	D	9

Portanto, o resultado é $X = (2D9)_{16}$.

□

2.5.2 Conversão da Base Hexadecimal para a Binária

Para converter um número expresso na base hexadecimal para a base binária, basta procedermos inversamente ao que vimos na Seção 2.5.1 anterior: do dígito menos para o mais significativo, tomamos cada dígito hexadecimal e o convertemos em 4 bits. Ainda que seja um número que não ocupe 4 bits, como o $(5)_{16} = (101)_2$, preenchemo-lo de zeros à esquerda do conjunto, até que ele forme um conjunto de 4 bits.

Exemplo 2.16. Converter $X = (3A9B17)_{16}$ para a base binária.

Devemos converter cada dígito hexadecimal em 4 bits:

3	A	9	B	1	7
↓	↓	↓	↓	↓	↓
0011	1010	1001	1011	0001	0111

Portanto, o resultado é $X = (11\ 1010\ 1001\ 1011\ 0001\ 0111)_2$.

□

2.6 OPERAÇÕES ARITMÉTICAS COM NÚMEROS BINÁRIOS

Quando pequenos, tivemos de aprender (ou decorar) a tabuada a fim de nos tornarmos capazes a fazer operações aritméticas. Cada operação aritmética básica (adição, subtração, multiplicação e divisão) tem sua própria tabuada a ser memorizada pelos alunos. Mas o que é a tabuada? Trata-se de uma tabela contendo o resultado da operação entre um número de 1 a 10 por sucessivos números de 1 a 10. Dessa forma, para cada uma das quatro operações aritméticas, tivemos de decorar 100 resultados. Não é à toa que muitas crianças têm dificuldade de memorizar.

No sistema de numeração binário, no entanto, toda essa dificuldade desaparece. Afinal, temos apenas dois símbolos, o 0 e o 1, de modo que temos apenas que memorizar quatro combinações de operandos por operação aritmética.

Aqui veremos apenas a adição e a subtração de números expressos em base binária. A multiplicação e a divisão serão vistas apenas na disciplina de Organização de Computadores.

2.6.1 Adição de Números Binários

Independente da base em que operamos – seja decimal, binária, dozenal, etc. –, a adição de dois números obedece às seguintes regras:

1. Arrumar operandos um abaixo do outro, começando pela direita, alinhando os bits de mesma posição relativa em uma mesma coluna.
2. Começando pelo dígito menos significativo, somar individualmente os dígitos de uma mesma posição:
 - a) Se o resultado tiver apenas um dígito, continuar somando do dígitos da próxima posição à esquerda.
 - b) Se o resultado tiver dois dígitos, teremos um “vai-um”, ou *transporte*, que deve ser somado aos dígitos da próxima posição à esquerda.

Dessa forma, para somar dois números expressos na base binária, temos de somar individualmente cada par de bits que ocupam a mesma posição relativa nos dois operandos. Para tanto, usamos a seguinte tabuada:

$0+0$	$=$	0
$0+1$	$=$	1
$1+0$	$=$	1
$1+1$	$=$	10

(0, vai 1)

Quando três bits são somados (dois bits do operando e um bit do “vai-um”), a tabuada fica da seguinte maneira:

$1 + 0 + 0$	$=$	01	(1, vai 0)
$1 + 0 + 1$	$=$	10	(0, vai 1)
$1 + 1 + 0$	$=$	10	(0, vai 1)
$1 + 1 + 1$	$=$	11	(1, vai 1)

Com base na tabuada de adição binária, podemos ver agora alguns exemplos de adição de números binários.

Exemplo 2.17. Qual o resultado de $(11100)_2 + (10011)_2$?

$$\begin{array}{rcccccccl} 1 & & & & & & & & \\ & 1 & 1 & 1 & 0 & 0 & \Rightarrow & & 28_{10} \\ + & 1 & 0 & 0 & 1 & 1 & \Rightarrow & + & 19_{10} \\ \hline 1 & 0 & 1 & 1 & 1 & 1 & \Rightarrow & & 47_{10} \end{array}$$

1

Exemplo 2.18. Qual o resultado de $(111)_2 + (11)_2$?

$$\begin{array}{r} \Rightarrow _{10} \\ \Rightarrow _{10} \\ + \Rightarrow + _{10} \\ \hline 1 \Rightarrow _{10} \end{array}$$

☐

2.6.2 Subtração de Números Binários

Em uma subtração $X - Y$, o operando X é conhecido como *minuendo*, e operando Y , como *subtraendo*. Para subtrair dois números expressos em base binária, procedemos de forma semelhante ao que fazemos na base decimal:

1. Arrumar o subtraendo abaixo do minuendo, começando pela direita, alinhando os bits x_i e y_i de mesma posição relativa i em uma mesma coluna.
2. Começando pelo dígito menos significativo, subtrair individualmente os dígitos de uma mesma posição:
 - a) Se $x_i \geq y_i$, escrever resultado na mesma coluna, abaixo do subtraendo e continuar subtraindo os dígitos da próxima posição à esquerda.
 - b) Se $x_i < y_i$, tomar emprestado o valor da base ($B = (2)_{10} = (10)_2$), somando-o ao dígito do minuendo (x_i); realizar a subtração $((x_i + B) - y_i)$; escrever resultado na mesma coluna, abaixo do subtraendo; e devolver o empréstimo, somando o valor da base com o dígito y_{i+1} do subtraendo da próxima posição à esquerda.

Portanto, para subtrair dois números expressos na base binária, temos de subtrair individualmente cada par de bits que ocupam a mesma posição relativa nos dois operando. Para tanto, usamos a seguinte tabuada:

0	−	0	=	0
1	−	0	=	1
1	−	1	=	0
10	−	1	=	1 (empresta 1)

Vejamos um exemplo.

Exemplo 2.19. Qual o resultado de $(1001001)_2 - (11111)_2$?

$$\begin{array}{r}
 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \Rightarrow 73_{10} \\
 - \qquad \qquad 1 \ 1 \ 1 \ 1 \ 1 \Rightarrow - 31_{10} \\
 \hline
 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \Rightarrow 42_{10}
 \end{array}$$

□

Veja que a operação de subtração é um pouco mais trabalhosa que a de adição. Como veremos no Capítulo 3 a seguir, podemos expressar a operação $X - Y$ como sendo a soma de X com $-Y$, ou seja, o complemento de Y . Da mesma forma que isso é mais fácil para humanos, a redução de operações complexas em outras mais básicas também simplifica o projeto de hardware.

2.7 REPRESENTAÇÃO DE NÚMEROS EM ESPAÇOS RESTRITOS

Até aqui, nossas considerações de conversão de base e operações aritméticas cobriam todo o conjunto dos números naturais (\mathbb{N}) mais o

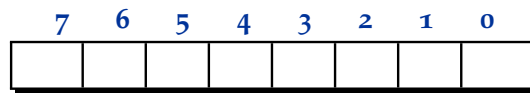
zero. No entanto, em sistemas computacionais, o espaço disponível para armazenamento de informação em memória é limitado. Ou seja, por maior que seja a memória, ela somente será capaz de representar uma ínfima quantidade de elementos do conjunto \mathbb{N} , que é infinito.

Por conta disso, o tamanho do espaço de memória disponível determinará a capacidade de representação de números. Por exemplo, com 10 bits de espaço, temos a possibilidade de representar $2^{10} = 1024$ valores distintos. Isso pode ser um espaço muito limitado em diversas aplicações.

Por outro lado, com 100 bits de espaço, temos a possibilidade de representar 2^{100} valores distintos. Pela eq. 4, essa quantidade de combinações equivale a aproximadamente 10^{30} . Este é um valor muito grande para as aplicações mais comuns do cotidiano. Como meio de comparação, estima-se que existam 6×10^{27} gotas de água na Terra, e que se passaram aproximadamente $4,7 \times 10^{17}$ segundos desde o surgimento do Universo.

Um byte é um grupo de oito bits.

Os computadores representam e processam informações agrupadas em um conjunto básico de bits, conhecido como *palavra*. Por muito tempo, os computadores usaram palavras de 32 bits. Atualmente, os sistemas computacionais mais modernos trabalham com palavras de 64 bits. Porém, para facilitar a compreensão, trabalhemos aqui com uma palavra de 8 bits:



Cada posição pode armazenar ou o símbolo 0 ou o símbolo 1. Logo, o número de combinações de símbolos 0 e 1 possíveis em uma sequência de 8 bits é dado por:

$$\underbrace{2 \cdot 2 \cdot 2 \cdots 2}_{8 \times} = 2^8$$

Se utilizarmos a sequência de bits 00000000 para representar o valor zero, 00000001 para representar o valor um, e assim por diante, teremos que a sequência 11111111 representará $255 (= 2^8 - 1)$. Como temos 255 valores de 1 a 255, o 256º valor representado é o zero.

Generalizando o raciocínio, em um espaço de n bits de memória, temos 2^n combinações únicas de bits. Se usarmos cada combinação para representar números inteiros não negativos, então os valores representados irão de 0 a $2^n - 1$.

overflow. E se, trabalhando com palavras de 8 bits, quisermos realizar a soma $255 + 1$? O que aconteceria?

$$\begin{array}{r} 11111111 \\ + \quad 00000001 \\ \hline 1 \quad 00000000 \end{array}$$

Como o espaço de armazenamento em memória é restrito a apenas 8 bits, o nono bit à esquerda é desprezado. Essa situação é conhecida como *transbordamento*, no sentido em que não há espaço para guardar o bit de transporte (vai-um) de um resultado.

Portanto, em um computador com palavras de 8 bits, o valor 256 é interpretado como 0, pois ele requer 9 bits para ser representado: 1 0000 0000. Porém, como a máquina dispõe de apenas 8 bits, o MSB (nona posição) é descartado, restando apenas 0000 0000.

No caso específico de números binários sem sinal, o transbordamento nos *indica* que houve *overflow*. É como se a contagem dos números acontecesse em círculos: você pode ter andado muitas casas, mas ao cruzar o ponto de partida (posição 0), a contagem recomeça e você perde tudo o que já andou.

A memória do computador reservada para armazenar um número natural se assemelha a um jogo de tabuleiro circular, como mostrado na Figura 6: há um número limitado de casas que podemos percorrer. Quando você anda mais casas do que o tabuleiro dispõe, o que vale não é o número de casas que você percorreu, mas sim a posição em que você está. É essa discrepância que se chama *overflow*.

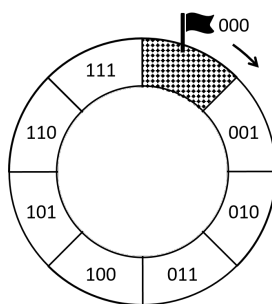


Figura 6: Analogia de um espaço restrito de numeração a um tabuleiro circular.

Na Seção 3.3.3, veremos outras formas de detectar *overflow*. Por ora, tenha em mente que *transbordamento* e *overflow* não são sinônimos.

Definição 5 (Overflow)

Overflow é uma condição na qual um espaço de memória não é suficientemente grande para representar um valor, em uma determinada notação numérica.

A condição de *overflow* é apenas um fenômeno. Ela não tem nenhuma conotação de certo ou de errado. Errado de verdade é o programador que esquece de verificar *overflow* em seu código, deixando-o propenso a ataques de segurança.

EXEMPLOS DE *overflow*. Por definição, o que determina se uma operação resulta ou não em *overflow* é a quantidade de bits disponíveis para representação. Veja os exemplos a seguir.

Exemplo 2.20. Qual o resultado de $28 + 19$, em um espaço de 6 bits?

$$\begin{array}{rcccccc}
 & 1 & & & & & \\
 & 0 & 1 & 1 & 1 & 0 & 0 \Rightarrow 28_{10} \\
 + & 0 & 1 & 0 & 0 & 1 & 1 \Rightarrow + 19_{10} \\
 \hline
 & 1 & 0 & 1 & 1 & 1 & 1 \Rightarrow 47_{10}
 \end{array}$$

Um espaço de 6 bits nos permite representar números naturais que variam de 0 a $63 (= 2^6 - 1)$. Como $28 + 19 = 47$, então é possível representar esse resultado em um espaço de 6 bits. Isso é comprovado pela soma em binário realizada acima. Nela, não houve transbordamento, o que é um indicativo de que não ocorreu *overflow*. □

Exemplo 2.21. Qual o resultado de $28 + 19$, em um espaço de 5 bits?

$$\begin{array}{rcccccc}
 & 1 & & & & & \\
 & 1 & 1 & 1 & 0 & 0 \Rightarrow 28_{10} \\
 + & 1 & 0 & 0 & 1 & 1 \Rightarrow + 19_{10} \\
 \hline
 \cancel{1} & 0 & 1 & 1 & 1 & 1 \Rightarrow 15_{10}
 \end{array}$$

Um espaço de 5 bits nos permite representar números naturais que variam de 0 a $31 (= 2^5 - 1)$. Como $28 + 19 = 47$, então não é possível representar esse resultado em um espaço de 5 bits. Isso é comprovado pela soma em binário realizada acima. Nela, houve transbordamento, o que é um indicativo de que ocorreu *overflow*.

O sexto bit 1 do resultado não pode ser armazenado em um espaço de apenas 5 bits. Por isso, diz-se que ele transborda, ou seja, ele se perde, pois não há local em memória para ele ser armazenado. Ele só existe na nossa mente humana. Com isso, o resultado de 5 bits fica sendo 01111_2 , o que equivale a 15_{10} . □

Retornando à analogia do tabuleiro circular de 8 casas, considere que estamos na casa 6 (110_2) e desejamos andar uma casa à frente. Ou seja, desejamos somar $110 + 001$, como mostra a Figura 7a. Como o resultado é 7, não ultrapassamos a cada de partida, e nem ocorre *overflow*.

Por outro lado, se quisermos avançar três casas a partir da casa 6, ultrapassaremos a linha de partida, e chegaremos à casa 1 novamente, como mostra a Figura 8. Porém, o esperado é que $6 + 3$ resultasse em 9, e não em 1. Como o tabuleiro tem apenas oito casas, não é possível registrar a casa de número 9, ocasionando *overflow*.

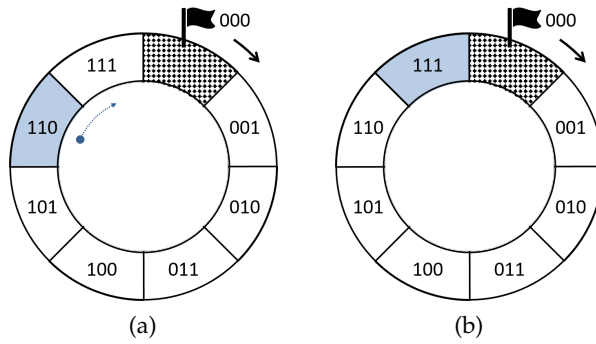


Figura 7: Analogia da soma $(110 + 001)_2$ em um tabuleiro circular.

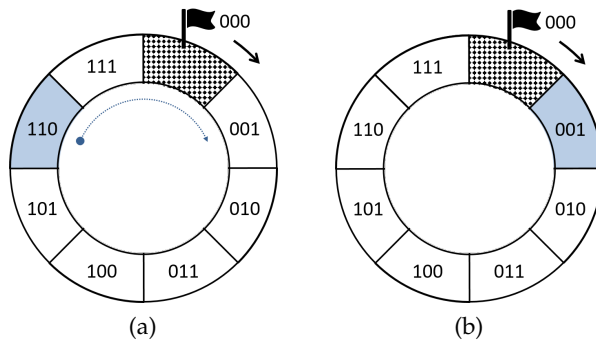


Figura 8: Analogia da soma $(110 + 011)_2$ em um tabuleiro circular.

Mas nem sempre a soma $6 + 3$ resulta em *overflow*. Se ampliarmos o tabuleiro para 16 casas, poderemos alcançar a cada de número 9 (1001_2), como mostra a Figura 9.

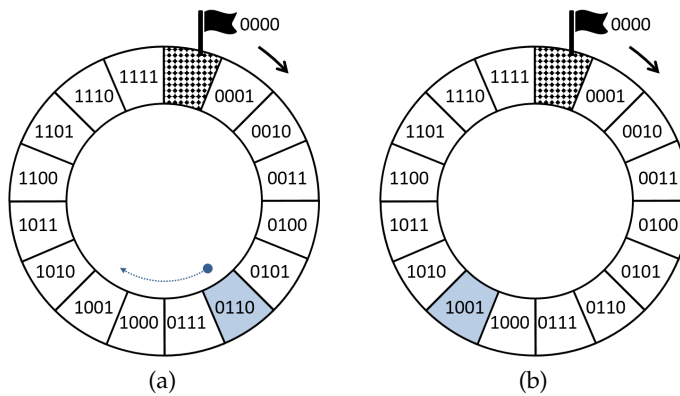


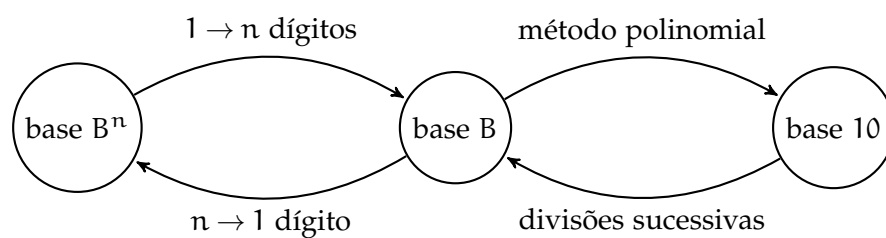
Figura 9: Analogia da soma $(0110 + 0011)_2$ em um tabuleiro circular maior, agora com 16 casas.

2.8 RESUMO DO CAPÍTULO

Palavras-chave:

- Número
- Numeral
- Sistema de numeração
- Sistema de numeração posicional
- Base numérica
- Dígito mais significativo
- Conversão de base
- Método das divisões sucessivas
- Método polinomial
- Bit
- Byte
- Dígito de transporte
- Transbordamento
- *Overflow*

Conversão de Bases Numéricas:



EXERCÍCIOS DO CAPÍTULO 2

2.1 Usando apenas lápis e papel (e talvez borracha), converta os números abaixo para as bases indicadas. Depois, confira o resultado com uma calculadora de programador.

BINÁRIO PARA DECIMAL

- a) $(101101)_2 = (\text{_____})_{10}$
 b) $(1010111)_2 = (\text{_____})_{10}$
 c) $(11111111)_2 = (\text{_____})_{10}$
 d) $(111000111)_2 = (\text{_____})_{10}$
 e) $(1000110001)_2 = (\text{_____})_{10}$

DECIMAL PARA BINÁRIO

- a) $(27)_{10} = (\text{_____})_2$
 b) $(31)_{10} = (\text{_____})_2$
 c) $(32)_{10} = (\text{_____})_2$
 d) $(33)_{10} = (\text{_____})_2$
 e) $(198)_{10} = (\text{_____})_2$

BINÁRIO PARA HEXADECIMAL

- a) $(101101)_2 = (\text{_____})_{16}$
 b) $(1010111)_2 = (\text{_____})_{16}$
 c) $(11111111)_2 = (\text{_____})_{16}$
 d) $(111000111)_2 = (\text{_____})_{16}$
 e) $(1000110001)_2 = (\text{_____})_{16}$

HEXADECIMAL PARA BINÁRIO

- a) $(AB9)_{16} = (\text{_____})_2$
 b) $(1F3)_{16} = (\text{_____})_2$
 c) $(401)_{16} = (\text{_____})_2$
 d) $(B0CA)_{16} = (\text{_____})_2$
 e) $(CED0)_{16} = (\text{_____})_2$

DECIMAL PARA HEXADECIMAL

- a) $(27)_{10} = (\text{_____})_{16}$
 b) $(31)_{10} = (\text{_____})_{16}$
 c) $(32)_{10} = (\text{_____})_{16}$
 d) $(33)_{10} = (\text{_____})_{16}$
 e) $(198)_{10} = (\text{_____})_{16}$

HEXADECIMAL PARA DECIMAL

- a) $(AB9)_{16} = (\text{_____})_{10}$
 b) $(1F3)_{16} = (\text{_____})_{10}$
 c) $(401)_{16} = (\text{_____})_{10}$
 d) $(B0CA)_{16} = (\text{_____})_{10}$
 e) $(CED0)_{16} = (\text{_____})_{10}$

2.2 Preencha os quadros em branco, realizando as conversões de base pedidas, de modo que cada coluna possua o mesmo valor numérico.

Binário			101110111
Hexadecimal		A8	
Decimal	197		

2.3 Realize as operações de adição e subtração indicadas abaixo envolvendo operandos expressos na base binária. Mostre também a operação equivalente na base decimal.

- a) $111 + 11$
 b) $11100 + 10011$
 c) $1101 + 1110$
 d) $1111 + 1100$
 e) $11 - 10$

- f) $11 - 101$
- g) $11011 - 1011$
- h) $10101 - 11110$

2.4 Considere um sistema de numeração de base $B > 1$. Responda:

- a) Quantos símbolos são disponíveis para representar números nessa base?
- b) Qual o menor número representável com apenas um símbolo?
- c) Qual o maior número representável com apenas um símbolo?
- d) Qual o valor – em função de B – do numeral 10 ?
- e) Qual o maior valor representável com dois símbolos?
- f) Quantos algarismos (aproximadamente) possui $(x)_B$ quando convertido na base decimal?

2.5 [29, ex. 1.18] Uma calculadora simples pode mostrar números decimais com, no máximo, seis dígitos. Qual é o tamanho da palavra binária necessária para representar tais números?

2.6 Qual o maior número decimal representável em uma máquina de N bits?

2.7 [28, ex. 2.1] Se circuitos digitais em computadores apenas funcionam com números binários, por que números hexadecimais são muito utilizados por profissionais da informática?

2.8 [11] Arrume, em ordem crescente, cada uma das seguintes triplas de números:

- a) $\{(2B)_{16}, (31)_{10}, (15)_8\}$
- b) $\{(44)_{10}, (3C)_{16}, (55)_8\}$

2.9 [15, ex. 1.19] A extinta civilização Maia utilizava um sistema de numeração com base 20. Qual o maior número que pode ser expresso com quatro dígitos nesse sistema?

2.10 Uma caixa alienígena com o símbolo “317” gravado na tampa foi descoberta por um grupo de cientistas. Ao abrirem a caixa, encontraram 207 objetos.

- a) Considerando que o alienígena tem um formato humanoide, quantos dedos ele tem nas duas mãos?
- b) Como o número $(ABC)_{16}$ seria representado no sistema de numeração alienígena?

- 2.11 [3, ex. 2.7] Em um exame, um aluno escreveu $(2756)_6$ como resposta a uma pergunta. Uma vez que 7 e 6 são maiores que 5, o maior dígito permitido em um sistema de base 6, a resposta deve estar errada. O que você acha que é o equivalente mais provável deste número em decimal, e por quê?
- 2.12 A primeira expedição a Marte provou a existência de civilizações inteligentes no planeta vermelho porque descobriu, gravada numa rocha, a equação $5x^2 - 50x + 125 = 0$, bem como as respectivas soluções, $x_1 = 5$ e $x_2 = 8$. O valor $x_1 = 5$ pareceu razoável aos cientistas da expedição, mas a outra solução indicava claramente que os marcianos não utilizavam, como nós, o sistema decimal de contagem (talvez porque não possuísem 10 dedos nas mãos). Quantos dedos os marcianos tinham em cada mão? Justifique.
- 2.13 [10, ex. 1.66] Um disco voador caiu em um igapó em Iranduba. Alunos de Ciência da Computação da UFAM investigaram os destroços e encontraram um manual contendo uma equação no sistema numérico alienígena: $325 + 42 = 411$. Se essa equação estiver correta, quantos dedos esses alienígenas devem ter?
- 2.14 [19, p.82] Ben e Alissa estão no meio de uma discussão. Ben diz “Para converter um número binário em decimal, eu começo pelo bit mais à esquerda em direção ao mais à direita. Pego o dobro do primeiro bit e somo com o próximo bit. Pego o dobro do resultado e somo ao bit seguinte. E assim vou repetindo até chegar ao bit menos significativo”. Alissa discorda: “Não, isso é leseira da sua cabeça. Isso só funciona na base 16, mas em vez de dobrar, você multiplica por 16”. Você concorda com Ben, com Alissa, com ambos ou com nenhum dos dois? Argumente.
- 2.15 [24, ex. 1.24]
- Mostre que um número de base b pode ser convertido para a base b^3 particionando os dígitos do número na base b em grupos de três dígitos consecutivos a partir do dígito menos significativo em direção à esquerda, e convertendo-se cada grupo em um dígito da base b^3 . (Dica: Represente o número da base b usando uma expansão de série de potências).
 - Mostre que um número na base b^3 pode ser convertido para a base b expandindo cada dígito do número de base b^3 em três dígitos consecutivos a partir do dígito menos significativo em direção à esquerda.
- 2.16 [24, ex. 1.25]
- Mostre como para representar cada um dos números $(5 - 1)$, $(5^2 - 1)$ e $(5^3 - 1)$ como um número de base 5.

- b) Generalize sua resposta na parte (a) e mostre como representar $(b^n - 1)$ como um número de base b , onde b pode ser qualquer número inteiro maior que 1, e n qualquer inteiro maior que 0. Dê uma derivação matemática de seu resultado.

2.17 [24, ex. 1.26]

- a) Mostre que o número 121_b , onde b é qualquer base maior que 2, é um quadrado perfeito (ou seja, é igual ao quadrado de um número inteiro).
- b) Repita a parte (a) para o número 12321_b , onde $b > 3$.
- c) Repita a parte (a) para o número 14641_b , onde $b > 6$.
- d) Repita a parte (a) para o número 1234321_b , onde $b > 4$.

2.18 [22][ex. 2.9] Em que base numérica B as seguintes operações são válidas?

- a) $9 + 8 = 11_B$
- b) $7 + 7 = 16_B$

REPRESENTAÇÃO DE NÚMEROS INTEIROS

OS NÚMEROS INTEIROS surgiram com a necessidade de subtrair uma quantidade maior de outra menor. Por exemplo, se tenho R\$ 100 em conta, qual será o meu saldo após descontar um cheque de R\$ 150? Para muitos de nós, parece intuitivo que a solução para esse problema seja simplesmente trocar os operandos na subtração e acrescentar o sinal de negativo (–) ao lado do resultado. Porém, que outros formatos poderíamos convencionar para representar esse resultado? Os bancos, por exemplo, costumam pintar valores negativos em vermelho.

E o que acontece em um mundo onde existem apenas dois símbolos, o 0 e o 1? Nesse mundo, o sinal negativo (–) é totalmente estranho, pois ele nem é 0, e nem é 1. Portanto, ele não pode ser usado para marcar um valor negativo. Consequentemente, temos de desenvolver alguma convenção para indicar quando os zeros e uns estão organizados para representar valores negativos, ou quando estão organizados para representar valores positivos.

Na verdade, vários sistemas (ou convenções) já foram desenvolvidos ao longo da história para representar números negativos usando apenas zeros e uns. Neste capítulo, veremos alguns deles, com destaque para a notação complemento de 2, o padrão atualmente usado pelos sistemas computacionais.

3.1 NOTAÇÃO SINAL E MAGNITUDE

A notação sinal e magnitude representa números positivos e negativos por meio de uma abordagem bem simples. Dada uma palavra de n bits, o bit mais significativo indica o *sinal* do número: 0, se positivo; 1, se negativo. Os demais $n - 1$ bits indicam a *magnitude* do número – ou seja, o seu módulo –, representado na base binária simples.

No exemplo abaixo, temos as representações de +18 e –18 na notação sinal e magnitude. Note que os $n - 1$ bits menos significativos das duas sequências são iguais. A diferença entre elas está apenas no bit mais significativo.

+18:	<table><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	0	0	0	1	0	0	1	0
0	0	0	1	0	0	1	0		
−18:	<table><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	1	0	0	1	0	0	1	0
1	0	0	1	0	0	1	0		

Uma palavra (word) é um conjunto ordenado de bits correspondente à unidade fundamental de informação que pode ser processada por um computador.

Note que a notação sinal e magnitude permite duas representações para o número zero:

+0:	<table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0		
−0:	<table><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0		

Duas representações do zero podem ser úteis quando trabalhamos com limites, pois podem indicar qual o sentido da convergência de uma função. Contudo, isso é válido se estivermos representando números reais. Como a notação sinal e magnitude representa números inteiros, duas representações do zero pode gerar ambiguidade.

3.1.1 Conversão da Base 10 para a Notação Sinal e Magnitude.

Para converter um número representado na base 10 para a notação sinal e magnitude, com n bits de tamanho, procedemos da seguinte forma (Figura 10):

1. Tomamos o valor absoluto (módulo) do número em base decimal e o convertemos para a base binária (método das divisões sucessivas, Seção 2.3).
2. Ao número binário obtido, acrescentamos zeros à esquerda, até a posição $n - 2$.
3. O bit mais significativo (posição $n - 1$) é determinado pelo sinal do número. Se for negativo, $a_{n-1} = 1$. Se for positivo, $a_{n-1} = 0$.

Exemplo 3.1. Qual o valor de $(+23)_{10}$ na notação sinal e magnitude de 8 bits de tamanho?

1. Primeiro convertemos $(+23)_{10}$ para a base binária:

23	2						
1	11	2					
	1	5	2				
		1	2	2			
			0	1	2		
				1	0		

Logo, $(23)_{10} = (10111)_2$.

2. Em seguida, acrescentamos zeros à esquerda, até o segundo bit mais significativo:

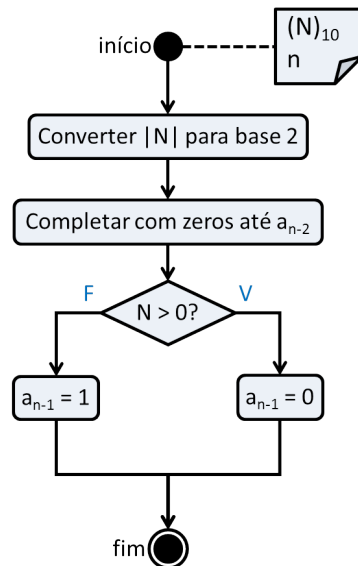


Figura 10: Algoritmo de conversão de um número inteiro expresso na base decimal para a notação sinal e magnitude.

7	6	5	4	3	2	1	0
	0	0	1	0	1	1	1

3. Por fim, como $+23$ é um número inteiro positivo, completamos o bit mais significativo com o valor 0:

7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	1

□

Exemplo 3.2. Qual o valor de $(-23)_{10}$ na notação sinal e magnitude de 8 bits de tamanho?

Do exemplo anterior, sabemos que $(23)_{10} = (10111)_2$. Como $+23$ é um número inteiro negativo, completamos o bit mais significativo com o valor 1:

7	6	5	4	3	2	1	0
1	0	0	1	0	1	1	1

□

3.1.2 Conversão da Notação Sinal e Magnitude para a Base 10.

Considere a seguinte sequência de n bits de um número em notação sinal e magnitude:

a_{n-1}	a_{n-2}	\dots	a_1	a_0
-----------	-----------	---------	-------	-------

Para encontrar seu valor A na base dez, procedemos de modo semelhante à conversão de número binários para decimal, exceto pelo bit mais significativo. Ou seja, cada bit da sequência deve ser multiplicado pela potência de dois correspondente à sua posição, exceto o bit mais significativo, que apenas indica o sinal do número:

$$A = (-1)^{a_{n-1}} \cdot \sum_{i=0}^{n-2} a_i \cdot 2^i \quad (5)$$

Exemplo 3.3. Qual o valor em decimal de 10101110_{SM} ?

7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	0

$$\begin{aligned} A &= (-1)^1 \cdot (\cancel{0 \cdot 2^6} + 1 \cdot 2^5 + \cancel{0 \cdot 2^4} + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + \cancel{0 \cdot 2^0}) \\ &= -(2^5 + 2^3 + 2^2 + 2^1) \\ &= -(32 + 8 + 4 + 2) \\ &= -46 \end{aligned}$$

□

Exemplo 3.4. Qual o valor em decimal de 01010001_{SM} ?

7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	1

$$\begin{aligned} A &= (-1)^0 \cdot (1 \cdot 2^6 + \cancel{0 \cdot 2^5} + 1 \cdot 2^4 + \cancel{0 \cdot 2^3} + \cancel{0 \cdot 2^2} + \cancel{0 \cdot 2^1} + 1 \cdot 2^0) \\ &= +(2^6 + 2^4 + 2^0) \\ &= +(64 + 16 + 1) \\ &= +81 \end{aligned}$$

□

3.1.3 Limites de Representação da Notação Sinal e Magnitude

Vimos no Capítulo 2 que, no sistema binário sem sinal, uma palavra de n bits é capaz de representar 2^n números. Na notação sinal e magnitude, metade ($2^n/2 = 2^{n-1}$) dessas possibilidades são usadas para representar números positivos e o zero com sinal positivo; e a outra

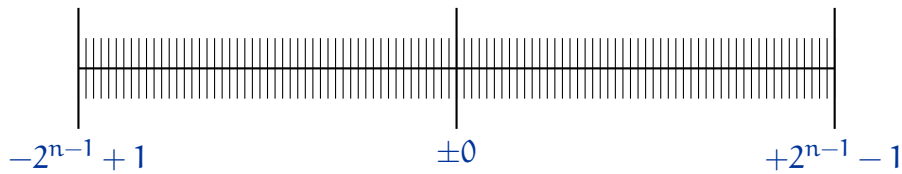


Figura 11: Faixa de valores inteiros possíveis de serem representados por uma palavra de n bits na notação sinal e magnitude.

metade, os números negativos e o zero com sinal negativo. Portanto, a notação sinal e magnitude é capaz de representar números de 0 a $2^{n-1} - 1$, no lado positivo da reta inteira, e de $-(2^{n-1} - 1)$ a 0, no lado negativo.

A Figura 11 ilustra a distribuição dos inteiros representáveis em notação sinal e magnitude para uma palavra de n bits. Assim, com uma palavra de um byte de tamanho, podemos representar os números inteiros compreendidos no intervalo de $-2^{8-1} + 1 = -127$ a $+2^{8-1} - 1 = +127$.

3.1.4 Desvantagens da Notação Sinal e Magnitude

Além da ambiguidade gerada pelas duas representações do número zero, a notação sinal e magnitude apresenta outra desvantagem. Como veremos mais adiante, operações aritméticas são implementadas usando-se uma combinação de portas lógicas. Quanto mais operações são oferecidas por um circuito, mais complexo é o seu projeto. No entanto, se pudermos fazer modificações simples para transformar uma operação aritmética em um caso especial de uma outra, teremos um produto mais completo com projeto de menor complexidade (e custo).

Ora, lembre-se que as operações aritméticas podem ser vistas como manipulações de símbolos. Se esses símbolos seguirem uma notação apropriada, poderemos ter manipulações menos complexas. É o que veremos nas próximas seções.

3.2 COMPLEMENTO DE 2

Os sistemas computacionais modernos utilizam a notação complemento de 2 como padrão para representar números inteiros em memória. Trata-se de um fruto de um longo trabalho de engenhosidade, traduzido hoje em regras simples que escondem uma bela maneira de se interpretar os números.

Antes de apresentar as regras nuas e frias, vamos mostrar os conceitos básicos de onde elas surgiram.

3.2.1 Complemento de Horas

Suponha que sejam 7 horas da noite. Uma amiga lhe diz que tentou falar com você 9 horas atrás. A partir dessas informações você logo deduz que essa tentativa deve ter ocorrido por volta das 10 horas da manhã.

Porém, pensando explicitamente, como você usou os números 7 e 9 para chegar a 10? Subtraindo 9 de 7, temos -2 , e não 10.

Na verdade, o que vários de nós fazemos é subtrair 9 de 12, que é o total de marcações de horas no relógio, obtendo 3, e somamos esse resultado à hora inicial (7), obtendo 10 como resultado. Tudo isso é feito de forma intuitiva, sem pensarmos muito a respeito.

Ao subtrair 9 horas de 7 horas, primeiro fizemos uma complementação do subtraendo (9) do limite de horas (12). Em seguida, a operação de subtração inicial ($7 - 9$) foi transformada em uma adição ($7 + 3$). Ou seja, no contexto de horas, nós normalmente transformamos a operação de subtração em duas outras mais simples: complementação e adição.

Pela Figura 12, podemos perceber, em termos visuais, que a complementação no relógio consiste em “espelhar”, em relação ao eixo vertical, a hora desejada. Em termos aritméticos, o oposto de uma hora h corresponde a $(12 - h)$.

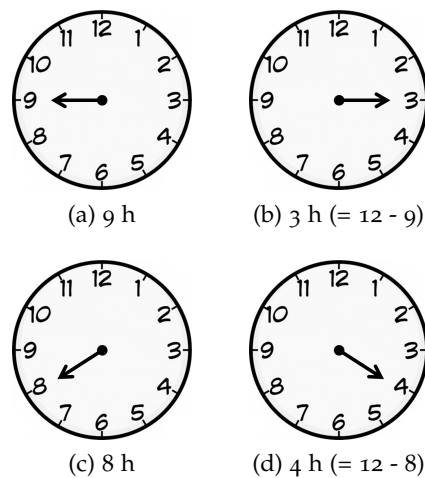


Figura 12: Horas complementares no relógio são simétricas em relação ao eixo vertical.

3.2.2 Complemento de Dez

Na aritmética de relógio, vimos que podemos transformar a subtração em duas operações mais simples: a complementação e adição. No entanto, como usamos a base 10 na maioria das aplicações do dia-a-

dia, e não a 12, é complicado estender o raciocínio para números maiores. Por isso, na explicação a seguir, vamos utilizar a base 10, à qual estamos acostumados desde pequenos.

Tomemos como exemplo o hodômetro, que registra a distância percorrida por um veículo. O mostrador do hodômetro possui um número limitado de casas decimais. Consideremos três posições, para facilitar a explicação. Se o carro andar 1002 km, então o hodômetro marcará apenas 002, pois não tem como registrar a casa decimal mais à esquerda, que contém o milhar 1.

Na vida real, hodômetros apenas incrementam; nunca decrementam. No entanto, imaginemos que o hodômetro do nosso exemplo regreda a contagem quando o carro anda em marcha ré. Nesse caso, se um carro com 000 km rodado andar 3 km em marcha ré, nosso hodômetro fictício marcará 997.

Semelhante ao relógio, que possui 12 marcações de horas, o nosso hodômetro tem mil marcações de quilometragem: 0 a 999. Assim, para encontrar como serão registrados 3 km negativos, subtraímos esse valor de 1000, obtendo 997. Ou seja, 997 km é o mesmo que -3 km na aritmética do hodômetro, uma vez que este não tem capacidade de representar o sinal de menos. Em outras palavras, 997 é o “complemento a mil” de 3.

Generalizando o exemplo, se um carro anda x quilômetros para a frente, o hodômetro registrará x . Se o carro anda x quilômetros de ré, o hodômetro marcará $1000 - x$. Dessa forma, x e seu oposto $-x$ somarão sempre 1000. Generalizando mais ainda, o valor registrado para o oposto de x dependerá do número de posições do hodômetro. Para o hodômetro de 3 posições, tínhamos $10^3 - x$. Logo, para um hodômetro de n posições, x quilômetros em ré serão registrados como $10^n - x$.

Consequente, o valor do oposto de um número registrado no hodômetro dependerá do número de posições deste. Não podemos ser precipitados em afirmar que o oposto de 6 sempre será 994. Isso só vale para o hodômetro de três posições. Para o hodômetro de cinco posições, 99994 é o oposto de 6.



Aritmética do hodômetro

O valor do oposto de um número inteiro depende do número de posições do hodômetro. Para um hodômetro de n posições, então o oposto de x será representado por $10^n - x$.

Vejamos outra situação. Considerando que o hodômetro marque 150 km, qual o valor que ele passará a marcar se o carro andar 240 km em marcha ré?

Bem, primeiro encontramos o oposto de 240, que é $1000 - 240 = 760$. Em seguida, somamos esse resultado à marcação original do hodômetro, o que resulta em $760 + 150 = 910$ km. Portanto, um hodômetro que marca 150 km, após rodar 240 km em marcha a ré, passará a marcar 910 km.

Observe que, por um lado, transformamos subtrações ($0 - 3$ e $150 - 240$) em adições por meio da complementação a 1000. Por outro lado, a complementação a 1000 envolve uma subtração, que é justamente a operação que estamos tentando evitar. Ou seja, temos de pensar em outra técnica de complementação que dispense definitivamente a subtração.

A complementação em relação a 1000 pode ser simplificada da seguinte forma. Em vez de encontrarmos o complemento de um número a 1000, é mais fácil encontrar sua diferença em relação a 999, pois não haverá “empréstimos” de uns. Lembre-se que a subtração de dois operandos é realizada pela comparação dos algarismos do subtraendo com os do minuendo que se encontram na mesma posição. Se tivermos um minuendo composto apenas de algarismos 9, garantimos que nenhum algarismo do subtraendo será maior que o do minuendo, de modo que nunca precisaremos de empréstimos.

Para encerrar a operação de complemento a 1000, somamos 1 ao resultado do complemento a 999, de modo a devolver o 1 subtraído de 1000. Assim, chegamos ao oposto de um número na aritmética do hodômetro de três posições.

A principal consequência de usarmos o complemento a 9 de cada algarismo a_i de um número denotado pela sequência $a_{n-1} \dots a_1 a_0$ em vez do complemento a 10^n é que podemos generalizar esse procedimento para hodômetros de tamanhos maiores, não limitados a apenas três casas decimais. Vejamos alguns exemplos para melhor entendimento.

Exemplo 3.5. *Qual é o complemento a 1000 de 123, ou seja, qual seria a marcação do hodômetro se o carro andar 123 km em marcha ré, a partir do quilômetro zero?*

$$\begin{array}{r} 123 \\ \downarrow \text{ Compl. 9 } \\ 876 \\ \downarrow +1 \\ 877 \end{array}$$

Logo, o hodômetro marcaria 877 km. □

Exemplo 3.6. *A indicação 765 corresponde ao complemento a 1000 de que número no hodômetro?*

$$\begin{array}{r}
 765 \\
 \downarrow \text{ Compl. } 9 \\
 234 \\
 \downarrow +1 \\
 235
 \end{array}$$

Portanto, 765 corresponde a -235 em complemento a 1000. \square

Veja que nas duas analogias, a do relógio (base 12) e a do hodômetro (base 10), tentamos substituir uma operação de subtração por duas outras mais simples (em tese): complementação e adição. No exemplo do relógio, a complementação é uma operação simples porque trabalhamos com apenas doze valores. No exemplo do hodômetro, recorremos ao empréstimo de uma unidade para que a complementação fosse realizada algarismo por algarismo, e não pelo valor total do número.

Mesmo assim, em ambos os exemplos, a complementação envolve uma subtração, ainda que em relação a um mesmo número. Por outro lado, se utilizarmos um sistema de numeração com apenas dois símbolos, a complementação será ainda mais fácil de ser resolvida, pois o complemento de um dos símbolos será necessariamente o outro, o que elimina de uma vez por todas a operação de subtração na complementação. Isso é o que veremos a seguir.

3.2.3 A Notação Complemento de 2 (Finalmente)

Vamos agora estender o raciocínio adotado na analogia do hodômetro para entender como funciona a notação complemento de 2 e as conversões de base associadas.

3.2.3.1 Conversão de Decimal para Complemento de 2

No hodômetro de três posições, convencionamos que o oposto de um número era o seu complemento de $10^3 = 1000$. Da mesma forma, na base binária, podemos convencionar que o oposto de um número de n bits é o seu complemento para o valor 2^n .

Também vimos que, dado um número $a_2a_1a_0$, é mais fácil subtraí-lo de $999 = 10^3 - 1$ do que de 1000. Isso porque deixamos de fazer empréstimos de 1s durante a subtração e passamos apenas a complementar cada algarismo a_i em relação a 9. Por fim, somamos 1 ao resultado para compensar o empréstimo de 1000 para 999.

Da mesma forma, a conversão de um número negativo expresso em base 10 para a notação complemento de 2 acontece da seguinte forma (Figura 13):

1. Tomamos o valor absoluto (módulo) do número em base decimal e o convertemos para a base binária.
2. Ao número binário obtido, acrescentamos zeros à esquerda, até chegarmos a uma sequência de n bits.
3. Para obter o complemento em relação a 2^n , subtraímos a sequência obtida no passo anterior de $2^n - 1$, que nada mais é do que uma sequência de n bits iguais a 1. Ou seja, vamos complementar cada bit da sequência com 1.
4. Por fim, somamos 1 ao resultado final, para compensar o empréstimo no passo anterior.

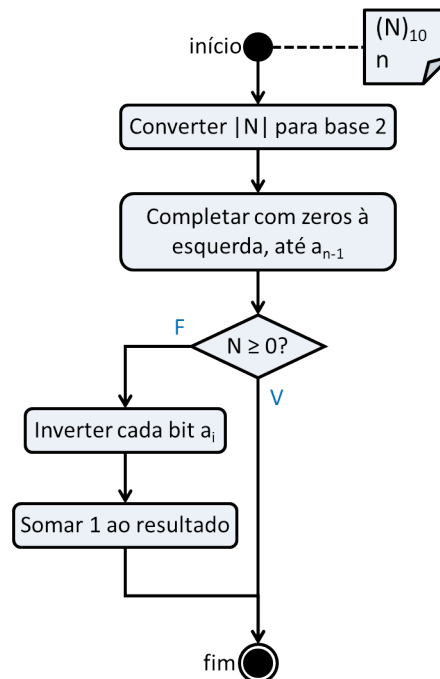


Figura 13: Algoritmo de conversão de um número inteiro expresso na base decimal para a notação complemento de 2.

Exemplo 3.7. Quanto vale -18 em complemento de 2, considerando uma sequência de $n = 8$ bits?

Em binário sem sinal, temos que $+18_{10} = 10010_2$. Logo:

$$\begin{array}{r}
 00010010 \\
 \downarrow \text{ Compl. 1} \\
 11101101 \\
 \downarrow +1 \\
 11101110
 \end{array}$$

Portanto, $-18 = 11101110_{C2}$.

□

3.2.3.2 Conversão de Complemento de 2 para Decimal

Considere a seguinte sequência de n bits, que representa um número em notação complemento de 2:

a_{n-1}	a_{n-2}	\dots	a_1	a_0
-----------	-----------	---------	-------	-------

Para encontrar o valor A representado por essa sequência na base dez, procedemos de modo semelhante à conversão de número binários para decimal, exceto pelo bit mais significativo. Ou seja, cada bit da sequência deve ser multiplicado pela potência de dois correspondente à sua posição, exceto pelo bit mais significativo, conforme a seguinte equação:

$$N = -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i \quad (6)$$

Exemplo 3.8. Qual o valor, na base dez, de 10101110_{C2} ?

7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	0

$$\begin{aligned} N &= -1 \cdot 2^7 + \cancel{0 \cdot 2^6} + 1 \cdot 2^5 + \cancel{0 \cdot 2^4} + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + \cancel{0 \cdot 2^0} \\ &= -2^7 + 2^5 + 2^3 + 2^2 + 2^1 \\ &= -128 + 32 + 8 + 4 + 2 \\ &= -82 \end{aligned}$$

□

Exemplo 3.9. Qual o valor em decimal de 01010001_{C2} ?

7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	1

$$\begin{aligned} N &= -\cancel{0 \cdot 2^7} + 1 \cdot 2^6 + \cancel{0 \cdot 2^5} + 1 \cdot 2^4 + \cancel{0 \cdot 2^3} + \cancel{0 \cdot 2^2} + \cancel{0 \cdot 2^1} + 1 \cdot 2^0 \\ &= +2^6 + 2^4 + 2^0 \\ &= +64 + 16 + 1 \\ &= +81 \end{aligned}$$

□

Tabela 2: Faixa de números inteiros representáveis na notação complemento de 2 em uma palavra de n bits.

Decimal	Complemento de dois
-2^{n-1}	1000 ... 0000
$-2^{n-1} + 1$	1000 ... 0001
\vdots	\vdots
-2	1111 ... 1110
-1	1111 ... 1111
0	0000 ... 0000
+1	0000 ... 0001
+2	0000 ... 0010
\vdots	\vdots
$+2^{n-1} - 2$	0111 ... 1110
$+2^{n-1} - 1$	0111 ... 1111

3.2.3.3 Limites de Representação da Notação Complemento de 2

Com uma palavra de n bits, podemos gerar 2^n padrões de bits. Na notação em complemento de 2, metade desses padrões são usados para representar os números negativos. A outra metade é usada para representar o zero e os números positivos. Portanto, temos uma assimetria em relação ao zero, pois representamos uma quantidade maior de números negativos do que de números positivos.

Conforme mostra a Tabela 2, o menor número inteiro negativo representável com n bits é -2^{n-1} , pois é a metade das combinações de 0s e 1s possíveis com n bits ($2^n/2 = 2^{n-1}$). A sequência de bits correspondente ao menor número inteiro negativo representável é composta pelo bit 1 seguido de $n - 1$ zeros.

Em complemento de 2, o valor -1 sempre corresponde a uma sequência de n bits iguais a 1.

Já o maior inteiro positivo representável com n bits é $2^{n-1} - 1$, pois um dos padrões de bits que começam com 0 foi usado para representar o valor zero. Daí, sobram apenas metade das combinações possíveis com n bits (2^{n-1}) menos a combinação do zero para representar os números positivos.

Outra forma de visualizar a faixa de números inteiros representáveis com n bits na notação complemento de 2 é pela Figura 14. Compare-a com a Figura 11 (p. 47), que ilustra a faixa de valores representáveis em notação sinal e magnitude.

3.2.3.4 Generalizando a Regra de Negação

Vimos que, para negar um número inteiro positivo em complemento de 2, devemos inverter seus bits e somar 1. Será se negarmos um

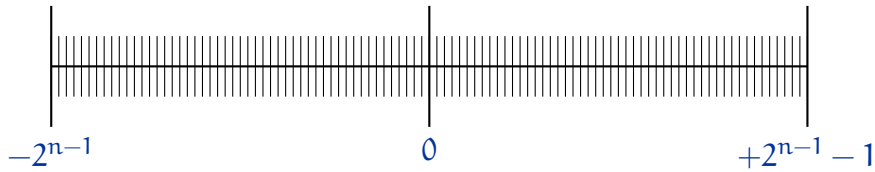


Figura 14: Faixa de valores inteiros possíveis de serem representados por uma palavra de n bits na notação complemento de 2.

número inteiro negativo vamos obter o número positivo de volta? Vamos tentar um exemplo.

Exemplo 3.10. Sabendo que 11101110_{C2} corresponde a -18 , vamos encontrar seu complemento de 2:

$$\begin{array}{r}
 11101110 \\
 \downarrow \text{ Compl. 1} \\
 00010001 \\
 \downarrow +1 \\
 00010010
 \end{array}$$

Conforme esperado, a negação da negação de $+18$ é ele próprio.

□

Mas ainda nossa pergunta não fica respondida definitivamente. Será se isso vale para qualquer número inteiro representado em complemento de 2?

Vamos demonstrar que sim. Para isso, considere a sequência de n bits $a_{n-1}a_{n-2}\cdots a_1a_0$, equivalente a um número inteiro N na notação complemento de 2. Logo, o valor de N na base dez é dado pela eq. 6:

$$N = -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i$$

Denotemos o complemento de 1 de cada bit a_i como $\overline{a_i}$. Logo, a negação de N em complemento de 2 será expressa por $\overline{N} + 1$:

$$-N = -\overline{a_{n-1}} \cdot 2^{n-1} + 1 + \sum_{i=0}^{n-2} \overline{a_i} \cdot 2^i$$

Ora, se a regra da negação da negação funcionar para qualquer caso de representação em complemento de 2, temos de provar que $N - \overline{N} = 0$:

$$\begin{aligned}
N - N &= -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i - \overline{a_{n-1}} \cdot 2^{n-1} + 1 + \sum_{i=0}^{n-2} \overline{a_i} \cdot 2^i \\
&= -(a_{n-1} + \overline{a_{n-1}}) \cdot 2^{n-1} + 1 + \sum_{i=0}^{n-2} (a_i + \overline{a_i}) \cdot 2^i \\
&= -2^{n-1} + 1 + \sum_{i=0}^{n-2} 2^i \\
&= -2^{n-1} + 1 + (2^{n-1} - 1) = \\
&= 0
\end{aligned}$$

Portanto, concluímos que a regra da negação de um número inteiro em notação complemento de 2 de n bits é válida para qualquer inteiro representável no intervalo $[-2^{n-1}; 2^{n-1} - 1]$. Existem, porém, dois casos especiais a considerarmos.

DOIS CASOS ESPECIAIS. Na notação complemento de 2, o oposto de zero é zero.

$$\begin{array}{c}
00000000 \\
\downarrow \text{ Compl. 1} \\
11111111 \\
\downarrow +1 \\
\underline{1} 00000000
\end{array}$$

Embora exista um transbordamento (vai um), ele é ignorado. Portanto, a negação do zero é o próprio zero, conforme esperado.

Já o segundo caso é um pouco mais estranho: o oposto de -2^{n-1} , o menor número negativo representável em complemento de 2 com n bits, é ele próprio.

$$\begin{array}{c}
10000000 \\
\downarrow \text{ Compl. 1} \\
01111111 \\
\downarrow +1 \\
10000000
\end{array}$$

Essa anomalia é inevitável. Como vimos na Seção 3.2.3.3, há uma desigualdade na quantidade de números negativos e positivos representáveis na notação complemento de 2. Consequentemente, existe uma representação para -2^{n-1} , mas não para 2^{n-1} . Por conta disso, ao negarmos -2^{n-1} , voltamos ao mesmo número.

3.2.3.5 Regras práticas

SINAL. Na notação complemento de 2, o bit mais significativo também indica o sinal. Se o bit mais significativo de um número for 0, então ele é positivo. Se for 1, então ele é negativo. Mas neste caso, o sinal não é apenas multiplicado ao número, como na notação sinal e magnitude (eq. 5). O bit mais significativo de uma palavra em complemento de 2 pode tornar o número negativo porque, no somatório da eq. 6, ele contribui com a maior parcela em módulo, mas subtraindo.

COMPARAÇÃO. Em certas situações, dadas duas sequências de bits em complemento de 2, desejamos saber qual a menor ou a maior. Uma abordagem seria convertê-las para a base 10 e comparar os resultados. Mas isso é trabalhoso e demorado. A partir da Tabela 2 observamos os seguintes critérios:

1. Se um número tem 1 no bit mais significativo, então ele é menor que um número que tem 0 nessa mesma posição. Ou seja, apesar de uma sequência iniciar com um algarismo maior, ela tem menor *valor*.
2. Nos bits menos significativos restantes, o bit 1 vale mais que o bit 0.

Exemplo 3.11. Considere as quatro sequências de bits S_1 , S_2 , S_3 e S_4 abaixo. Arrume-as em ordem crescente, supondo que estejam representadas em complemento de 2.

S_1 : 11111010

S_2 : 00111011

S_3 : 01000111

S_4 : 10100010

Em complemento de 2, as sequências que têm o bit 1 na posição mais significativa representam valores negativos. No caso de palavras de 8 bits, essa posição contribui com $-2^7 = -128$ no valor do número. Portanto, S_1 e S_4 são as menores sequências e S_2 e S_3 são as maiores.

Para determinar quem é maior entre cada par, analisamos a próxima posição, a segunda mais significativa. As sequências que tiverem mais bits 1 concentrados nas posições mais significativas (exceto a mais à esquerda), representarão números maiores, em decorrência do somatório da eq. 6. Logo, S_3 é maior que S_2 , pois possui 1 na posição 6 ao passo que S_2 possui bit 0 nessa mesma posição. Pelo mesmo motivo, S_1 é maior que S_4 .

Portanto, as sequências ficam assim arrumadas em ordem crescente: $S_3 > S_2 > S_1 > S_4$. \square

3.2.3.6 Desvantagens da Notação Complemento de 2

A partir da discussão anterior, percebemos que a notação complemento de 2 não facilita a comparação de sequências de bits. Existem

dois critérios de comparação: um para o bit mais significativo e outro para os demais bits da palavra. Dessa maneira, ou o circuito digital responsável pela comparação deverá ser projetado para fazer esse tratamento especial, ou o programador, sabendo que seu hardware não faz essa diferenciação de critérios, deverá corrigir isso via software.

Tal desvantagem não impede que a notação complemento de 2 seja amplamente utilizada em sistemas computacionais para representar números inteiros. No entanto, para representar o expoente de números reais, utilizamos a notação *com excesso*, que facilita a operação de comparação. É o que veremos na Seção 3.4.

3.3 OPERAÇÕES ARITMÉTICAS EM COMPLEMENTO DE 2

Todas as operações aritméticas básicas entre números representados em complemento de 2 podem ser reduzidas à adição. Aqui, veremos apenas a adição e a subtração. A multiplicação e divisão ficam para um curso mais avançado.

3.3.1 Adição em Complemento de 2

Para somar dois números em complemento de 2, procedemos da mesma maneira que vimos na Seção 2.6.1 para números binários sem sinal: alinham-se os operandos pela direita, e somam-se os bits de mesma coluna, verificando-se o transporte (vai-um).

Exemplo 3.12. Qual o resultado de $00101001 + 01001011$, em notação complemento de 2 com 8 bits?

$$\begin{array}{rcccccccc}
 & & & & 1 & & 1 & 1 & \\
 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & \Rightarrow 41_{10} \\
 + & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \Rightarrow + 75_{10} \\
 \hline
 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & \Rightarrow 116_{10}
 \end{array}$$

□

Caso o bit de transporte (vai-um) exceda o tamanho da palavra, ele é desconsiderado. Nesse caso, teremos uma situação de *transbordamento*, mas não de *overflow*, como veremos na Seção 3.3.3.

Exemplo 3.13. Qual o resultado de $11101010 + 11001011$, em notação complemento de 2 com 8 bits?

$$\begin{array}{rcccccccc}
 & 1 & 1 & & & 1 & & 1 & \\
 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & \Rightarrow -22_{10} \\
 + & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \Rightarrow + -53_{10} \\
 \hline
 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \Rightarrow -75_{10}
 \end{array}$$

□

Pelos Exemplos 3.12 e 3.13 acima, percebe-se que, durante a adição em complemento de 2, não precisamos nos preocupar com o sinal dos operandos. Para obter o resultado, basta somarmos os bits, independentemente do peso que eles representam.

Além disso, no Exemplo 3.13, houve um transbordamento do vai-um resultante da soma dos bits mais significativos dos operados. Apesar disso, não houve *overflow*, pois o resultado (-75_{10}) é representável no intervalo $[-128; +127]$ segundo a notação complemento de 2 com 8 bits. Lembre-se que o conceito de *overflow* não está necessariamente amarrado a transbordamento de bits, mas sim à capacidade de uma notação numérica e um espaço limitado de memória em armazenar um valor numérico. Mais detalhes na Seção 3.3.3.

3.3.2 Subtração em Complemento de 2

A notação complemento de 2 visa facilitar a realização de operações aritméticas, transformando a subtração em uma adição. Quanto menor o conjunto de procedimentos a serem executados por um sistema computacional, mais simples e mais barato ele será.

Para subtrair dois números em complemento de 2, devemos fazer o seguinte:

1. Manter o minuendo e inverter o sinal do subtraendo em complemento de dois.
2. Somar o minuendo ao subtraendo com sinal invertido.

Vejamos alguns exemplos:

Exemplo 3.14. Qual o resultado de $01001011 - 00000101$, em notação complemento de 2 com 8 bits?

RESPOSTA. Primeiro devemos inverter o sinal de 00000101 :

$$\begin{array}{r}
 00000101 \\
 \downarrow \text{ Compl. 1} \\
 11111010 \\
 \downarrow +1 \\
 11111011
 \end{array}$$

Agora somamos o resultado 11111011 ao minuendo 01001011 :

$$\begin{array}{r}
 \begin{array}{cccccccc}
 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
 + & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
 \hline
 \cancel{1} & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0
 \end{array}
 \Rightarrow
 \begin{array}{r}
 75_{10} \\
 + \\
 -5_{10} \\
 \hline
 70_{10}
 \end{array}
 \end{array}$$

□

Exemplo 3.15. Qual o resultado de $11001011 - 11111011$, em notação complemento de 2 com 8 bits?

RESPOSTA. Primeiro devemos inverter o sinal de 11111011 (-5). Do exemplo anterior, já sabemos que ele é o oposto de 00000101 ($+5$). Então basta somarmos 00000101 ao minuendo 11001011 :

$$\begin{array}{rcccccccc}
 & & & & 1 & 1 & 1 & 1 \\
 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & \Rightarrow & -53_{10} \\
 + & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \Rightarrow & +5_{10} \\
 \hline
 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & \Rightarrow & -48_{10}
 \end{array}$$

□

3.3.3 Overflow em Complemento de 2

Como já vimos anteriormente, o *overflow* ocorre quando um número tem mais bits do que o espaço de memória disponível para armazenamento. No caso específico de números binários sem sinal, uma maneira de detectar *overflow* é quando há transbordamento de bits. No entanto, para a notação complemento de 2, a forma de detectar *overflow* é outra.

Veja que o conceito de *overflow* é o mesmo para qualquer notação. O que muda são as técnicas que empregamos para detectá-lo. Em números binários não sinalizados, a técnica para se verificar *overflow* foi o transbordamento (Seção 2.7). Na notação complemento de 2, o transbordamento não indica necessariamente *overflow*, como vimos no Exemplo 3.13. Antes de apresentarmos as técnicas de detecção de *overflow* em complemento de 2, vejamos mais dois exemplos.

Exemplo 3.16. Qual o resultado de $01101001 + 01001011$, em notação complemento de 2 com 8 bits?

RESPOSTA. Como os dois números começam por zero, sabemos que ambos são positivos. Vamos alinhá-los à direita para somar somamos as parcelas bit a bit:

$$\begin{array}{rcccccccc}
 & & & & 1 & & & 1 & & 1 & 1 \\
 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & \Rightarrow & +105_{10} \\
 + & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & \Rightarrow & +75_{10} \\
 \hline
 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & \Rightarrow & -108_{10} \quad ?
 \end{array}$$

□

No Exemplo 3.16, o resultado esperado em base decimal seria:

$$105 + 75 = 180$$

Entretanto, o resultado obtido foi um número negativo (-108). Isso aconteceu porque em notação complemento de 2 com 8 bits só podemos representar números inteiros pertencentes ao intervalo de -2^7 a $+2^7 - 1$. Como $180 > 127$, ocorreu um *overflow*, pois a capacidade do espaço de memória foi excedida.

Por outro lado, no Exemplo 3.16, não houve transbordamento que nos indicasse *overflow*. Então, que outro indicador podemos então utilizar para detectar *overflow* em números expressos em complemento de 2? Vejamos mais um exemplo.

Exemplo 3.17. Qual o resultado de $10101001 - 01001011$, em notação complemento de 2 com 8 bits?

RESPOSTA. Como se trata de uma subtração de números expressos em notação complemento de 2, vamos inverter o sinal do subtraendo:

$$\begin{array}{r}
 01001011 \\
 \downarrow \text{Compl.1} \\
 10110100 \\
 \downarrow +1 \\
 10110101
 \end{array}$$

Agora somamos as parcelas, alinhando-as à direita:

$$\begin{array}{r}
 \begin{array}{cccccccc}
 & 1 & & 1 & & & 1 & \\
 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & \Rightarrow & -87_{10} \\
 + & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & \Rightarrow & + & -75_{10} \\
 \hline
 \cancel{1} & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & \Rightarrow & +86_{10} & ?
 \end{array}
 \end{array}$$

□

No Exemplo 3.17, o resultado esperado em base decimal seria:

$$-87 - 75 = -162$$

Entretanto, o resultado obtido foi um número positivo ($+86$). Como $-162 < -128$, ocorreu um *overflow*, pois mais uma vez a capacidade do espaço de memória foi excedida ao tentarmos representar um valor com mais de 8 bits de tamanho.

Você já deve ter notado que, em vez de transbordamento, o que indica a ocorrência de *overflow* na notação complemento de 2 é a *troca de sinal* do resultado em relação aos operandos:

- Se somarmos dois números inteiros de sinais opostos, o resultado com certeza ficará no intervalo representável, de modo que não haverá *overflow*.

- Se somarmos dois números inteiros de sinais iguais e o resultado mantiver o mesmo sinal, então não teremos *overflow* (Exemplos 3.12 e 3.13).
- Por outro lado, se somarmos dois números inteiros de sinais iguais, haverá *overflow* se o resultado tiver sinal diferente, pois terá ultrapassado o limite de representação em memória na notação complemento de 2 (Exemplos 3.16 e 3.17).

3.4 NOTAÇÃO COM EXCESSO

A notação com excesso convencionada que o valor literal de uma sequência de n bits deve ser diretamente proporcional ao valor representado. Ou seja, quanto maior o valor da sequência de bits em notação binária simples, maior o valor representado na base decimal. Dessa forma, o menor valor representável com uma palavra de n bits será descrito por uma sequência de n 0s, ao passo que o maior valor representável será descrito por uma sequência de n 1s.

A notação com excesso também é conhecida por “representação polarizada” [26] ou “excesso x ” [27], onde $x = 2^{n-1} - 1$ e n é o tamanho em bits da palavra. Por exemplo, quando se considera uma palavra de 8 bits, costuma-se dizer “notação com excesso de 127”.



Atenção: O que é o excesso?

Observe bem que a notação com excesso desloca o *valor* representado pela sequência de bits, somando uma constante de *offset*. O posicionamento dos bits na sequência não é modificado.

A Tabela 3 mostra os padrões de bits em notação com excesso e seus valores correspondentes na base decimal e na notação complemento de 2. Veja que, na notação com excesso, as sequências que têm 0 no bit mais significativo são usadas para representar os menores números: os negativos e o zero. Já as sequências cujo bit mais significativo é 1 são usadas para representar os maiores números: apenas os positivos.

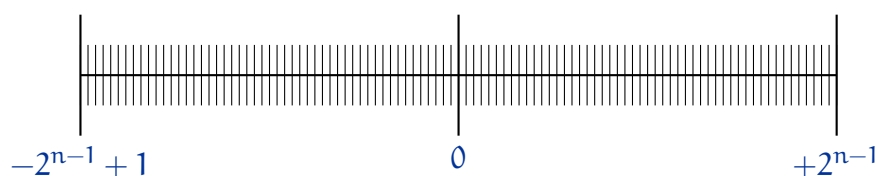
A notação com excesso facilita projeto de circuitos comparadores, pois todas as posições de bits podem ser comparadas segundo os mesmos critérios, sem aumento na complexidade do hardware, nem da programação. Por conta disso, ela é usada para representar o expoente de números reais, como veremos no Capítulo 4.

3.4.1 Propriedades da Notação com Excesso

SINAL. Lembre-se que a premissa básica da notação com excesso é que os valores numéricos representados sejam proporcionais ao valor

Tabela 3: Comparação das notações complemento de 2 e com excesso para uma palavra de n bits.

Decimal	Notação com excesso	Complemento de dois
-2^{n-1}	—	1000 ... 0000
$-2^{n-1} + 1$	0000 ... 0000	1000 ... 0001
$-2^{n-1} + 2$	0000 ... 0001	1000 ... 0010
\vdots	\vdots	\vdots
-2	0111 ... 1101	1111 ... 1110
-1	0111 ... 1110	1111 ... 1111
0	0111 ... 1111	0000 ... 0000
+1	1000 ... 0000	0000 ... 0001
+2	1000 ... 0001	0000 ... 0010
\vdots	\vdots	\vdots
$+2^{n-1} - 2$	1111 ... 1101	0111 ... 1110
$+2^{n-1} - 1$	1111 ... 1110	0111 ... 1111
$+2^{n-1}$	1111 ... 1111	—

Figura 15: Faixa de valores inteiros possíveis de serem representados por uma palavra de n bits na notação com excesso.

nominal da sequência de bits. Consequentemente, os padrões de bits que têm 0 na posição mais significativa representam os menores números, que são os negativos. Já os padrões que têm 1 na posição mais significativa representam os maiores números, que são os positivos. Isso é o contrário do que acontece com a notação complemento de 2.

LIMITES DE REPRESENTAÇÃO. Independentemente da convenção adotada, um espaço de n bits é capaz de representar 2^n números. Na notação com excesso, metade dessas possibilidades serão usadas para representar números positivos, de 1 (1000 ... 0000) a $+2^{n-1}$ (1111 ... 1111). A outra metade de combinações de bits é usada para representar o zero (0111 ... 1111) e os números negativos de -1 (0111 ... 1110) a $-2^{n-1} + 1$ (0000 ... 0000). Portanto, diferente da notação complemento de 2, a notação com excesso representa uma quantidade maior de números positivos do que de negativos. Observe a Figura 15 e compare-a com as Figuras 11 e 14.

PARIDADE. Nas notações vistas anteriormente (binário simples, sinal e magnitude, e complemento de 2), os números terminados em

zero (posição menos significativa da palavra) eram múltiplos da base, ou seja, eram pares, enquanto que os números terminados em 1 eram ímpares. Observe a Tabela 3 e note que essa regra se inverte para números em notação com excesso.

3.4.2 Conversão da Base Decimal para a Notação com Excesso

Existem pelo menos duas formas de se converter um número representado na base 10 para a notação com excesso, com n bits de tamanho:

1. Tomamos o número na base decimal, somamos com $x = 2^{n-1} - 1$ e n ainda na base decimal, e convertemos o resultado para complemento de 2.
2. Tomamos o número na base decimal, convertemos para complemento de 2, e somamos bit a bit com a sequência 0111 ... 1111 (ou seja, o bit zero seguido de 2^{n-1} bits um).

Note que, pela Tabela 3, a sequência 0111 ... 1111 corresponde ao valor zero em notação com excesso, ou seja, essa soma (excesso) *não* altera o valor do número que está sendo convertido.

Os procedimentos acima estão descritos na Figura 16.

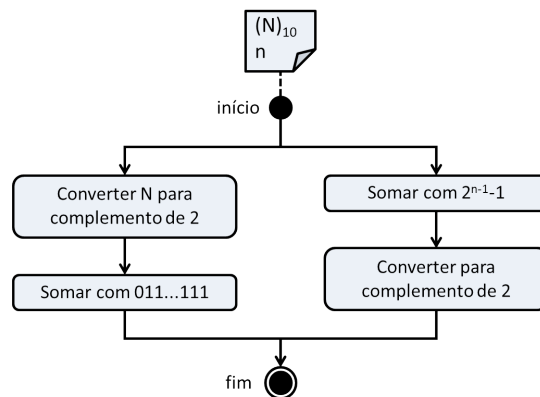


Figura 16: Algoritmo de conversão de um número inteiro expresso na base decimal para a notação com excesso.

Exemplo 3.18. Quanto vale -18 em notação com excesso, considerando-se uma palavra de $n = 8$ bits?

1. Método 1: adicionar excesso ainda na base decimal.

$$\begin{array}{c}
 (-18)_{10} \\
 \downarrow +2^{8-1}-1 \\
 (+109)_{ND} \\
 \downarrow \text{Compl.2} \\
 01101101
 \end{array}$$

Logo, -18 corresponde a 01101101 em notação com excesso de 8 bits.

2. Método 2: adicionar excesso após conversão para complemento de 2.

$$\begin{array}{c}
 (-18)_{10} \\
 \downarrow \text{Compl.2(Exemplo 3.7)} \\
 11101110_{C2} \\
 \downarrow +0111\ 1111 \\
 1\ 01101101
 \end{array}$$

Da mesma forma, -18 corresponde a 01101101 em notação com excesso de 8 bits.

□

Exemplo 3.19. Quanto vale $+18$ em notação com excesso, considerando-se uma palavra de $n = 8$ bits?

1. Método 1: adicionar excesso ainda na base decimal.

$$\begin{array}{c}
 (+18)_{10} \\
 \downarrow +2^{8-1}-1 \\
 (+145)_{ND} \\
 \downarrow \text{Compl.2} \\
 10010001
 \end{array}$$

Logo, $+18$ corresponde a 10010001 em notação com excesso de 8 bits.

2. Método 2: adicionar excesso após conversão para complemento de 2.

$$\begin{array}{c}
 (+18)_{10} \\
 \downarrow \text{Compl.2} \\
 00010010_{C2} \\
 \downarrow +01111111 \\
 10010001
 \end{array}$$

Da mesma forma, $+18$ corresponde a 10010001 em notação com excesso de 8 bits.

□

3.4.3 Conversão da Notação com Excesso para a Base Decimal

Considere a seguinte sequência de n bits representando um número em notação com excesso:

a_{n-1}	a_{n-2}	\dots	a_1	a_0
-----------	-----------	---------	-------	-------

Para encontrar seu valor A na base decimal, procedemos de modo inverso ao da conversão da base decimal para a notação com excesso:

1. Tomamos o número na notação com excesso, convertemos para a base decimal como se fosse binário simples, e subtraímos o valor $x = 2^{n-1} - 1$ do resultado.
2. Tomamos o número na notação com excesso, subtraímos dele a sequência $0111 \dots 1111$ (ou seja, somamos com a sequência $1000 \dots 0001$), e convertemos o resultado para a base decimal.

Exemplo 3.20. Qual o valor, na base dez, de 10101110_{ND} ?

7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	0

1. Método 1: subtrair excesso do número na base decimal.

$$\begin{aligned}
 A &= 1 \cdot 2^7 + \cancel{0 \cdot 2^6} + 1 \cdot 2^5 + \cancel{0 \cdot 2^4} + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + \cancel{0 \cdot 2^0} \\
 &= 2^7 + 2^5 + 2^3 + 2^2 + 2^1 \\
 &= 128 + 32 + 8 + 4 + 2 \\
 &= 174
 \end{aligned}$$

Logo, 10101110_{ND} corresponde a $174 - 127 = 47$ na base decimal.

2. Método 2: subtrair excesso na notação complemento de 2.

$$\begin{array}{r}
 10101110_{ND} \\
 \downarrow +1000\ 0001 \\
 1\ 00101111_{C2} \\
 \downarrow \text{Base } 10 \\
 47
 \end{array}$$

Da mesma forma, 10101110_{ND} corresponde a 47 na base decimal.

□

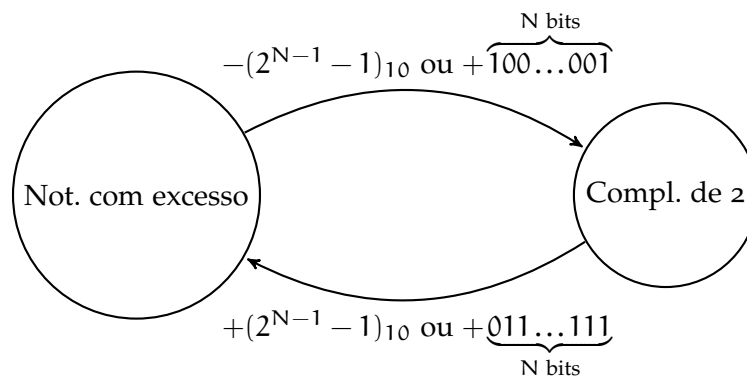
A escolha entre o método 1 ou 2 para conversão é questão pessoal. Depende de sua familiaridade com operações com números binários.

3.5 RESUMO DO CAPÍTULO

Palavras-chave:

- Notação sinal e magnitude
- Notação com excesso
- Notação complemento de 1
- Transbordamento
- Notação complemento de 2
- *Overflow*

Conversão de Notações Sinalizadas de N Bits:



EXERCÍCIOS DO CAPÍTULO 3

- 3.1 Qual a quantidade mínima de bits requeridos para representar o intervalo de valores entre 0 e 65 535 no formato:
- inteiro não sinalizado?
 - complemento de dois?
- 3.2 Qual a faixa de números inteiros representáveis em complemento de 2 utilizando-se palavras dos tamanhos abaixo? Expresse seu resultado em potências de 2 e em valores aproximados de potências de 10.
- 8 bits?
 - 16 bits?
 - 32 bits?
 - 64 bits?
- 3.3 Realize as operações indicadas abaixo em notação binária complemento de 2 de 8 bits.
- $23 - 9$
 - $9 - 23$
 - $-19 - 1$
 - $-19 + 1$
 - $89 + 76$
 - $-89 - 76$
- 3.4 Discuta por que as seguintes afirmações são falsas:
- Para encontrar o valor em base decimal do número 00100101, representado em notação complemento de dois de 8 bits, basta inverter os bits (11011010) e somar 1 (11011011), resultando em $(219)_{10}$.
 - A sequência 1111 representa o mesmo valor em qualquer uma das seguintes notações de 4 bits: complemento de um, complemento de dois e magnitude de sinal.
 - A soma de dois números positivos representados em notação complemento de dois resultará sempre em outro número positivo.
 - A soma de dois números negativos representados em notação complemento de dois resultará sempre em outro número negativo.
 - Overflow é um conceito restrito à representação binária. Se os operandos estivessem na base decimal, não haveria esse problema.

- f) Considerando palavras de 4 bits, toda vez que dois padrões que iniciam por zero (0xxx) e (0xxx) são somados e a sequência resultante inicia por um (1xxx), então podemos afirmar que ocorreu um overflow.
- 3.5 [10, ex. 1.67] Ben e Alissa estão no meio de outra discussão. Ben diz “Todos os inteiros maiores que zero e divisíveis por seis têm exatamente dois dígitos 1 em sua representação binária”. Alissa discorda: “Não, seu leso. São todos os números que têm uma quantidade par de 1s na sua representação binária”. Você concorda com Ben, com Alissa, com ambos ou com nenhum dos dois? Argumente.
- 3.6 [10, ex. 1.68] Ben e Alissa estão no meio de mais outra discussão. Ben diz “Eu posso obter o oposto de um número em complemento de 2 primeiro subtraindo 1, e depois invertendo todos os bits do resultado”. Alissa diz “Não, seu leso. Eu faço isso examinando e copiando cada bit do número, começando pelo menos significativo. Quando encontro o primeiro 1, inverte todos os outros bits subsequentes”. Você concorda com Ben, com Alissa, com ambos ou com nenhum dos dois? Argumente.
- 3.7 O que é um *overflow* e como ele pode ser detectado? Como o *overflow* em números sem sinal difere do *overflow* em números com sinal?
- 3.8 [20, ex. 9.2] Prove as seguintes afirmações:
- Um número inteiro x expresso na notação binária sem sinal é uma potência de dois se, e somente se, o resultado da operação lógica AND bit-a-bit entre x e $x - 1$ for zero.
 - Um número inteiro binário sem sinal $(a_{n-1} a_{n-2} \dots a_1 a_0)_2$ é divisível por 3 se, e somente se, $\sum_{i \text{ par}} a_i - \sum_{i \text{ ímpar}} a_i$ for múltiplo de 3.
- 3.9 [20, ex. 9.7] Números *negabinários* utilizam o conjunto de dígitos $\{0, 1\}$ e a base $B = -2$. O valor de um número negabinários na base 10 é calculado da mesma maneira que um número binário, mas com os termos contendo potências ímpares da base sendo negativos. Assim, tanto números positivos e negativos podem ser representados sem a necessidade de um bit de sinal separado ou um esquema de complementação.
- Qual o intervalo de valores representáveis em uma representação negabinária de 3 bits?
 - E de 10 bits?
 - Dado um número negabinário, como você determina o seu sinal?

REPRESENTAÇÃO DE NÚMEROS REAIS

O CONJUNTO DOS NÚMEROS REAIS é formado por todos os valores que representam uma quantidade ao longo de uma reta contínua. Assim, os números reais compreendem todos os números racionais, tais como o inteiro -42 e a fração $7/4$, e todos os números irracionais, tais como $\sqrt{2}$ (1,41421356..., um número irracional algébrico) e π (3,14159265..., um número transcendental).

Os números reais são incontáveis. Isso implica que o conjunto dos números reais \mathbb{R} não pode ser associado biunivocamente ao conjunto dos números naturais \mathbb{N} . Ou seja, embora ambos os conjuntos sejam infinitos, não existe uma função que associe os elementos de um conjunto aos elementos do outro. Dessa forma, dizemos que a cardinalidade de \mathbb{R} é maior que a cardinalidade de \mathbb{N} . Em outras palavras, o infinito de \mathbb{R} é muito maior que o infinito de \mathbb{N} .

No Capítulo 2, vimos que representar números naturais em sistemas computacionais não é uma tarefa trivial, pois o espaço em memória é um recurso finito, ao passo que a quantidade de valores a serem representados é infinita. Tal dificuldade se amplifica quando temos que representar números reais. Uma notação com essa finalidade deve ser capaz de representar números muito grandes, tal como a quantidade de átomos no universo observável, e também números muito pequenos, como a massa de um elétron.

Além disso, entre a menor e a maior grandeza que se deseja representar, existe uma infinidade de valores intermediários, tais como as dízimas e os números irracionais, que nunca poderão ser representados de forma exata utilizando-se uma quantidade limitada de bits. Portanto, os resultados de operações envolvendo números reais representados em computador frequentemente devem ser arredondados de modo a caber em um espaço finito de memória [7].

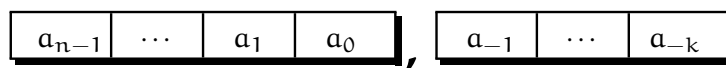
4.1 REPRESENTAÇÃO DE NÚMEROS REAIS EM BASE BINÁRIA

Os números reais costumam ser representados em duas partes: inteira e fracionária. O sinal de vírgula marca a separação entre elas. Já nos países de língua inglesa, utiliza-se o ponto no lugar da vírgula. Esse fato explica o nome das notações que veremos mais adiante: ponto fixo e ponto flutuante.

Considere um número real qualquer expresso na base decimal, contendo n algarismos na parte inteira e k algarismos na parte fracionária. Logo, sua *representação* é dada por uma sequência de $n + k$ algarismos, indicados por a_i , $-k \leq i < n$:

Números **algébricos** são aqueles que podem ser a raiz de uma equação polinomial com coeficientes racionais. Números **transcendentais**, “vão além” dos números algébricos porque não podem ser expressos apenas com raízes de tais equações.

A **cardinalidade** de um conjunto é a quantidade de elementos desse conjunto.



Por exemplo, o número 98,765 possui cinco algarismos, sendo $n = 2$ na parte inteira e $k = 3$ na parte fracionária. Ele é composto pela seguinte soma:

$$98,765 = 9 \cdot 10^1 + 8 \cdot 10^0 + 7 \cdot B^{-1} + 6 \cdot B^{-2} + 5 \cdot B^{-3}$$

4.1.1 Conversão de Números Reais da Base Binária para Decimal

Como consequência do raciocínio anterior, podemos estender a eq. 3 (pág. 22) para converter números reais expressos em uma base B qualquer para a base decimal:

$$(N)_{10} = a_{n-1} \cdot B^{n-1} + \cdots + a_1 \cdot B^1 + a_0 \cdot B^0 + a_{-1} \cdot B^{-1} + \cdots + a_{-k} \cdot B^{-k} \quad (7)$$

Como vimos no Capítulo 2, na base binária, os pesos das posições da parte inteira são potências positivas de 2. Logo, pela eq. 7, os pesos das posições da parte fracionária são potências *negativas* de 2.

Lembramos mais uma vez que *todos os cálculos devem ser realizados na aritmética da base de destino*. Ou seja, cada algarismo a_i da base B e o próprio valor da base B devem ser expressos na base 10.

Exemplo 4.1. Converter $(101,101)_2$ para a base dez.



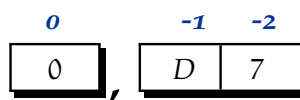
$$(N)_{10} = 1 \cdot 2^2 + \cancel{0 \cdot 2^1} + 1 \cdot 2^0 + 1 \cdot 2^{-1} + \cancel{0 \cdot 2^{-2}} + 1 \cdot 2^{-3}$$

$$(N)_{10} = 4 + 0 + 1 + 0,5 + 0 + 0,125 = 5,625$$

□

Note que o mesmo procedimento pode ser estendido para converter frações de outras bases para a base decimal.

Exemplo 4.2. Converter $(0,D7)_{16}$ para a base dez.



$$(N)_{10} = 13 \cdot 16^{-1} + 7 \cdot 16^{-2}$$

$$(N)_{10} = 13 \cdot 0,0625 + 7 \cdot 0,00390625$$

$$(N)_{10} = 0,8125 + 0,02734375 = 0,83984375$$

□



Atenção: Não converta grupos de bits da parte fracionária como se fossem inteiros

No Exemplo 4.1, a sequência de bits 101 corresponde a $(5)_{10}$ na parte inteira. Porém, na parte fracionária, ela corresponde a $(0,625)_{10}$. Portanto, lembre-se que a conversão de número reais da base binária para a decimal é realizada de acordo com o peso de cada posição, e não por grupos de bits.

Exemplo 4.3. Converter $(0,0001)_2$ para a base dez.

0	-1	-2	-3	-4
0	0	0	0	1

$$(N)_{10} = 1 \cdot 2^{-4}$$

$$(N)_{10} = \frac{1}{16} = 0,0625$$

□

O Exemplo 4.3 sugere outro fato interessante: números reais em base binária podem ser facilmente convertidos em frações decimais. Por exemplo, na base decimal, o número 0,8 pode ser escrito como $\frac{8}{10}$, o número 0,67 pode ser escrito como $\frac{67}{100}$, o número 0,123 pode ser escrito como $\frac{123}{1000}$, e assim por diante.

De modo similar, a parte fracionária de um número binário pode ser convertida em fração, como mostra a Tabela 4. O algoritmo de conversão funciona em quatro passos:

Tabela 4: Números binários podem ser convertidos em frações decimais.

Fonte: [4].

Binário	Fração	Decimal
$0,0_2$	$\frac{0}{2}$	$0,0_{10}$
$0,01_2$	$\frac{1}{4}$	$0,25_{10}$
$0,010_2$	$\frac{2}{8}$	$0,25_{10}$
$0,0011_2$	$\frac{3}{16}$	$0,1875_{10}$
$0,00110_2$	$\frac{6}{32}$	$0,1875_{10}$
$0,001101_2$	$\frac{13}{64}$	$0,203125_{10}$
$0,0011010_2$	$\frac{26}{128}$	$0,203125_{10}$
$0,00110011_2$	$\frac{51}{256}$	$0,19921875_{10}$

1. Desloque a vírgula em tantas k casas para a direita de modo que o número deixe de ter parte fracionária.
2. Converta o número inteiro assim obtido para a base decimal. Use-o como *numerador*.
3. Calcule 2^k . Use o resultado como *denominador*.
4. Monte a fração, dividindo o numerador pelo denominador.

Na base decimal, deslocar a vírgula de um número real para a direita é mesmo que multiplicá-lo por 10. Por exemplo, $12,34 \times 10 = 123,4$. Da mesma maneira, deslocar a vírgula para a esquerda é mesmo que dividi-lo por 10. Por exemplo, $12,34 \div 10 = 1,234$.

Já na base binária, o deslocamento da vírgula fará com que o número seja multiplicado ou dividido pelo valor da base, que é 2. Logo, $0,01_2 \times 2 = 0,1_2$, ou seja, $0,25_{10} \times 2 = 0,5_{10}$. Do mesmo jeito, $0,01_2 \div 2 = 0,001_2$, ou seja, $0,25_{10} \div 2 = 0,125_{10}$.



Quase 1,0

Na base decimal, números no formato $0,99 \dots 9$ representam valores bem próximos de 1,0. Analogamente, na base binária, números no formato $0,11 \dots 1_2$ representam valores imediatamente abaixo de 1,0 para a quantidade de posições utilizadas.



Comparação entre frações em bases diferentes

Quando tomamos um número inteiro, 11 por exemplo, e duas bases B_1 e B_2 , seu valor será maior quando interpretado na maior das bases. Por exemplo $(11)_{10}$ é maior que $(11)_2$, que vale $(5)_{10}$. Por outro lado, com números fracionários, ocorre o inverso, pois os pesos de cada posição são potências negativas da base. Assim, $(0,11)_{10}$ equivale a onze partes de cem, ao passo que $(0,11)_2$ equivale a $0,50 + 0,25 = 0,75$, ou seja, setenta e cinco partes de cem. Portanto, $(0,11)_{10} < (0,11)_2$, embora $(11)_{10} > (11)_2$.

4.1.2 Conversão de Números Reais da Base Decimal para a Binária

Para converter números reais da base decimal para a base binária, adotamos um procedimento constituído de duas etapas. Na primeira

etapa, convertemos a parte *inteira*, usando o método das *divisões* sucessivas (Seção 2.3). Na segunda etapa, convertemos a parte *fracionária*, usando o método das *multiplicações* sucessivas, similar ao anterior:

1. Tome a parte fracionária $(F)_{10}$ do número a ser convertido e a multiplique por 2.
2. Coloque a parte inteira do resultado (0 ou 1) à direita da parte fracionária $(F)_2$ do novo número na base binária.
3. Repita os passos 1 e 2 com a parte fracionária do resultado, até que o resultado seja zero ou até alcançar o número de dígitos (precisão) desejado.

Na verdade, o procedimento acima pode ser aplicado para qualquer outra base $B > 1$, conforme o fluxograma da Figura 17.

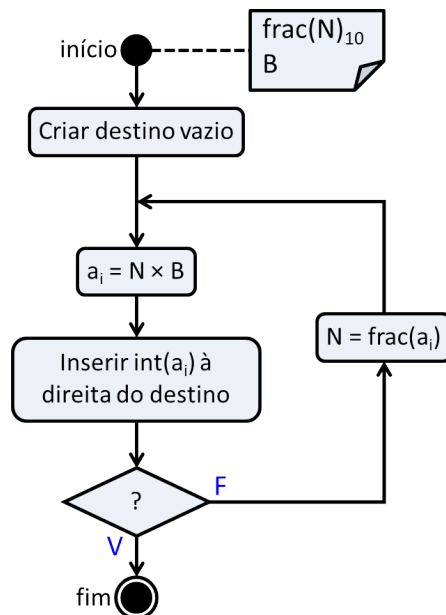


Figura 17: Algoritmo de conversão de um número real fracionário expresso na base decimal para uma base B qualquer.

Exemplo 4.4. Converter $(0,171875)_{10}$ para a base binária.

$0,171875 \times 2$	$=$	$\begin{bmatrix} 0 \\ ,34375 \end{bmatrix}$	\Rightarrow Fração: 0,0
$0,34375 \times 2$	$=$	$\begin{bmatrix} 0 \\ ,6875 \end{bmatrix}$	\Rightarrow Fração: 0,00
$0,6875 \times 2$	$=$	$\begin{bmatrix} 1 \\ ,375 \end{bmatrix}$	\Rightarrow Fração: 0,001
$0,375 \times 2$	$=$	$\begin{bmatrix} 0 \\ ,75 \end{bmatrix}$	\Rightarrow Fração: 0,0010
$0,75 \times 2$	$=$	$\begin{bmatrix} 1 \\ ,5 \end{bmatrix}$	\Rightarrow Fração: 0,00101
$0,5 \times 2$	$=$	$\begin{bmatrix} 1 \\ ,0 \end{bmatrix}$	\Rightarrow Fração: 0,001011

Como atingimos zero na parte fracionária a partir da quinta multiplicação, terminamos a conversão. Logo, $(0,171875)_{10} = (0,001011)_2$.

□

Exemplo 4.5. Converter $(0,6)_{10}$ para a base binária.

$$\begin{array}{rcll}
 0,6 \times 2 & = & \boxed{1} & ,2 \Rightarrow \text{Fração: } 0,1 \\
 0,2 \times 2 & = & \boxed{0} & ,4 \Rightarrow \text{Fração: } 0,10 \\
 0,4 \times 2 & = & \boxed{0} & ,8 \Rightarrow \text{Fração: } 0,100 \\
 0,8 \times 2 & = & \boxed{1} & ,6 \Rightarrow \text{Fração: } 0,1001 \\
 0,6 \times 2 & = & \boxed{1} & ,2 \Rightarrow \text{Fração: } 0,10011 \\
 0,2 \times 2 & = & \boxed{0} & ,4 \Rightarrow \text{Fração: } 0,100110
 \end{array}$$

Note que as multiplicações sucessivas continuarão infinitamente, pois a parte fracionária nunca chegará a ser zero. Como resultado, obtemos uma dízima em base binária de período 1001₂. Logo, $(0,6)_{10} = (0,\overline{1001})_2$.

□



Dízimas são uma questão de ponto de vista

A fração 0,6 é um número exato na base decimal. No entanto, é uma dízima periódica na base binária. Da mesma maneira, $\frac{1}{3} = 0,3333\dots$ é uma dízima periódica na base decimal, mas na base ternária esse valor é dado simplesmente por 0,1.

Exemplo 4.6. Converter $(0,72)_{10}$ para a base binária com 8 bits de precisão na parte fracionária.

$$\begin{array}{rcll}
 0,72 \times 2 & = & \boxed{1} & ,44 \Rightarrow \text{Fração: } 0,1 \\
 0,44 \times 2 & = & \boxed{0} & ,88 \Rightarrow \text{Fração: } 0,10 \\
 0,88 \times 2 & = & \boxed{1} & ,76 \Rightarrow \text{Fração: } 0,101 \\
 0,76 \times 2 & = & \boxed{1} & ,52 \Rightarrow \text{Fração: } 0,1011 \\
 0,52 \times 2 & = & \boxed{1} & ,04 \Rightarrow \text{Fração: } 0,10111 \\
 0,04 \times 2 & = & \boxed{0} & ,08 \Rightarrow \text{Fração: } 0,101110 \\
 0,08 \times 2 & = & \boxed{0} & ,16 \Rightarrow \text{Fração: } 0,1011100 \\
 0,16 \times 2 & = & \boxed{0} & ,32 \Rightarrow \text{Fração: } 0,10111000
 \end{array}$$

Apesar de não termos atingido o zero, chegamos à precisão desejada (oito bits). Por isso encerramos a conversão na oitava multiplicação. Logo, $(0,72)_{10} = (0,10111000)_2$.

Se não houvesse limitação de bits e continuássemos o procedimento, veríamos que um período começaria a se formar somente a partir do 16º dígito. Nesse caso, $(0,72)_{10} = (0,\overline{1011\ 1000\ 0101\ 0001\ 1110})_2$.

□

DEMONSTRAÇÃO DO MÉTODO DAS MULTIPLICAÇÕES SUCESSIVAS. Por que o método das multiplicações sucessivas funciona? Note que a parte fracionária F de um número real é um valor pertencente ao intervalo $[0; 1[$. Em notação binária, $(F)_2$ é expressa assim:

$$\boxed{0}, \boxed{a_{-1}} \boxed{a_{-2}} \boxed{a_{-3}} \boxed{a_{-4}} \dots$$

$a_i \in \{0, 1\}, \forall i \in \mathbb{Z}_-$, ou seja, cada dígito a_i só pode assumir os valores 0 ou 1.

Consequentemente, a partir da eq. 7, temos que $(F)_{10}$ é dado por:

$$\begin{aligned} (F)_{10} &= a_{-1} \cdot 2^{-1} + a_{-2} \cdot 2^{-2} + a_{-3} \cdot 2^{-3} + \dots \\ (F)_{10} &= (a_{-1}) \cdot 2^{-1} + (a_{-2} \cdot 2^{-1}) \cdot 2^{-1} + (a_{-3} \cdot 2^{-2}) \cdot 2^{-1} + \dots \\ (F)_{10} &= 2^{-1} \cdot (a_{-1} + (a_{-2} \cdot 2^{-1}) + (a_{-3} \cdot 2^{-2}) + \dots) \\ (F)_{10} &= 2^{-1} \cdot (a_{-1} + (a_{-2} \cdot 2^{-1}) + (a_{-3} \cdot 2^{-1}) \cdot 2^{-1} + \dots) \\ (F)_{10} &= 2^{-1} \cdot (a_{-1} + 2^{-1} \cdot (a_{-2} + 2^{-1} \cdot (a_{-3} + \dots))) \end{aligned}$$

Se multiplicarmos cada lado da expressão anterior por 2, temos:

$$2 \cdot (F)_{10} = a_{-1} + \underbrace{2^{-1} \cdot (a_{-2} + 2^{-1} \cdot (a_{-3} + \dots))}_{X_1}$$

Da expressão acima, note que a_{-1} , que é um valor entre zero e um, é a parte inteira de $2 \cdot (F)_{10}$. Se tomarmos a parte fracionária X_1 de $2 \cdot (F)_{10}$ e a multiplicarmos por 2, temos:

$$2 \cdot X_1 = a_{-2} + 2^{-1} \cdot (a_{-3} + \dots)$$

Assim, obteremos o próximo dígito a_{-2} de $(F)_2$. Repetindo esse processo sucessivamente, encontraremos os demais bits a_i de $(F)_2$.

Dessa maneira, mostramos que os métodos de conversão de base são apenas variantes da fórmula de conversão polinomial básica (eq. 7). O que muda de um método para outro é que cada um busca manipular os algoritmos da forma mais conveniente, procurando manter a maior parte das operações na base decimal, com a qual estamos familiarizados.

4.1.3 Erros de Arredondamento

Considere uma palavra fracionária binária de quatro bits de tamanho. Considere ainda duas sequências consecutivas de 4 bits, 0,1110 e 0,1111. Seus valores na base decimal são dados por:

$$\begin{array}{lcl} 0,1110 & \Rightarrow & 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} \Rightarrow 0,8750 \\ 0,1111 & \Rightarrow & 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} \Rightarrow 0,9375 \end{array}$$

Percebe-se que a fração decimal 0,9270 não pode ser representada de forma exata usando-se apenas 4 bits: quando ela é convertida para a base binária, obtemos $(0,111011010100\dots)_2$, mas o valor truncado em quatro bits, $(0,1110)_2$, corresponde a $(0,8750)_{10}$.

A única maneira de solucionar o problema é reservar mais bits para a representação binária. Porém, isso requer mais espaço em memória e, conseqüentemente, traz um custo maior associado.

4.1.4 Limitações da Notação Fixa

Uma maneira ingênua de armazenar números reais em memória é seguir as mesmas etapas pelas quais foram convertidos: reservar um espaço para a parte inteira e outro espaço para a parte fracionária.



Essa abordagem tem um problema: para atender às diversas aplicações do cotidiano, ela precisa reservar espaço suficiente para armazenar números muito grandes ($\gg 2^n$) e também números muito pequenos ($\ll 2^{-k}$). Contudo, raramente há aplicações em que valores muito grandes e muito pequenos são manipulados simultaneamente. Portanto, reservar espaços distintos para a parte inteira e para a parte fracionária desperdiça recursos computacionais.

Por conta dessa limitação, a notação de ponto fixo é raramente usada em sistemas computacionais para representar números reais. Na Seção 4.2.1 a seguir, veremos que, na prática, os números reais são mais comumente representados em termos de expoente e fração, em vez de parte inteira e parte fracionária.

4.2 REPRESENTAÇÃO EM PONTO FLUTUANTE

<http://news.bbc.co.uk/2/hi/science/nature/3085885.stm>

Estima-se que o número de átomos no universo observável seja em torno do seguinte valor:

70 000 000 000 000 000 000 000

<http://arxiv.org/pdf/1203.5425v1.pdf>

Por outro lado, a massa de um elétron foi determinada pelo físico J. J. Thomson em 1897 como tendo o seguinte valor em quilogramas:

0,000 000 000 000 000 000 000 000 000 910 938 291

A notação científica é uma forma de escrever números reais muito grandes ou muito pequenos como o produto de um número no intervalo $[1;10[$ (significando) por uma potência da base 10.

Esses números, quando expressos em notação comum, são difíceis de serem lidos: bilhões, trilhões, quatrilhões? Bilionésimos, trilionésimos, quatrilionésimos? Por isso, no meio acadêmico, utilizamos a notação científica para representar números muito grandes ou muito pequenos. Assim, o número de átomos no universo é expresso mais facilmente como 7×10^{22} e a massa do elétron como $9,10938291 \times 10^{-31}$.

A notação científica desloca a vírgula decimal para um lugar conveniente, escrevendo a parte significativa do número com uma menor quantidade de dígitos. O sentido e a quantidade de posições deslocadas são compensados no expoente em base dez.

A notação científica é a base para representar números reais como *ponto flutuante*. Em inglês, o ponto é usado no lugar da vírgula para separar a parte inteira da fracionária de um número real. O adjetivo flutuante vem da possibilidade de deslocar o ponto decimal.

Após vários anos de experimentações com diversos formatos de ponto flutuante, sofrendo com resultados inconsistentes e incompatíveis, a indústria de computadores adotou o formato IEEE 754, proposto pelo *Institute of Electrical and Electronics Engineers* [20, 7]. Ou seja, embora nossa discussão daqui para frente trate apenas do padrão IEEE 754, enfatizamos que essa não é a única opção possível para representar números ponto flutuante. É apenas a opção que a maioria dos sistemas computacionais utiliza na atualidade.

4.2.1 O Padrão IEEE 754

Para converter um número real em notação ponto flutuante, começamos obtendo sua notação científica em base binária. Do Exemplo 4.4, vimos que $(0,171875)_{10} = (0,001011)_2$. Logo, sua representação em notação científica binária é a seguinte:

$$+1,011 \times 2^{-3}$$

Note que a expressão acima utiliza base de numeração *mista*: a parte significativa está expressa em binário; já a base 2 e seu expoente -3 estão expressos em decimal.

Generalizando, um sistema numérico de ponto flutuante $\mathbb{F} \subset \mathbb{R}$ qualquer é um subconjunto dos números reais cujos elementos x têm a seguinte forma na base $B > 1$:

$$x = (-1)^s \times M \times B^E \quad (8)$$

onde:

- s é o sinal de x , que indica se x é positivo ($s = 0$) ou negativo ($s = 1$);
- $1 \leq M < B$ é o *significando* e tem p dígitos; e
- E é o expoente de x na base B e corresponde à quantidade de posições que a vírgula deve ser deslocada para que $1 \leq M < B$.

O significando deve ser composto por um único algarismo diferente de zero na parte inteira e os demais algarismos devem estar na parte fracionária. No caso da base binária ($B = 2$), isso implica que a parte inteira deve conter apenas um único algarismo 1. A quantidade

Precisão vem do latim praecisus (cortar antes). Nas ciências, refere-se à menor unidade de medida usada para expressar um valor aproximado, ou seja, à menor medida com que um intervalo é "cortado".

p de bits do significando determina a *precisão*, ou seja, a granularidade do intervalo de números reais representados.



Atenção: *Significando ou Mantissa?*

Alguns autores usam o termo *mantissa* para se referir ao significando. Porém, há dois problemas com esse termo:

1. *Mantissa* denota a parte fracionária de um logaritmo, que não é a mesma coisa que a parte fracionária de um número ponto flutuante.
2. É uma designação obsoleta, como relata David Goldberg no seu famoso artigo *What every computer scientist should know about floating-point arithmetic* [7], de 1991.

Portanto, neste texto, usamos o termo *significando* para denotar a parte significativa de um número ponto flutuante normalizado, composto por sua parte inteira (sempre igual a 1) e pela sua parte fracionária.

Em sistemas computacionais, números ponto flutuante têm base $B = 2$ e são representados por meio de três campos:

SINAL Um único bit que indica se o número é positivo ou negativo.

EXPOENTE Conjunto de k bits que representam o expoente do número ponto flutuante normalizado. Utiliza a notação com excesso, em vez da notação complemento de 2. Ela é explicada na Seção 3.4.

FRAÇÃO Conjunto de n bits localizados à direita da vírgula (parte fracionária do significando). Logo, $M = 1 + f$ e $p = n + 1$. A parte inteira não é representada nos formatos float e double, pois assume-se implicitamente que ela é sempre igual a 1.

O padrão IEEE 754 define três formatos básicos de precisão, resumidos na Tabela 5. A Figura 18 mostra como o padrão IEEE 754 convencionou a montagem dos campos de sinal (s), expoente (E) e fração (f) na memória de sistemas computacionais.

Nas subseções a seguir, detalhamos como os três campos dos números de ponto flutuante são representados segundo o padrão IEEE 754, tomando o formato de precisão simples (32 bits) como referência.

Tabela 5: Quantidade de bits ocupadas pelos formatos de número ponto flutuante definidos pelo padrão IEEE 754.

Formato	Total	Sinal	Expoente	Parte inteira	Parte fracionária
Precisão simples	32 bits	1	8	—	23
Precisão dupla	64 bits	1	11	—	52
Precisão estendida	80 bits	1	15	1	63



Figura 18: Codificação dos campos de um número ponto flutuante segundo o padrão IEEE 754. A extensão dos campos “expoente” e “fração” diferencia os formatos de precisão entre si.

4.2.1.1 Sinal

O sinal do número representado é indicado pelo bit mais significativo da sequência de bits. Independente do formato de precisão adotado (simples, duplo ou estendido), este campo sempre ocupa um único bit. A convenção adotada para representar o sinal é a mesma utilizada na notação Sinal e Magnitude (Seção 3.1): 0, se positivo; 1, se negativo.

4.2.1.2 Expoente

O expoente de um número ponto flutuante é um número inteiro. Ele representa a quantidade de posições (casas) binárias em que a vírgula foi deslocada para que o número ponto flutuante ficasse normalizado, ou seja, com um único dígito 1 na parte inteira.

O objetivo do expoente é manter o valor total do número após o deslocamento da vírgula:

- Quando ele é *positivo*, significa que o número normalizado ficou menor que o original após a vírgula ter sido deslocada para a *direita*.
- Quando ele é *negativo*, significa que o número normalizado ficou maior que o original após a vírgula ter sido deslocada para a *esquerda*.

Na notação IEEE 754, o campo expoente é armazenado imediatamente à direita do bit de sinal. Qual a razão disso? Facilitar a comparação. Por exemplo, quando comparamos $1,000011 \times 2^{-3}$ com $1,00111 \times 2^{+5}$, pouco importa olhar a parte fracionária. Apenas comparando os expoentes já somos capazes de determinar que $1,00111 \times 2^{+5}$ é o maior entre os dois, pois tem o maior expoente.

Adicionalmente, o campo expoente é representado em *notação com excesso*, vista na Seção 3.4. Basicamente, tomamos o expoente original do número a ser representado e somamos a um valor fixo, chamado *excesso* (*bias*, em inglês). O valor do excesso é $2^{k-1} - 1$, sendo

k o tamanho do campo expoente (8 bits na precisão simples, 11 bits na precisão dupla). Dessa forma, os expoentes negativos são sempre representados como positivos, em notação binária, o que facilita o projeto de circuitos comparadores.

4.2.1.3 Parte Inteira

Nos formatos de precisão simples e dupla, a parte inteira do significando é omitida, pois assume-se que ela é sempre igual a 1, devido à normalização.

Já no formato de precisão estendida (80 bits), a parte inteira é representada por um bit entre os campos “expoente” e “fração”. Se for 0, indica que o valor representado está desnormalizado; se for 1, indica que está normalizado. Detalhes mais específicos da notação estendida podem ser encontrados na versão 2008 do padrão IEEE-754 [12], ou na Wikipedia [30].

4.2.1.4 Fração

Para obter o campo *fração* de um número ponto flutuante x , expresso na base decimal, adote o seguinte procedimento:

1. Converta o número real $(x)_{10}$ da base decimal para a binária. Primeiro, converta a parte inteira, usando o método de divisões sucessivas. Depois, a parte fracionária, usando o método das multiplicações sucessivas.
2. Normalize o número, deslocando a vírgula de $(x)_2$ até que reste apenas um único algarismo 1 na parte inteira.
3. Altere o valor do expoente, conforme o número de posições deslocadas.
4. Se a parte fracionária do número normalizado tiver menos de 23 bits de tamanho, complete com zeros (para números exatos em binário) ou com o período (para dízimas binárias). Esta será a fração.
5. Se a parte fracionária do número normalizado tiver mais de 23 bits de tamanho, elimine os bits menos significativos excedentes. Esta será a fração.

A normalização é necessária para garantir que cada número ponto flutuante tenha apenas uma única representação em formato binário. Por exemplo, o número $(23)_{10} = (10111)_2$ pode ser expresso de diversas maneiras equivalentes:

$$\begin{aligned}
+101,11 &\times 2^{+3} \\
+1,0111 &\times 2^{+4} \\
+0,10111 &\times 2^{+5} \\
+1011100 &\times 2^{-2}
\end{aligned}$$

Uma vez que a parte inteira de um número normalizado é sempre 1, é desnecessário armazená-la. Por isso, esse bit fica implícito à notação IEEE 754. Essa convenção permite representar mais um bit da parte fracionária, conferindo maior precisão.

4.2.1.5 Exemplos

Agora vamos juntar tudo em casos de exemplo, convertendo números reais da base decimal para a notação IEEE 754 e vice-versa.

Exemplo 4.7. Converta o valor $(0,171875)_{10}$ para a notação IEEE 754 com precisão simples.

SINAL. Como o sinal é positivo, então o bit de sinal é 0.

EXPOENTE. Do Exemplo 4.4, sabemos que $(0,171875)_{10} = (0,001011)_2 = +1,011 \times 2^{-3}$. Portanto, o expoente é $(-3)_{10}$. Convertendo-o para a notação com excesso, somamos $2^{8-1} - 1 = 127$ de excesso, obtendo 124. Na base binária, $124_{10} = (1111100)_2$. Logo, o campo expoente é 01111100.

FRAÇÃO. O campo fração de $1,011 \times 2^{-3}$ no padrão IEEE 754 precisão simples será dado pela parte fracionária 011 seguida de vinte zeros, para completar o tamanho de 23 bits (Tabela 5).

Portanto, o número $(0,171875)_{10}$ será representado da seguinte maneira na memória de um computador, segundo a notação IEEE 754:

0	0111 1100	0110 0000 0000 0000 0000 000
---	-----------	------------------------------

□

Exemplo 4.8. Converta o valor $(-27,6)_{10}$ para a notação IEEE 754 com precisão simples.

SINAL. Como o sinal é negativo, então o bit de sinal é 1.

EXPOENTE. No Exemplo 4.5, vimos que a fração $(0,6)_{10}$ forma uma dízima quando convertida em binário: $(0,1001\ 1001\ \dots)_2$. Já a parte inteira $(27)_{10} = (11011)_2$. Logo:

$$(-27,6)_{10} = -11011,1001\dots = -1,1011\overline{1001} \times 2^{+4}$$

Somando o expoente 4 com o excesso 127, temos 131, o que equivale a $(10000011)_2$.

FRAÇÃO. Estendendo a parte fracionária para 23 bits, temos a seguinte representação em memória de computador:

1	1000 0011	1011 1001 1001 1001 1001 100
---	-----------	------------------------------

□

Exemplo 4.9. A sequência 1 10000000 111000000000000000000000 corresponde a que valor em notação IEEE 754 com precisão simples?

SINAL. Como o bit de sinal é 1, então o sinal do número é negativo.

EXPOENTE. O campo expoente é 10000000, que equivale a $(128)_{10}$. Subtraindo o excesso 127 desse valor, ficamos com um expoente igual a $(1)_{10}$.

FRAÇÃO. A partir da eq. 8, recuperamos a parte inteira implícita igual a 1 no número normalizado, de modo que temos:

$$\begin{aligned}x &= -(1,111)_2 \times 2^1 \\x &= -(11,11)_2\end{aligned}$$

1	0	-1	-2
1	1	1	1

Agora, usamos a eq. 7 para converter $(11,11)_2$ para a base decimal:

$$\begin{aligned}x &= -(2^{+1} + 2^0 + 2^{-1} + 2^{-2}) \\x &= -(2 + 1 + 0,5 + 0,25) \\x &= -3,75\end{aligned}$$

Outro modo de chegar à resposta final é raciocinar assim:

$$\begin{aligned}x &= -(1,111)_2 \times 2^1 \\x &= -(1111)_2 \times 2^{-2} \\x &= -15 \times \frac{1}{4} \\x &= -\frac{15}{4} \\x &= -3,75\end{aligned}$$

□

Exemplo 4.10. A sequência 0x3F50 0000 corresponde a que valor em notação IEEE 754 com precisão simples?

Sabemos que a sequência de bits está expressa em base hexadecimal, por conta do símbolo “0x”. Assim, primeiro devemos convertê-la para notação binária:

3	F	5	0	0	0	0	0
↓	↓	↓	↓	↓	↓	↓	↓
0011	1111	0101	0000	0000	0000	0000	0000

Em seguida, arrumamos os bits segundo a notação IEEE 754 com precisão simples: da esquerda para a direita, 1 bit para o sinal, 8 bits para o expoente, e os 23 bits restantes para a fração:

0	0111 1110	1010 0000 0000 0000 0000 000
---	-----------	------------------------------

SINAL. Como o bit de sinal é 0, então o sinal do número é positivo.

EXPOENTE. O campo expoente é 01111110, que equivale a $(126)_{10}$. Subtraindo o excesso 127 desse valor, ficamos com um expoente igual a $(-1)_{10}$.

FRAÇÃO. A partir da eq. 8, recuperamos a parte inteira implícita igual a 1 no número normalizado, de modo que temos:

$$\begin{aligned}x &= (1,101)_2 \times 2^{-1} \\x &= (0,1101)_2\end{aligned}$$

<i>0</i>	<i>-1</i>	<i>-2</i>	<i>-3</i>	<i>-4</i>
0	1	1	0	1

Agora, usamos a eq. 7 para converter $(0,1101)_2$ para a base decimal:

$$\begin{aligned}x &= (2^{-1} + 2^{-2} + 2^{-4}) \\x &= (0,5 + 0,25 + 0,0625) \\x &= 0,8125\end{aligned}$$

Outro modo de chegar à resposta final é raciocinar assim:

$$\begin{aligned}x &= (1,101)_2 \times 2^{-1} \\x &= (1101)_2 \times 2^{-4} \\x &= 13 \times \frac{1}{16} \\x &= \frac{13}{16} \\x &= 0,8125\end{aligned}$$

□

4.3 CASOS ESPECIAIS

Nem todas as 2^{32} combinações de bits em uma palavra de 32 bits são usadas para representar números ponto flutuante. A notação IEEE 754 reserva algumas sequências de bits para representar algumas situações especiais, tais como o zero, infinito, divisão por zero, entre outras. Ao longo desta seção, veremos os seguintes casos especiais:

1. Representação do zero.
2. Representação do infinito.
3. Situações inválidas (NaN – *Not a Number*).
4. Números desnormalizados.

4.3.1 Como Representar o Zero na Notação IEEE 754?

Devido à normalização, não é possível representar o zero pela eq. 8. Ou seja, como a parte inteira implícita sempre vale 1, não há como obter zero seguindo a regra geral da notação IEEE 754. Por conta disso, o zero é tratado como caso especial.

Quando todos os bits do campo expoente e do campo fração são iguais a zero, a notação IEEE 754 convencionou que o número representado não é $1,0 \times 2^{-127}$, mas sim o próprio valor 0:

0	0000 0000	0000 0000 0000 0000 0000 000
1	0000 0000	0000 0000 0000 0000 0000 000

4.3.2 Infinito

Quando todos os bits do campo expoente são iguais a 1 e todos os bits do campo fração são iguais 0, a notação IEEE 754 convencionou que o valor representado é o infinito positivo ($+\infty$) ou o infinito negativo ($-\infty$), dependendo do bit de sinal. Nesse caso, o valor $\pm 1,0 \times 2^{128}$ não é representado, para dar lugar à representação do infinito.

0	1111 1111	0000 0000 0000 0000 0000 000
1	1111 1111	0000 0000 0000 0000 0000 000

Ter uma representação do infinito é útil, pois dá condições ao programador de decidir se tratará o infinito como um erro ou se prosseguirá com a execução do programa.

4.3.3 Situações Inválidas (NaN)

Uma sequência de operações matemáticas pode resultar em situações não previstas na aritmética envolvendo números reais. Por exemplo, o resultado de $\sqrt{-1}$ não pertence ao conjunto dos números reais. Divisões por zero ou por infinito também geram situações indefinidas. Em computação, tais situações são conhecidas como *NaN* – *Not a Number* (não é um número).

Quando todos os bits do campo expoente são iguais a 1 e o campo fração é diferente de $000\dots000$, a notação IEEE 754 convencionou que algum tipo de NaN está sendo representado.

0	1111 1111	$xxx\dots xxx \neq 000\dots000$
1	1111 1111	$xxx\dots xxx \neq 000\dots000$

4.3.4 Números Desnormalizados

A regra geral adotada pelo padrão IEEE 754 é representar um número ponto flutuante em sua forma normalizada. Ou seja, implicitamente considera-se que a parte inteira do número representado é 1. Contudo, note que essa abordagem cria dois problemas:



Figura 19: Formato de 32 bits considerando somente números normalizados.
Fonte: [26].

1. O zero não pode ser representado de forma normalizada. Para contornar esse problema, convencionou-se que, quando todos os bits dos campos expoente e fração forem preenchidos por zero, então estaremos representando o valor zero, e não $1,0 \times 2^{-127}$.
2. Já que 2^{-127} não é mais representado, então o menor número representável em precisão simples (32 bits) é 2^{-126} . Pela Propriedade 4.3, sabemos que é possível representar 2^{23} números no intervalo de 2^{-126} a 2^{-125} . No entanto, não haveria nenhum número representado no intervalo de 0 a 2^{-126} , conforme ilustra a Figura 19.

Usando apenas a notação normalizada, não há garantia de $x - y = 0 \Leftrightarrow x = y$, com $x, y \in \mathbb{F}$. Ou seja, um sistema de ponto flutuante \mathbb{F} normalizado não garante que a diferença entre dois números $x, y \in \mathbb{F}$ é zero se e somente se eles forem iguais.

Vejam os um exemplo. Na Figura 19, o menor número positivo normalizado representável é $y = 2^{-126}$, que corresponde à sequência 0 00000001 0000...0000. Assim, o próximo menor número representável é $x = 2^{-126} \times (1 + 2^{-23})$, que corresponde ao padrão 0 00000001 0000...0001. Portanto:

$$\begin{aligned} x - y &= 2^{-126} \times (1 + 2^{-23}) - 2^{-126} \\ &= \cancel{2^{-126}} + 2^{-126} \times 2^{-23} - \cancel{2^{-126}} \\ &= 2^{-126} \times 2^{-23} \\ &= 2^{-149} \end{aligned}$$

Ora, como o menor número representável é $y = 2^{-126}$, então a diferença $x - y = 2^{-149}$ não pode ser representada (*underflow*). Por conta disso, se usarmos a notação normalizada, essa diferença seria arredondada para zero, embora isso não seja verdadeiro no conjunto \mathbb{R} .

Para contornar os problemas apontados, o padrão IEEE 754 determina o uso da notação *desnormalizada*. Ela é indicada quando todos os bits do campo expoente são 0 e a fração é diferente de zero:

0	0000 0000	xxx...xxx \neq 000...000
1	0000 0000	xxx...xxx \neq 000...000

Na notação desnormalizada de 32 bits, convencionou-se que a parte inteira implícita vale 0, e não 1, e que o expoente vale -126 . Com isso, o menor número positivo representável com essa precisão tem o seguinte padrão de bits:

0	0000 0000	0000 0000 0000 0000 0000 001
---	-----------	------------------------------

$$\text{Parte inteira: } 0,\underbrace{000\dots001}_{23 \text{ bits}} = 2^{-23}$$

$$\text{Expoente: } 2^{-126}$$

$$\text{Total: } 2^{-23} \times 2^{-126} = 2^{-149}$$

Logo, o menor valor representável na forma desnormalizada com 32 bits é justamente a menor diferença possível entre dois elementos quaisquer $x, y \in \mathbb{F}$.

Exemplo 4.11. Qual a sequência de bits correspondente a 2^{-130} na notação IEEE 754 com precisão de 32 bits?

Formatando 2^{-130} de acordo com a eq. 8, temos:

$$2^{-130} = 1,0 \times 2^{-130} \quad (9)$$

No entanto, sabemos que o menor expoente representável com 32 bits de precisão é -126 . Logo, reescrevemos a eq. 9 em função de 2^{-126} :

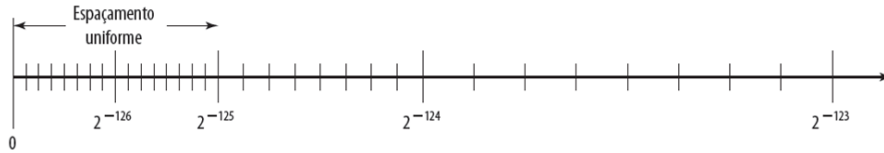
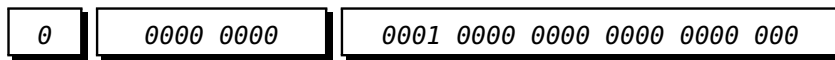


Figura 20: Formato de 32 bits com números desnormalizados. Fonte: [26].

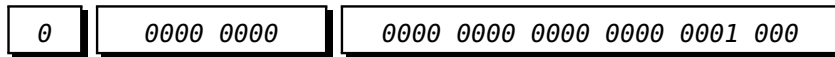
$$\begin{aligned} 2^{-130} &= 1,0 \times 2^{-4} \times 2^{-126} \\ &= 0,0001 \times 2^{-126} \end{aligned}$$

Portanto, o número 2^{-130} será representado da seguinte maneira na memória de um computador, segundo a notação IEEE 754 de 32 bits:

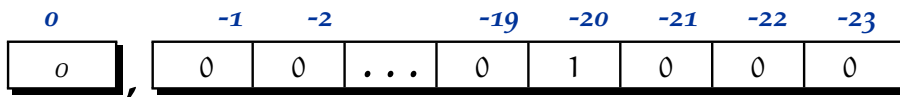


□

Exemplo 4.12. Qual o valor correspondente a sequência de bits abaixo, segundo a notação IEEE 754 de 32 bits?



Como o campo expoente é zero, mas o campo fração é diferente de zero, sabemos que se trata de um formato desnormalizado da notação IEEE 754. Portanto, sabemos que o expoente é -126 . Falta determinar a parte fracionária:



$$\begin{aligned} (N)_{10} &= (0 + 2^{-20}) \times 2^{-126} \\ (N)_{10} &= 2^{-146} \end{aligned}$$

□

4.4 PROPRIEDADES

Nesta seção, comentaremos algumas consequências decorrentes da notação IEEE 754. Nossa análise enfocará o formato de precisão simples (32 bits), mas pode ser facilmente estendida para os formatos de precisão dupla (64 bits) e precisão estendida (80 bits).

Propriedade 4.1 (Maior número representável). *O maior número max representável na notação IEEE 754 de precisão simples é $(2 - 2^{-23}) \times 2^{+127}$*

O maior número ponto flutuante representável na padrão IEEE 754 será aquele correspondente ao maior expoente e maior fração. A maior fração é dada por uma sequência de 23 dígitos 1. O maior expoente, que é expresso em notação com excesso, seria a sequência 11111111. Contudo, ela está reservada para representar o $\pm\infty$ e NaNs. Logo, o maior número representável no padrão IEEE 754 será dado, em módulo, por:

0	1111 1110	1111 1111 1111 1111 1111 111
---	-----------	------------------------------

Essa sequência corresponde ao seguinte número em base decimal:

$$\text{EXPOENTE: } (11111110)_2 = 254 \implies 254 - 127 = +127$$

FRAÇÃO: Em aritmética de base 2, temos:

$$(1, \underbrace{111 \dots 111}_{23\text{bits}})_2 = (10, 0)_2 - (0, \underbrace{000 \dots 001}_{23\text{bits}})_2$$

que na base decimal equivale a $(2 - 2^{-23})$.

Propriedade 4.2 (Menor número normalizado representável). *O menor número min representável (em módulo) na notação IEEE 754 de precisão simples é 2^{-126} .*

De modo semelhante, o menor número representável na padrão IEEE 754 será aquele correspondente ao menor expoente e menor fração. A menor fração é dada por uma sequência de 23 dígitos 0. O menor expoente, que é expresso em notação com excesso, seria a sequência 00000000. Contudo, ela está reservada para representar o zero e os números desnormalizados. Logo, o menor número normalizado representável no padrão IEEE 754 será dado, em módulo, por:

0	0000 0001	0000 0000 0000 0000 0000 000
---	-----------	------------------------------

Essa sequência corresponde ao seguinte valor, em base decimal:

$$\text{EXPOENTE: } 1 - 127 = -126$$

FRAÇÃO: 1,0.

Propriedade 4.3 (Família de significandos de mesmo expoente). *É possível representar 2^{23} valores no intervalo entre dois expoentes consecutivos: 2^x e 2^{x+1} , $-126 \leq x \leq +126$.*

Entre dois expoentes consecutivos, 2^x e 2^{x+1} , o significando varia de 1,00...00 a 1,11...11. Como a parte fracionária tem 23 bits de tamanho, isso nos dá 2^{23} valores intermediários.

Como consequência da Propriedade 4.3, podemos afirmar que os números ponto flutuante não são uniformemente distribuídos ao longo da reta numérica, tal como os números inteiros em notação complemento de 2. A Figura 21 ilustra esse fato.

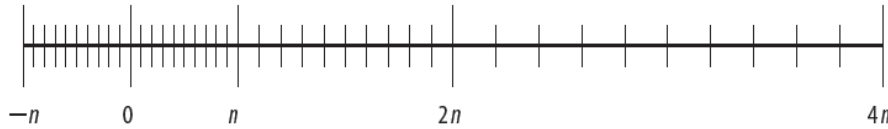


Figura 21: Densidade de distribuição de números ponto flutuante ao longo da reta numérica. Fonte: [26].

4.5 OVERFLOW E UNDERFLOW

Quando representamos números ponto flutuante em um espaço limitado de memória, um velho problema continua existindo: muitos números são tão grandes que exigem uma quantidade de bits maior que aquele espaço oferece. Esse é o problema do *overflow*.

Porém, a representação de números ponto flutuante traz um outro inconveniente: alguns números são tão pequenos, em módulo, que também precisam ser representados por uma quantidade de bits maior que o espaço de memória disponível. É o caso de frações muito pequenas, menores que o menor valor representável (Propriedade 4.2). Esse problema é conhecido como *underflow*.

A Figura 22 esboça o significado visual de *overflow* e *underflow*. *Overflow* ocorre quando um resultado é menor que $-\max$ ou maior que $+\max$, sendo \max o maior número representável em uma notação ponto flutuante (Propriedade 4.1). Por outro lado, *underflow* ocorre quando um resultado se encontra no intervalo compreendido entre $-\min$ e $+\min$, sendo \min o menor número representável (em módulo) em uma notação ponto flutuante (Propriedade 4.2).

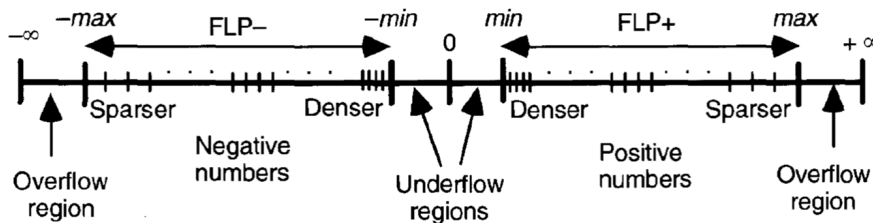


Figura 22: Subintervalos e regiões especiais na representação ponto flutuante. Fonte: [21].

Note que todos os problemas de representação em ponto flutuante não se resumem às faixas de *overflow* e *underflow*. Mesmo no interior da faixa de números representáveis, há uma infinidade de valores que não podem ser representados. Por exemplo, $16 + 2^{-23}$ não pode ser representado na notação IEEE 754 de precisão simples.

Para realizar essa adição, as parcelas devem ser representadas com o mesmo expoente. Ou seja, a vírgula de uma das parcelas deve ser deslocada para que seu expoente seja alinhado ao da outra parcela. Poderíamos deslocar a vírgula da parcela maior à direita. Porém, não haveria espaço em memória para guardar os algarismos deslocados em direção à parte inteira do valor, pois o espaço de representação é utilizado para guardar a parte fracionária do significando.

Portanto, devemos deslocar a vírgula da parcela menor à esquerda. Assim, seu expoente será aumentado e seu significando diminuído, em compensação:

$$\begin{aligned}
 16 + 2^{-23} &= 1,0 \cdot 2^4 + 1,0 \cdot 2^{-23} \\
 &= 1,0 \cdot 2^4 + \underbrace{0,00 \dots 0001}_{27 \text{ casas}} \cdot 2^4 \\
 &= \underbrace{1,00 \dots 0001}_{27 \text{ casas}} \cdot 2^4 \text{ (truncamento)} \\
 &= 1,0 \cdot 2^4
 \end{aligned}$$

Em precisão simples, o campo fração tem apenas 23 casas binárias de tamanho. Dessa forma, os quatro bits adicionais serão descartados, de modo que o resultado de $16 + 2^{-23}$ será 16.



Não use ponto flutuante quando números inteiros são suficientes

Se você precisa computar valores exatos, represente os dados como inteiros e somente converta-os para a escala correta durante a exibição. Por exemplo, em aplicações financeiras, ninguém gostaria de arcar com o prejuízo que uma série de arredondamentos pode provocar. Assim, todos os cálculos devem ser efetuados em centavos, e só no final podem ser divididos por 100 para serem exibidos em reais.

4.6 RESUMO DO CAPÍTULO

Tabela 6: Resumo dos principais parâmetros associados aos formatos IEEE 754.

Parâmetro	Precisão simples	Precisão dupla	Precisão estendida	Geral
Tamanho	32 bits	64 bits	80 bits	n bits
Largura do expoente	8 bits	11 bits	15 bits	k bits
Precisão (em bits)	24 bits	53 bits	64 bits	p bits
Precisão (em dígitos)	7	16	19	$\lfloor (p-1) \times \log_{10} 2 \rfloor + 1$
Largura da fração	23 bits	52 bits	63 bits	p - 1 bits
Excesso (base 2)	127	1023	16383	$2^{k-1} - 1$
Maior expoente, base 2 (emax)	127	1023	16383	$2^{k-1} - 1$
Menor expoente, base 2 (emin)	-126	-1022	-16382	$-2^{k-1} + 2$
Maior/Menor expoente (base 10)	± 38	± 308	± 4932	$\lfloor e_{\max} \times \log_{10} 2 \rfloor$

EXERCÍCIOS DO CAPÍTULO 4

4.1 Explique por que as seguintes afirmações estão incorretas:

- Considerando que $(5)_{10}$ corresponde a $(0101)_2$, então $(1,5)_{10}$ é igual a $(1,0101)_2$.
- Sabemos que $(11)_2 < (13)_{10}$. Logo, deslocando-se a vírgula à esquerda, podemos afirmar que $(1,1)_2 < (1,3)_{10}$.
- Da mesma forma que $(20)_{16} > (20)_{10}$, podemos dizer que $(0,2)_{16} > (0,2)_{10}$.
- Os números $(1,5)_{10}$ e $(1,5)_{16}$ representam os mesmos valores, pois a base hexadecimal é a mesma coisa que a decimal, exceto pelos números A, B, C, D, E e F.
- O valor de $(1,5)_{16}$ na base decimal é 24, pois $1,5 \times 16 = 24$.

4.2 Qual a vantagem de se usar a notação com excesso para representar o expoente de um número de ponto flutuante?

4.3 Considere o formato de ponto flutuante IEEE 754 de precisão simples. Argumente sobre as seguintes questões.

- Os números com parte fracionária deixam de ser representados pela notação a partir de que expoente?
- A partir de que expoente deixam-se de se representar números inteiros consecutivos?

- c) Quantos números podem ser representados no formato desnormalizado?
- 4.4 Determine a representação binária no padrão IEEE 754 em precisão simples para os seguintes números:
- | | |
|------------------|--------------------|
| a) $(20)_{10}$ | d) $(-0,125)_{10}$ |
| b) $(20,5)_{10}$ | e) $(-5/6)_{10}$ |
| c) $(0,1)_{10}$ | f) $(-2)_{10}$ |
- 4.5 Considere um sistema de representação de números ponto flutuante, com 8 bits: três para o campo de expoente e quatro para a fração. Responda às perguntas abaixo:
- Qual o maior valor representável, desconsiderando uma representação especial para o infinito?
 - Qual o menor valor representável (em módulo), desconsiderando uma representação especial para o zero?
 - Desenhe uma reta contendo todos os valores representáveis entre 0,5 e 3.
 - Por que a quantidade de números representados entre 1 e 2 é a mesma que entre 4 e 8, embora este último intervalo seja quatro vezes maior?
- 4.6 [11] Arrume, em ordem crescente, cada uma das seguintes triplas de números:
- $\{(1,1)_2, (1,4)_{10}, (1,5)_{16}\}$
 - $\{(1,5)_8, (1,5)_{16}, (1,5)_{10}\}$
- 4.7 Quanto bits são necessários, no mínimo, para representar os seguintes valores?
- O número de Avogadro $(6,02 \times 10^{23})$?
 - A fração 0,0000000005?
- 4.8 [29, ex. 1.22] Suponha que deseja representar a fração decimal 0,6 em binário, usando uma palavra de 4 bits.
- Encontre qual a fração binária que fornece o valor mais aproximado.
 - Qual o erro percentual se utilizarmos essa representação?
 - Repita o problema considerando uma palavra binário de 8 bits.
- 4.9 [29, ex. 1.23] Considere um número decimal $x = 0,3141$. Qual é a menor palavra binária que pode ser utilizada para representar esse valor com um erro de arredondamento menor do que 0,3%?

- 4.10 Considere as três sequências de bits abaixo. Arrume-as em ordem crescente, interpretando-as segundo as notações numéricas indicadas abaixo. Não é necessário converter os padrões em notação decimal.

```

1010 1101 1010 0000 0000 0000 0000 0010
0001 0111 0001 0000 0000 0000 0000 0100
0110 1011 0111 0000 0000 0000 0000 1000

```

- a) complemento de 2.
 b) Excesso de $2^{31} - 1$.
 c) Ponto flutuante (IEEE 754) com precisão simples.
- 4.11 [18, ex. 2.18] Para cada uma das seguintes sequências de bits em notação IEEE 754 de precisão simples, mostre o valor numérico em termos de um significando e um expoente na base 2. Por exemplo: $(1.11)_2 \times 2^5$.

- a) 0 10000011 011000000000000000000000
 b) 1 10000000 000000000000000000000000
 c) 1 00000000 000000000000000000000000
 d) 1 11111111 000000000000000000000000
 e) 0 11111111 110100000000000000000000
 f) 0 00000001 100100000000000000000000
 g) 0 00000011 011010000000000000000000

- 4.12 Considere o seguinte número ponto flutuante de precisão dupla: BFDC 0000 0000 0000.

- a) Qual o sinal desse número?
 b) Qual o expoente (base decimal), da base 2 implícita na notação IEEE 754?
 c) Qual o significando?
 d) Qual o valor desse número na base decimal?
- 4.13 [18, ex. 2.20] Utilizando o formato IEEE 754 de precisão simples, mostre o valor (em binário e em decimal) de:
- a) O maior número positivo representável (obs: ∞ não é um número).
 b) O menor número positivo normalizado.
 c) O menor número positivo em formato desnormalizado.
 d) O menor espaçamento entre números normalizados.
 e) O maior espaçamento entre números normalizados.

f) A quantidade de números normalizados representáveis (incluindo o 0; note que ∞ e NaN não são números).

4.14 [18, ex. 2.21] Dois programadores escreveram geradores de números aleatórios para números de ponto flutuante normalizados usando o mesmo método. O gerador do Programador A cria números aleatórios no intervalo fechado de 0 a $1/2$, e o gerador do programador B cria números aleatórios no intervalo fechado de $1/2$ a 1. O gerador do programador B funciona corretamente, mas o do Programador A produz uma distribuição de números aleatórios deslocada. Qual deve ser o provável problema com a abordagem do programador A?

4.15 [4, ex. 2.92] Por volta do ano 250 a.C., o matemático grego Arquimedes provou que $223/71 < \pi < 22/7$. Se ele tivesse acesso a um computador e à biblioteca `<math.h>`, ele teria sido capaz de determinar que a aproximação em ponto flutuante de precisão simples para o π tem a representação hexa de `0x40490FDB`. Naturalmente, trata-se de uma aproximação, uma vez que π é irracional.

- Qual a fração binária representada por esse valor ponto flutuante?
- Qual a representação binária de $22/7$?
- A partir de que posição (em relação ao ponto binário) essas duas aproximações divergem uma da outra?

4.16 Considere as seguintes representações de números ponto flutuante IEEE 754 com precisão simples:

$x = 0101\ 1111\ 1011\ 1110\ 0100\ 0000\ 0000\ 0000$

$y = 0011\ 1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000$

$z = 1101\ 1111\ 1011\ 1110\ 0100\ 0000\ 0000\ 0000$

Execute as seguintes operações, mostrando todos os cálculos:

- $x + y$
- (resultado do item a) + z
- Por que esse resultado não é comutativo, ou seja, igual a $x + z + y$?

A LINGUAGEM C OFERECE amplo suporte para manipular bits de dados numéricos. No entanto, poucos são os livros que fornecem detalhes de como fazer isso. Neste capítulo, apresentamos as principais operações de leitura e escrita de dados numéricos em diversas notações: não sinalizada, complemento de 2 e ponto flutuante. Também explicamos como cada um desses tipos de dado numérico pode ser manipulado por meio de operações específicas.

5.1 TIPOS NUMÉRICOS EM C

No contexto da programação, uma variável designa um local da memória onde fica armazenado um valor manipulável pelo programa. A Linguagem C exige que as variáveis sejam declaradas *antes* de seu uso, ou seja, o programador deve informar, no início do código, o nome da variável (*identificador*) e o tipo de valor que pode ser armazenado nela (*tipo de dados*).

O tipo determina as seguintes características de uma variável [23]:

NATUREZA: é a espécie de dado representado. Por exemplo, podemos ter um caractere, um número inteiro, um número real ou um tipo personalizado.

TAMANHO: é o espaço (em bytes) necessário para armazenar os valores do tipo.

REPRESENTAÇÃO: é a forma (conjunto de regras) como os bits armazenados devem ser interpretados. Por exemplo, podemos ter complemento de 2, formato IEEE 754, ASCII, entre outros.

FAIXA DE REPRESENTAÇÃO: é o intervalo de valores válidos para o tipo. Por exemplo, 0 a 255, ou -128 a $+127$.

No contexto de linguagens de programação, o **tipo** indica as características de dados que uma variável pode assumir: número inteiro, número real, caracteres, valores lógicos

Código 1: Exemplo de declaração de variáveis em linguagem C.

```
1 ...  
2     int    x1;  
3     float  x2;  
4     char   x3;  
5 ...
```

O Código 1 apresenta um trecho de um programa C que declara três variáveis distintas: x1, x2, x3. Cada uma delas ocupa uma região diferente na memória do computador. Por exemplo:

- A variável x1 pode ser usada para representar números inteiros (natureza), de 32 bits (tamanho), segundo a notação complemento de 2 (representação), dentro do intervalo que vai de -2^{31} a $+2^{31} - 1$ (faixa de representação).
- A variável x2 pode ser usada para representar números reais, de 32 bits, segundo a notação IEEE 754, dentro do intervalo que vai de $1,17549 \times 10^{-38}$ a $3,40282 \times 10^{+38}$, em módulo.
- A variável x3 pode ser usada para representar caracteres, de 8 bits, segundo o formato ASCII, dentro do intervalo de códigos que vão de -128 a $+127$, que são mapeados para um caractere correspondente.

5.1.1 Tipos Inteiros

A Tabela 7 exibe os tipos inteiros básicos suportados pela Linguagem C. Nela, também são indicados o tamanho e a faixa de valores representados por variáveis assim declaradas. A palavra-chave `unsigned` indica que os valores representados são inteiros sem sinal (não-negativos). Quando o termo `unsigned` é omitido, os valores representados são inteiros sinalizados, ou seja, em notação Complemento de 2.

Na prática, o número de bits alocados para representar um tipo dependerá de dois elementos: o compilador e a arquitetura da máquina onde o código-fonte C é compilado. Por exemplo, o compilador gcc adota o tamanho de 32 ou de 64 bits para o tipo `long int` dependendo se a máquina-alvo for de 32 ou 64 bits, respectivamente. Outro compilador poderá adotar outra convenção.

O Código 9 imprime o tamanho (em bytes) do espaço em memória ocupado pelo tipo `int`. Essa informação é dada pelo resultado da função `sizeof()`, tendo o tipo `int` como argumento. A função `printf()`

Tabela 7: Tamanho e faixa de valores dos tipos inteiros.

Tipo	Tamanho (bits)	Mínimo	Máximo
<code>char</code>	8	-128	127
<code>short int</code>	16	-32.768	32.767
<code>int</code>	32	-2.147.483.648	2.147.483.647
<code>long int</code>	32 ou 64	?	?
<code>long long int</code>	64	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
<code>unsigned char</code>	8	0	255
<code>unsigned short int</code>	16	0	65.535
<code>unsigned int</code>	32	0	4.294.967.295
<code>unsigned long int</code>	32 ou 64	0	?
<code>unsigned long long int</code>	64	0	18.446.744.073.709.551.615

é usada para imprimir esse resultado na tela. Você pode modificar o Código 9 para encontrar o espaço ocupado por outros tipos de variável. Basta trocar a expressão “int” por outro tipo desejado, inclusive os tipos de ponto flutuante, que serão abordados a seguir.

Código 2: Modelo de código para encontrar o tamanho (em bytes) do espaço em memória ocupado pelos tipos da Linguagem C.

```

1 #include <stdio.h>
2 int main() {
3     printf("Espaco em memoria: %ld bytes \n", sizeof(int));
4
5     return 0;
6 }
```

5.1.2 Tipos Ponto Flutuante

A Tabela 8 exibe os tipos ponto flutuante básicos suportados pela Linguagem C. Nela, também são indicados o tamanho e a faixa de valores representados por variáveis assim declaradas, em módulo. A palavra-chave `unsigned` não se aplica para os tipos ponto flutuante.

Nesta seção, enfocamos o *espaço* de memória usado para armazenar as variáveis. Porém, muitas vezes é necessário *interpretar* os valores armazenados em memória, a fim de que sejam exibidos em um dispositivo de saída, tal como um monitor, ou um arquivo em disco. Esse processo é tratado na seção a seguir.

Tabela 8: Tamanho e faixa de valores dos tipos ponto flutuante.

Tipo	Tamanho (bits)	Mínimo	Máximo
float	32	$1,175494 \times 10^{-38}$	$3,402823 \times 10^{+38}$
double	64	$2,225074 \times 10^{-308}$	$1,797693 \times 10^{+308}$
long double	128	$3,362103 \times 10^{-4932}$	$1,189731 \times 10^{+4932}$

5.2 ENTRADA E SAÍDA DE DADOS NUMÉRICOS EM C

Linguagens procedurais como C funcionam a partir da seguinte premissa: dados são inseridos como entrada, armazenados em memória, manipulados pelo programa, e entregues a um dispositivo de saída. Na linguagem C, usamos a função `scanf()` para ler a entrada de dados inseridos via teclado, e a função `printf()` para produzir a saída de dados, que são escritos na tela.

Ao executar um programa C, os dados numéricos ficam armazenados na memória do computador, onde são identificados pelo nome da variável. Como indica a Figura 23, os acessos de escrita e de leitura são sempre realizados a partir dessa região de memória.

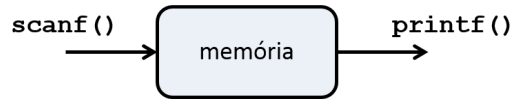


Figura 23: As funções `scanf()` e `printf()` acessam a memória do computador identificada pela variável contida em seus argumentos.

Basicamente, as funções `scanf()` e `printf()` requerem dois argumentos para ler e exibir, respectivamente, dados numéricos:

String (série, tira, cadeia) é uma sequência conectada de caracteres tratados como um só item de dados [25].

1. Nome da(s) variável(is) de entrada ou saída; e
2. Uma *string* contendo especificações de como converter os dados guardados na memória para serem exibidos na tela, ou de como converter os dados digitados via teclado para serem armazenados em memória.

Para utilizar essas duas funções em um programa C, você deve incluir a biblioteca `stdio.h` no preâmbulo do seu código-fonte. O Código 3 apresenta um exemplo simples de como ler um número inteiro positivo digitado via teclado (linha 7), armazená-lo na variável `x`, e escrever o valor de `x` na tela do computador (linha 10).

Código 3: Exemplo básico de entrada e saída de dados em linguagem C.

```

1  #include <stdio.h>
2  int main() {
3      unsigned x;          /* Inteiro NAO sinalizado */
4
5      /* Leitura de valores */
6      printf("Digite o valor de x: ");
7      scanf("%u", &x);
8
9      /* Impressao de valores */
10     printf("\n Valor de x: %u", x);
11
12     return 0;
13 }
  
```

Vamos entender como o Código 3 é executado. Se você já conhece a linguagem C, pode pular a explicação a seguir e ir direto à Seção 5.3.

Todo programa C é formado por um conjunto de funções. A função principal, chamada `main`, é aquela por onde o programa inicia a execução. Toda função em C deve indicar quais os seus argumentos (entrada) e qual a sua saída.

No Código 3, a função `main` não espera receber nenhum argumento como entrada, por isso seus parênteses estão vazios. A saída (resultado) da função principal deve ser 0, a fim de que o sistema operacional entenda que a execução do programa foi bem sucedida. A devolução do valor 0 é indicada pelo comando `return` na linha 12.

Como a função principal deve retornar um valor inteiro, seu tipo é indicado como `int`, no início da linha 2.

Na Seção 5.3 a seguir, detalhamos como a função `printf()` funciona.

5.3 A FUNÇÃO printf()

A função `printf()` instrui o computador a imprimir uma mensagem de texto (*string* de caracteres) no terminal de vídeo. Por exemplo, o comando `printf("Fiat lux")` imprimirá na tela a mensagem Fiat lux, sem aspas.

A *string* de caracteres contida entre as aspas é composta de pelo menos um dos seguintes elementos:

CARACTERES ORDINÁRIOS: são impressos na tela da maneira como são informados entre aspas no argumento da função, tal como no exemplo anterior.

DIRETIVAS DE FORMATAÇÃO: descrevem como os argumentos seguintes da função `printf()` devem ser interpretados, para converter os dados numéricos armazenados na memória do computador em caracteres impressos na tela. Por exemplo, na linha 10 do Código 3, `"%u"` é uma diretiva que instrui o programa para interpretar os bits da região de memória denotada pela variável `x` como um número inteiro sem sinal (*unsigned*).

Diretiva é uma instrução não executada diretamente, usada para controlar o compilador ou interpretador de uma linguagem de programação.

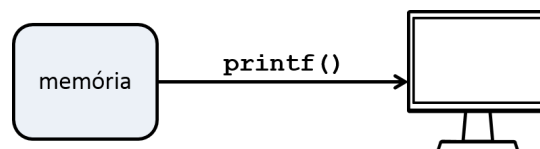


Figura 24: A função `printf()` lê dados da memória sem modificá-los, apresentando-os na tela sob algum formato especificado.

Como ilustrado na Figura 24, a função `printf()` transforma sequências de bits armazenados na memória em caracteres exibidos na tela. Assim, sequências de bits sempre estarão guardadas na memória no formato binário. O modo de exibi-las na tela *não* as modifica de maneira alguma. Analogamente, quando as luzes de uma sala de apagam, os objetos ali contidos não somem, eles apenas deixam de serem vistos. Ou ainda, se uma luz azul for projetada sobre objetos amarelos, a tinta que os reveste não passará a ser verde, mas sim a cor com que são *exibidos* é que passará a ser verde.

As diretivas de formatação são compostas por um grupo de caracteres, que começa com o símbolo `%`, seguido obrigatoriamente por um caractere *especificador de formato*. Este último define o tipo e o formato como um valor armazenado em memória será *interpretado* para ser

apresentado como uma sequência de caracteres na tela. Entre o símbolo % e o *especificador de formato*, podemos ter, opcionalmente, quatro outros itens, como mostra a Figura 25:

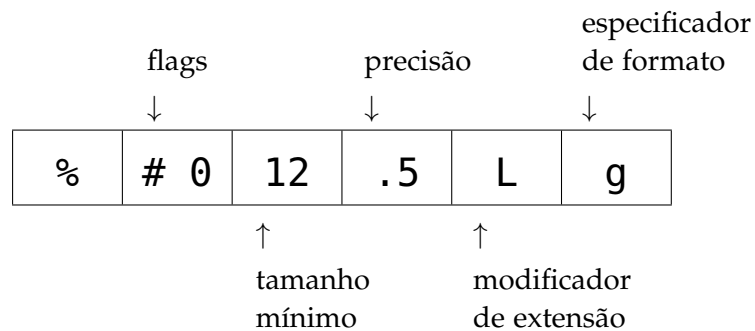


Figura 25: Sintaxe de uma diretiva de formatação.

MODIFICADOR DE EXTENSÃO: altera a quantidade de bits de como a posição de memória será interpretada, sem alterar os dados armazenados em memória. Está detalhado na Seção 5.3.2.

PRECISÃO: indica o número de casas decimais a serem impressas na tela, no caso de dados numéricos em ponto flutuante. Está detalhado na Seção 5.3.5.

TAMANHO MÍNIMO: define a quantidade mínima de caracteres a serem impressos na tela para representar o valor passado como argumento. Está detalhado na Seção 5.3.4.

FLAGS: são opcionais e mais de uma é permitida. Alteram alinhamento, impressão de sinalização e preenchimento de zeros à esquerda. Estão detalhadas na Seção 5.3.3.

5.3.1 Especificadores de formato

Os especificadores de formato são caracteres que definem o tipo e o formato como um valor armazenado em memória será *interpretado* para ser apresentado como uma sequência de caracteres na tela. Ou seja, a linguagem C permite que uma variável seja declarada segundo um tipo, mas tenha seu valor impresso segundo um outro tipo especificado. No entanto, essa operação *não altera* o valor, nem o tipo da variável impressa, mas apenas a forma de como o padrão de bits é interpretado e apresentado na tela.



Atenção: Especificadores de formato não alteram os valores armazenados em memória

Especificadores de formato (%d, %u, %x) servem para interpretar o dado durante a impressão. Os bits armazenados em memória continuam os mesmos.

A escolha do especificador de formato depende do tipo do argumento da função printf(). Por exemplo, a linguagem C não permite que variáveis ponto flutuante sejam impressas como inteiras, ou vice-versa.

Tabela 9: Especificadores de formato para números inteiros. Fontes: [23, 16].

Especificador	Efeito
d	O argumento é impresso como um inteiro na base decimal.
u	O argumento é impresso como um inteiro não sinalizado na base decimal.
x, X	O argumento é impresso em base hexadecimal. Se o especificador for x (minúsculo), os dígitos a–f serão impressos em caracteres minúsculos. Se for X, então serão impressos em caracteres maiúsculos.
o	O argumento é impresso em base octal.

FORMATAÇÕES INTEIRAS. Os especificadores de formato inteiro relacionados na Tabela 9 são utilizados para imprimir valores inteiros (int, unsigned e variantes). Eles não podem ser utilizados para imprimir valores ponto flutuante (float ou double). Inicialmente, o argumento (variável ou valor) correspondente é convertido para a notação informada pela diretiva. Em seguida, o resultado é formatado em uma sequência de caracteres (*string*) a ser impressa na tela.

Código 4: Modelo de código C sobre os principais especificadores de formato de tipos inteiros, abordados no Exemplo 5.1.

```

1 #include <stdio.h>
2 int main() {
3     int arg = 155;
4
5     printf("\n Valor de arg (diretiva %o): %o", arg);
6
7     return 0;
8 }
```

Exemplo 5.1. A Tabela 10 apresenta a saída produzida pelas diretivas de formatação de números inteiros em programas C semelhantes ao Código 4.

Por exemplo, a última linha da tabela mostra que, para o valor 155 armazenado em uma variável chamada *arg* do tipo *int*, o comando `printf("%o", arg)` produz a saída 233 na tela. Para observar o efeito dos demais exemplos da tabela, substitua a diretiva desejada na linha 5, ou o valor atribuído à variável *arg* na linha 3.

Tabela 10: Exemplos de diretivas de formatação para números inteiros e seus respectivos resultados.

Diretiva	Argumento	Impressão	Argumento	Impressão
%d	155	155	-155	-155
%u	155	155	-155	4294967141
%x	155	9b	-155	ffffff65
%X	155	9B	-155	FFFFFF65
%o	155	233	-155	3777777545

□

No Exemplo 5.1, o valor 155 é impresso como 155 tanto na notação decimal sinalizada (%d) como na não sinalizada (%u), pois seu sinal é positivo. Nesse caso, tanto a notação binária simples como a complemento de dois usam a mesma convenção para representar números menores ou iguais a $2^{n-1} - 1$, sendo *n* o tamanho da palavra.

A maioria dos compiladores C atuais utilizam 32 bits para representar o tipo *int*. Para ter certeza disso, mande imprimir o resultado da função `sizeof(int)`. De qualquer maneira, $155 \ll 2^{31} - 1$. Assim, as diretivas %d e %u exibiram o mesmo resultado na tela. Por fim, usando o método das divisões sucessivas, é fácil verificar que 155 corresponde a $(9B)_{16}$ ou a $(233)_8$.

Por outro lado, quando o argumento é um valor negativo, -155, a notação decimal sinalizada (%d) imprime o valor como o conhecemos na aritmética cotidiana. Já a diretiva não sinalizada (%u) faz o seguinte. O valor -155 corresponde a 1111 1111 1111 1111 1111 1111 0110 0101 em complemento de dois. Essa sequência, quando convertida em hexadecimal, resulta em FFFF FF65. Quando convertida à base octal, resulta em 37 777 777 545, conforme mostra a Tabela 10.



Atenção: Diretivas de formatação não alocam espaço em memória

Diretivas de formatação – tais como %X e %ld – servem para interpretar o dado inserido via teclado, quando usadas na função `scanf()`, ou o dado armazenado em memória, quando usadas na função `printf()`. A alocação de espaço em memória acontece durante a declaração de variáveis.

FORMATAÇÕES DE PONTO FLUTUANTE. Os especificadores de formato da Tabela 11 são utilizados para exibir valores reais em notação IEEE 754, seja de precisão simples (float) ou de precisão dupla (double). Primeiro, o argumento da função printf() – que pode ser uma variável ou valor literal – é convertido para a notação indicada pela diretiva. Em seguida, o resultado é formatado em uma sequência de caracteres que será impressa na tela.

Tabela 11: Especificadores de formato para ponto flutuante. Fontes: [23, 16].

Especificador	Efeito
f	O valor é impresso no formato <i>decimal</i> : sinal, parte inteira, ponto, parte fracionária.
e	O valor é impresso no formato <i>científico</i> : sinal, parte inteira, ponto, parte fracionária, caractere e (indicando expoente de uma potência de 10), valor do expoente em base decimal.
a, A	O valor é impresso no formato <i>hexadecimal</i> , pela notação IEEE 754: sinal, 0x, 1. (parte inteira), parte fracionária em hexadecimal, caractere p (indicando expoente de uma potência de 2), valor do expoente em base decimal (sem deslocamento). A diretiva A imprime os caracteres especiais da base hexadecimal em letras maiúsculas, enquanto que a diretiva a os imprime em minúsculas.
g, G	O argumento deve ser double. O valor é impresso no formato da diretiva f ou e, dependendo da precisão. Mais detalhes em [23, 16]. Os zeros à direita da parte fracionária são removidos.

Código 5: Modelo de código C sobre os principais especificadores de formato de tipos ponto flutuante, abordados no Exemplo 5.2.

```

1 #include <stdio.h>
2 int main() {
3     float arg = 13.75;
4
5     printf("\n Valor de arg (diretiva %e): %e", arg);
6
7     return 0;
8 }
```

Exemplo 5.2. A Tabela 12 a seguir apresenta a saída produzida pelas diretivas de formatação de números ponto flutuante em programas C semelhantes ao Código 5.

Por exemplo, a terceira linha da tabela mostra que, para um valor 13.75 armazenado em uma variável chamada *arg* do tipo *float* ou *double*, o comando `printf("%e", arg)` produz a saída `1.375000e+001` na tela. Para observar o efeito dos demais exemplos listados na tabela, substitua a diretiva desejada na linha 5, ou o valor atribuído à variável *arg* na linha 3.

Tabela 12: Exemplos de diretivas de formatação para números ponto flutuante e seus respectivos resultados.

Diretiva	Argumento	Impressão	Argumento	Impressão
%f	13.75	13.750000	0.875	0.875000
%e	13.75	1.375000e+001	0.875	8.750000e-001
%A	13.75	0X1.B80000P+3	0.875	0X1.C00000P-1

□

No Exemplo 5.2, o valor $(13,75)_{10}$ é impresso dessa mesma maneira pela diretiva %f. Já a diretiva %e faz com que esse valor seja apresentado na notação científica, provocando o deslocamento da vírgula uma casa decimal para a esquerda, o que é compensado no expoente (e+001). Em binário simples, o valor $(13,75)_{10}$ equivale a $(1101,11)_2$. Deslocando-se a vírgula em três casas binárias para a esquerda, temos $(1,1011\ 1000)_2 \times 2^3$. Assim, a diretiva %A substitui os bits da parte fracionária por símbolos hexadecimal (.B800) e indica que houve um deslocamento de três casas binárias para a esquerda (P+3).

As diretivas %f e %A fazem com que os números reais sejam exibidos na tela com precisão simples. Logo, a parte fracionária é representada com 23 bits. Lembre-se que, na conversão binário–hexadecimal, cada quatro dígitos binários são convertidos em um dígito hexadecimal. Por conta disso, um 24^{o} bit é anexado aos 23 existentes, de modo a imprimir 6 casas hexadecimais correspondentes à parte fracionária do número. Esses 24 bits espelham os dados tais quais estão armazenados em memória.

Já na conversão binário–decimal, os 23 bits da parte fracionária equivalem a $\lfloor 23 \times \log_{10} 2 \rfloor = 6$ casas decimais. Por isso, a diretiva %f imprime, por padrão, seis casas decimais na tela.

Em contraste, no lado direito da Tabela 12, para representar o valor $(0,875)_{10}$ em notação científica, deslocamos a vírgula para a direita, de modo que a diretiva %e imprime na tela um expoente negativo (e-001). Em formato binário simples, o valor $(0,875)_{10}$ equivale a $(0,111)_2$. Deslocando-se a vírgula em uma casa binária para a direita, temos $(1,1100)_2 \times 2^{-1}$. Assim, a diretiva %A substitui os bits da parte fracionária por símbolos hexadecimais (.C000) e indica que houve um deslocamento de uma casa binária para a direita (P-1).

Portanto, a diretiva %A é mais indicada quando se deseja observar como os bits correspondentes a um número ponto flutuante estão armazenados em memória segundo o padrão IEEE-754. Já as diretivas

%f e %e são mais indicadas quando desejamos interpretar os valores ponto flutuante na base decimal.

5.3.2 Modificadores de extensão

Os modificadores de extensão alteram a quantidade de bits com que o argumento será formatado. Note, mais uma vez, que a variável não é modificada, pois a função printf() apenas faz a *leitura* dos valores armazenados em memória para apresentá-los na tela. O que se modifica é a *apresentação* na tela da variável informada como argumento. Assim, uma variável do tipo int, que normalmente tem 32 bits, pode ser apresentada na tela como se tivesse apenas 8 bits (%hhd) ou 64 bits (%lld) de tamanho.

O efeito dos modificadores de extensão depende muito do compilador utilizado. As explicações aqui contidas referem-se ao gcc 4.7.3. Você pode obter efeitos diferentes se utilizar outros compiladores. Por conta disso, para evitar resultados não planejados, recomendamos que você utilize modificadores de tipo correspondentes ao tamanho do tipo do argumento impresso. Além disso, quando as combinações de modificadores de extensão e especificadores de tipo não casam com o tipo do argumento, alguns compiladores exibem mensagens de alarme (*warnings*) durante a compilação.

A Tabela 13 apresenta os modificadores de extensão mais comuns associados à impressão de variáveis numéricas. A coluna “Efeito” diz que a combinação “Modificador” e “Especificador” limitará o tamanho do valor impresso à mesma quantidade de bits correspondente ao tipo informado. Por exemplo, a diretiva %hhx especifica que o valor impresso na tela terá o mesmo tamanho de bits ocupado por uma variável unsigned char, mesmo que o argumento informado à função printf() seja de outro tipo.

Código 6: Modelo de código C sobre modificadores de extensão de tipos inteiros, abordados no Exemplo 5.3.

```

1 #include <stdio.h>
2 int main() {
3     int arg = 300;
4
5     /* Impressao de valores */
6     printf("\n Valor de arg (diretiva %hhd): %hhd", arg);
7
8     return 0;
9 }
```

Exemplo 5.3. A Tabela 14 apresenta a saída produzida pelas diretivas de modificadores de extensão em programas C semelhantes ao Código 6. Cada

Tabela 13: Modificadores de extensão. Fontes: [23, 16].

Modificador	Especificador	Efeito
hh	d	char
	o, u, x, X	unsigned char
h	d	short int
	o, u, x, X	unsigned short int
l	d	long int
	o, u, x, X	unsigned long int
	f, e, a, A	sem efeito
ll	d	long long int
	o, u, x, X	unsigned long long int
L	f, e, a, A	long double

linha da tabela apresenta um exemplo de diretiva (primeira coluna), dois exemplos de valores aos quais a diretiva é aplicada (300 e 70000), e resultados correspondentes impressos na tela.

Por exemplo, a terceira linha da tabela mostra que, para um valor 300 armazenado em uma variável chamada `arg` do tipo `int`, o comando `printf("%hhhd", arg)` produz a saída 44 na tela. Para observar o efeito dos demais exemplos listados na tabela, substitua a diretiva desejada na linha 6, ou o valor atribuído à variável `arg` na linha 3.

Tabela 14: Exemplos de diretivas de modificação de extensão de extensão.

Diretiva	Argumento	Impressão	Argumento	Impressão
%d	300	300	70000	70000
%hhhd	300	44	70000	112
%hd	300	300	70000	4464
%ld	300	300	70000	70000
%lld	300	-4625656705227685588	70000	-4634787943137865360

□

5.3.3 Flags

As *flags* são caracteres especiais que alteram a formatação padrão definida pelos especificadores de formato. Mais de uma flag pode ser utilizada ao mesmo tempo. A Tabela 15 fornece uma lista das flags mais comuns.

5.3.4 Tamanho mínimo

O especificador de tamanho mínimo define a quantidade mínima de caracteres que serão usados para representar o valor formatado. Se

Tabela 15: Flags da função printf. Fontes: [23, 16].

Flag	Efeito
-	Alinha valor à esquerda, com impressão de espaços à direita, se forem necessários para completar o tamanho mínimo do campo. Sem essa <i>flag</i> , alinha-se o valor à direita, por padrão.
+	No caso de números sinalizados, força a exibição do sinal do valor, seja positivo (+) ou negativo (-). Sem essa <i>flag</i> , exibe-se, por padrão, o sinal apenas dos números negativos.
#	Torna explícita a notação utilizadas. Para números na base octal, inicia o valor exibido com um 0. Para números na base hexadecimal, inicia o valor exibido com um 0x (ou 0X).
0	Imprime zeros em vez de espaços à esquerda do número, a fim de que ele ocupe todo o espaço indicado pelo <i>tamanho mínimo</i> .

o valor formatado possui uma quantidade *menor* de caracteres que o tamanho mínimo especificado, então espaços e zeros são usados para atingir o tamanho especificado. Se possui uma quantidade *maior* de caracteres, então ele é impresso com todos os seus caracteres, sem truncamento [23, 16]. Ou seja, o especificador é ignorado nesse último caso.

5.3.5 Precisão

Para as diretivas de formatação de ponto flutuante (f, e, a), o especificador de precisão determina o número de dígitos impressos na tela após o ponto decimal. Se a quantidade de dígitos da parte fracionária for *maior* que a precisão especificada, então o valor impresso é arredondado; se for *menor*, então o valor impresso é estendido com zeros à direita. Caso a precisão não seja especificada, seu valor padrão é de 6 casas decimais, conforme discutido na Seção 5.3.1.

Código 7: Código C para exemplificar o uso de flags aliadas a tipos inteiros, conforme abordado no Exemplo 5.4.

```

1 #include <stdio.h>
2 int main() {
3     short int arg = 3530;
4
5     printf("\n Valor de arg (diretiva %+3hd): %+3hd", arg);
6
7     return 0;
8 }
```

Exemplo 5.4. A Tabela 16 a seguir apresenta a saída produzida por diversas flags de formatação para valores inteiros armazenados em variável do tipo `short int`, em programas C semelhantes ao Código 7. Cada linha

mostra um exemplo de diretiva, dois valores aos quais ela é aplicada (3530 e -3530), e os respectivos resultados que são exibidos na tela.

Por exemplo, a terceira linha da tabela mostra que, para um valor 3530 armazenado em uma variável chamada *arg* do tipo *short int*, o comando `printf("%+3hd", arg)` produz a saída +3530 na tela. Para observar o efeito dos demais exemplos listados na tabela, substitua a diretiva desejada na linha 5, ou o valor atribuído à variável *arg* na linha 3.

Tabela 16: Exemplos de diretivas de formatação para valores *short int*.

Diretiva	Argumento	Impressão	Argumento	Impressão
"%hd"	3530	3530	-3530	-3530
"%+3hd"	3530	+3530	-3530	-3530
"%+8hd"	3530	+3530	-3530	-3530
"%+08hd"	3530	+0003530	-3530	-0003530
"%-+8hd"	3530	+3530	-3530	-3530
"%.2hd"	3530	3530	-3530	-3530
"%.6hd"	3530	003530	-3530	-003530
"%8.6hd"	3530	003530	-3530	-003530

□

Código 8: Código C para exemplificar o uso de flags aliadas a tipos de precisão dupla, conforme abordado no Exemplo 5.5.

```

1 #include <stdio.h>
2 int main() {
3     long double arg = 450.3;
4
5     printf("\n Valor de arg (diretiva %13Lf): %13Lf", arg);
6
7     return 0;
8 }
```

Exemplo 5.5. A Tabela 17 a seguir apresenta a saída produzida pelas diretivas de formatação para valores armazenados em variável do tipo *long double* (precisão dupla), em programas C semelhantes ao Código 8. Cada linha mostra um exemplo de diretiva, dois valores aos quais ela é aplicada (450.3 e -450.3), e os respectivos resultados que são exibidos na tela.

Por exemplo, a quarta linha da tabela mostra que, para um valor -450.3 armazenado em uma variável chamada *arg* do tipo *long double*, o comando `printf("%13Lf", arg)` produz a saída -450.300000 alinhado à esquerda da tela. Para observar o efeito dos demais exemplos listados na tabela, substitua a diretiva desejada na linha 5, ou o valor atribuído à variável *arg* na linha 3.

□

Tabela 17: Exemplos de diretivas de formatação para valores long double.

Diretiva	Argumento	Impressão	Argumento	Impressão
"%Lf"	450.3	450.300000	-450.3	-450.300000
"%+4Lf"	450.3	+450.300000	-450.3	-450.300000
"%13Lf"	450.3	450.300000	-450.3	-450.300000
"%13Le"	450.3	4.503000e+02	-450.3	-4.503000e+02
"%#13Le"	450.3	4.503000e+02	-450.3	-4.503000e+02
"%+-Lf"	450.3	+450.300000	-450.3	-450.300000
"%013Lf"	450.3	000450.300000	-450.3	-00450.300000
"%13.Lf"	450.3	450	-450.3	-450
"%.4Lf"	450.3	450.3000	-450.3	-450.3000
"%13.8Lf"	450.3	450.30000000	-450.3	-450.30000000
"%13.Le"	450.3	5e+02	-450.3	-5e+02
"%#13.Le"	450.3	5.e+02	-450.3	-5.e+02

5.4 OPERAÇÕES LÓGICAS BIT A BIT

Nas operações lógicas bit a bit, o processador alinha os operandos bit a bit, e os bits de cada posição são operados individualmente, sem considerar o valor ou significado da string de bits.

A Tabela 18 descreve as quatro operações lógicas bit a bit suportadas pela Linguagem C. As operações AND, OR e XOR envolvem dois operandos. Nesse caso, os bits de mesma posição de cada operando são processados um de cada vez, sem considerar os resultados obtidos nas posições vizinhas. Por outro lado, a operação NOT é executada sobre apenas um único operando.

Tabela 18: Operadores bit a bit. Fontes: [23].

Operação	Operador	Efeito
AND	&	Cada bit do resultado é igual a 1, se os bits de mesma posição dos operandos forem iguais a 1; ou igual a zero, caso contrário.
OR		Cada bit do resultado é igual a zero, se os bits de mesma posição dos operandos forem iguais a zero; ou igual a 1, caso contrário.
XOR	^	Cada bit do resultado é igual a zero, se os bits de mesma posição dos operandos forem iguais entre si; ou igual a 1, caso contrário.
NOT	~	Troca o valor de cada bit do operando. Corresponde ao complemento de 1.

Código 9: Modelo de código para encontrar o tamanho (em bytes) do espaço em memória ocupado pelos tipos da Linguagem C.

```

1  #include <stdio.h>
2  int main() {
3      short int a = 123; /* 0x7B -> b01111011 */
4      short int b = 20;  /* 0x14 -> b00011110 */
5
6      printf("AND: %X \n", a & b);
7      printf("OR : %X \n", a | b);
8      printf("XOR: %X \n", a ^ b);
9      printf("NOT: %X \n", ~a);
10
11     return 0;
12 }

```

```

AND: 10
OR : 7F
XOR: 6F
NOT: FFFFFFF84

```

5.5 DESLOCAMENTO DE BITS

O deslocamento de bits é uma operação realizada sobre a cadeia de bits de um tipo inteiro. Essa sequência de bits pode ser movida para a esquerda (<<) ou para a direita (>>). Operações aritméticas, como a adição e a divisão, transformam os *valores* operados. Já as operações de deslocamento de bits transformam a *cadeia de bits* [4].

5.5.1 Deslocamento à Esquerda

A operação $a \ll k$ desloca os bits da variável inteira a em k posições para a *esquerda*. Desse modo, os k bits mais significativos de a são *descartados* e os k espaços criados na extremidade direita são preenchidos com zero.

Consequentemente, o número de posições deslocadas k deve ser maior ou igual zero, caso contrário, o compilador emitirá uma mensagem de aviso. Se k for maior que o número de bits de a , o resultado será invariavelmente igual a zero.

Note que o deslocamento à esquerda $a \ll k$ corresponde à multiplicação $a \times 2^k$. Porém, se o produto for maior que o tamanho de a , poderá haver truncamento.

Vejamos algumas situações de exemplo. Em todas elas, a operação de deslocamento funciona da mesma forma. Altera-se apenas a forma de interpretarmos os bits.

DESLOCAMENTO DE UM VALOR NÃO SINALIZADO. Observe o Código 10 e sua saída correspondente. Ele imprime o resultado, em base decimal e binária, do deslocamento à esquerda dos bits de uma variável unsigned char. A função printbin será detalhada na Seção 5.6, por isso não se preocupe como ela funciona por enquanto.

Inicialmente, atribui-se o valor dez (00001010_2) à variável a. Em seguida, por meio do laço for, os bits de a são deslocados de zero a oito posições à esquerda. Isso faz com que, em base decimal, o resultado vá dobrando de valor: 20, 40, 80, e assim por diante.

Porém, quando a cadeia de bits 00001010 é deslocada *cinco* posições à esquerda, o bit 1 mais significativo é descartado. Por isso, em base decimal, o valor obtido (64) não corresponde ao esperado ($320 = 10 \times 2^5$). Outra forma de entender esse resultado é lembrar que o valor 320 não pode ser armazenado em um espaço não sinalizado de apenas 8 bits, o qual representa, no máximo, $255 = 2^8 - 1$.

Código 10: Deslocamento à esquerda de uma variável unsigned char.

```

1 #include <stdio.h>
2 /* Procedimento para imprimir os bits de uma variavel inteira */
3 void printbin(unsigned char a) {
4     /* Imprime cada bit da variavel 'a' */
5     for (unsigned i = 8; i > 0; i--) {
6         printf("%u", a >> 7);
7         a = a << 1;
8     }
9     printf("\n");
10 }
11
12 int main() {
13     unsigned char a = 10;
14     for (unsigned i = 0; i <= 8; i++) {
15         printf("a << %d = %3hhu: ", i, a << i);
16         printbin(a << i);    /* Deslocamento de 'i' posicoes */
17     }
18     return 0;
19 }

```

```

a << 0 = 10: 00001010
a << 1 = 20: 00010100
a << 2 = 40: 00101000
a << 3 = 80: 01010000
a << 4 = 160: 10100000
a << 5 = 64: 01000000
a << 6 = 128: 10000000
a << 7 = 0: 00000000
a << 8 = 0: 00000000

```

DESLOCAMENTO DE UM VALOR SINALIZADO. Agora, observe o Código 11 e sua saída correspondente. Nele, uma variável `char`, sinalizada de 8 bits e com valor inicial de -10 , tem seus bits deslocados de zero a oito posições em direção à esquerda. Isso faz com que o resultado expresso na base decimal vá dobrando de valor: -20 , -40 , -80 , e assim por diante.

Desta vez, quando a cadeia de bits `00001010` é deslocada *quatro* posições à esquerda, o MSB, indicador do sinal de um número em complemento de dois, muda de um para zero. Ou seja, o número perde o sinal negativo. Dessa forma, na base decimal, o valor obtido (96) não corresponde ao esperado ($-160 = 10 \times 2^4$). Outra forma de entender esse resultado é lembrar que o valor -160 não pode ser armazenado em um espaço *sinalizado* de apenas 8 bits, o qual representa, no mínimo, $-128 = -2^7$.

Código 11: Deslocamento à esquerda de uma variável `char` (sinalizada).

```

1  #include <stdio.h>
2  /* Procedimento para imprimir os bits de uma variavel inteira */
3  void printbin(unsigned char a) {
4      /* Imprime cada bit da variavel 'a' */
5      for (unsigned i = 8; i > 0; i--) {
6          printf("%u", a >> 7);
7          a = a << 1;
8      }
9      printf("\n");
10 }
11
12 int main() {
13     unsigned char a = -160;
14     for (unsigned i = 0; i <= 8; i++) {
15         printf("a << %d = %3hhd: ", i, a << i);
16         printbin(a << i);    /* Deslocamento de 'i' posicoes */
17     }
18     return 0;
19 }
```

```

a << 0 = -10: 11110110
a << 1 = -20: 11101100
a << 2 = -40: 11011000
a << 3 = -80: 10110000
a << 4 =  96: 01100000
a << 5 = -64: 11000000
a << 6 = -128: 10000000
a << 7 =   0: 00000000
a << 8 =   0: 00000000
```


5.5.2 Deslocamento à Direita

A operação $a \gg k$ desloca os bits da variável inteira a em k posições para a *direita*. Desse modo, os k bits menos significativos de a são *descartados*. Já os k espaços criados na extremidade esquerda de a podem ser preenchidos de duas maneiras:

- No modo *lógico*, a extremidade esquerda é sempre preenchida com k zeros.
- No modo *aritmético*, a extremidade esquerda é preenchida com k repetições do bit mais significativo (MSB) da variável a original. Essa convenção pode parecer estranha, mas ela *preserva o sinal do operando*, na notação complemento de 2.

O sinal usado pela Linguagem C para o deslocamento à direita (\gg) não distingue o modo lógico do aritmético. O *tipo* da variável deslocada é que determina qual dos dois modos será aplicado:

- Para variáveis *não sinalizadas* (unsigned), aplica-se o modo *lógico*.
- Para variáveis *sinalizadas* (signed), aplica-se o modo *aritmético*.

Note que o deslocamento à direita $a \gg k$ corresponde à divisão inteira $a \div 2^k$. Vejamos algumas situações de exemplo.

DESLOCAMENTO LÓGICO. Observe o Código 12 e sua saída. Nele, uma variável unsigned char, com valor inicial de 160, tem seus bits deslocados de zero a oito posições em direção à *direita*. Isso faz com que o resultado vá caindo pela metade: 80, 40, 20, 10, 5, 2, 1.

Código 12: Deslocamento lógico à direita de uma variável unsigned char.

```

1 #include <stdio.h>
2 /* Procedimento para imprimir os bits de uma variavel inteira */
3 void printbin(unsigned char a) {
4     /* Imprime cada bit da variavel 'a' */
5     for (unsigned i = 8; i > 0; i--) {
6         printf("%u", a >> 7);
7         a = a << 1;
8     }
9     printf("\n");
10 }
11
12 int main() {
13     unsigned char a = 160;
14     for (unsigned i = 0; i <= 8; i++) {
15         printf("a >> %d = %3hu: ", i, a >> i);
16         printbin(a >> i);    /* Deslocamento de 'i' posicoes */
17     }
18     return 0;
19 }

```

Quando a cadeia de bits 10100000 é deslocada *seis* posições à direita, o bit 1 mais à direita é descartado. Mesmo assim, o valor obtido continua correspondendo ao *quociente* inteiro de $160 \div 2^6 = 2$.

Pela saída produzida, percebemos que o deslocamento à direita aplicado foi do tipo lógico, pois zeros foram inseridos nas posições vagas à esquerda apesar da variável *a* ter 1 no MSB. O programa procedeu assim porque a variável *a* foi declarada como `unsigned char`.

```
a » 0 = 160: 10100000
a » 1 =  80: 01010000
a » 2 =  40: 00101000
a » 3 =  20: 00010100
a » 4 =  10: 00001010
a » 5 =   5: 00000101
a » 6 =   2: 00000010
a » 7 =   1: 00000001
a » 8 =   0: 00000000
```

DESLOCAMENTO ARITMÉTICO. Observe o Código 13 e sua saída. Nele, uma variável `char`, sinalizada e com valor inicial de +80, tem seus bits deslocados de zero a oito posições em direção à *direita*. Isso faz também com que o resultado vá caindo pela metade: 40, 20, 10, 5, 2, 1.

Código 13: Deslocamento aritmético à direita de uma variável `char`.

```
1  #include <stdio.h>
2  /* Procedimento para imprimir os bits de uma variavel inteira */
3  void printbin(unsigned char a) {
4      /* Imprime cada bit da variavel 'a' */
5      for (unsigned i = 8; i > 0; i--) {
6          printf("%u", a >> i);
7          a = a << 1;
8      }
9      printf("\n");
10 }
11
12 int main() {
13     char a = 80;
14     for (unsigned i = 0; i <= 8; i++) {
15         printf("a >> %d = %3hhd: ", i, a >> i);
16         printbin(a >> i);    /* Deslocamento de 'i' posicoes */
17     }
18     return 0;
19 }
```

De modo semelhante ao exemplo anterior, quando a cadeia de bits 01010000 é deslocada *cinco* posições à direita, obtemos o quociente inteiro 2, como esperado: $80 \div 2^5 = 80 \div 32 = 2$.

Olhando apenas a saída produzida, não é possível percebermos se o deslocamento à direita aplicado foi no modo lógico ou no aritmético. Como o MSB de $+80 = 01010000_2$ é zero, não sabemos se os bits 0 à esquerda foram inseridos por definição (modo lógico) ou pela repetição do MSB (modo aritmético).

```
a » 0 = 80: 01010000
a » 1 = 40: 00101000
a » 2 = 20: 00010100
a » 3 = 10: 00001010
a » 4 = 5: 00000101
a » 5 = 2: 00000010
a » 6 = 1: 00000001
a » 7 = 0: 00000000
a » 8 = 0: 00000000
```

Por outro lado, se alterarmos o valor inicial da variável *a* para -80 , na linha 13, perceberemos claramente que o Código 13 aplica o deslocamento à direita no modo aritmético, pois o MSB de $-80 = 10110000_2$, que vale um, é replicado para preencher os espaços que vão sendo criados na extremidade esquerda.

Em inglês, o deslocamento aritmético também é conhecido como *sticky shift*, ou seja, “deslocamento grudento” em tradução literal. É como se o MSB fosse um chiclete, que vai esticando à medida em que a sequência de bits é deslocada à direita: $11 \dots 11$ ou $00 \dots 00$.

```
a » 0 = -80: 10110000
a » 1 = -40: 11011000
a » 2 = -20: 11101100
a » 3 = -10: 11110110
a » 4 = -5: 11111011
a » 5 = -3: 11111101
a » 6 = -2: 11111110
a » 7 = -1: 11111111
a » 8 = -1: 11111111
```

5.5.3 Resto da Divisão Inteira

5.6 IMPRESSÃO DE VALORES EM FORMATO BINÁRIO

Na Seção 5.3, vimos que a Linguagem C oferece diretivas de formatação pré-definidas para imprimir valores numéricos nas bases decimal, octal e hexadecimal. No entanto, não oferece uma diretiva para imprimir valores na base binária. Para atingir esse objetivo, temos de escrever um pouco mais de código.

Também vimos que a Linguagem C trata valores inteiros e ponto flutuante de forma diferente. Portanto, teremos que adotar estratégias distintas para imprimir na tela a sequência de bits que representa valores de cada tipo de variável. É o que veremos nas duas seções a seguir.

5.6.1 Impressão de Inteiros em Binário

Antes de qualquer discussão, é importante ter em mente que todos os dados armazenados na memória de um computador estão codificados em binário. A restrição que a Linguagem C impõe para visualizar a sequência de bits não implica que devemos fazer uma nova conversão de bases numéricas. Pelo contrário, devemos contornar essa restrição lançando mão dos recursos que a Linguagem C oferece.

Existem duas maneiras básicas para atribuir um valor a uma variável em C:

1. A variável pode receber um valor numérico estático, chamado de *valor imediato*. Por exemplo: `a = 666`.
2. A variável pode receber um valor digitado via teclado. Por exemplo: `scanf("%u", &a)`.

Bem, na verdade, há ainda uma terceira maneira: uma variável pode receber o valor de outra variável, mas esse caso pode ser recursivamente reduzido a um dos dois casos básicos anteriores.

No primeiro caso, o valor imediato será transformado em uma sequência de bits *durante a compilação*. Por exemplo, durante a atribuição `int x = 19`, o número 19 será convertido para 10011. Essa sequência será gravada no código executável do programa e, quando este for executado, ela será carregada em memória.

No segundo caso, valores lidos pela função `scanf()` também serão transformados em uma sequência de bits, mas desta vez durante o *tempo de execução* do programa. Por exemplo, se o usuário pressionar as teclas 1 e 9 durante a execução do comando `scanf("%d", &x)`, o código ASCII dessas teclas será fornecido pelo Sistema Operacional à função `scanf()`. Esta, por sua vez, converterá esses caracteres em uma sequência de bits segundo a notação complemento de 2 de 32 bits (considerando que a variável `x` seja do tipo `int`).

Em qualquer um dos casos, o valor da variável `x` será representado em memória por uma sequência de bits. Nosso problema agora é justamente imprimir esses bits sem que uma nova conversão seja feita, pois ela já aconteceu, ou durante a compilação, ou durante a execução.

Já que a Linguagem C não permite que todos os bits da sequência sejam impressos de forma direta, temos de escrever um código que imprima um bit de cada vez. Portanto, o código exige um laço de

repetição que seja executado tantas vezes quanto for o tamanho da variável, em bits. Por exemplo, se estivermos imprimindo uma variável do tipo `int`, o laço deverá ser repetido 32 vezes.

E o que deve ser executado em cada iteração? Isso depende da ordem como a saída do programa é apresentada na tela, que normalmente é da esquerda para a direita. Dessa forma, nosso algoritmo deve imprimir primeiramente o bit mais significativo da sequência desejada. Como fazemos isso? Uma forma simples é deslocar as $n - 1$ posições menos significativas da sequência de n bits para a direita, de modo que apenas o bit mais significativo reste nela. Daí, ele já pode ser impresso, pois as casas à esquerda serão completadas com zero (certificando-se que o tipo seja `unsigned`).

Para as demais iterações, devemos cuidar para que o segundo bit mais significativo seja deslocado em uma posição à esquerda, para que fique na posição mais significativa e o procedimento anterior possa ser repetido.

O Código 14 apresenta uma função que implementa o algoritmo descrito, considerando variáveis de 8 bits de tamanho (`char`). Tente modificá-la para impressão de tipos mais longos, com o `short int`, `int` ou o `long long int`.

Código 14: Função que imprime a sequência de bits de uma variável `char`.

```
1 void printbin_shift(unsigned char a) {  
2     unsigned int i;      /* Contador */  
3  
4     /* Imprime cada bit da variavel 'a' */  
5     for (i = 8; i > 0; i--) {  
6         printf("%u", a >> 7);  
7         a = a << 1;  
8     }  
9 }
```

A função apresentada pelo Código 14 tem um problema. O valor original da variável é perdido à medida que os deslocamentos são aplicados sobre ela. O Código 15 resolve esse problema utilizando uma máscara de bits. A máscara é definida de tal modo que, a cada iteração, apenas um bit é extraído da variável para impressão na tela. Entre uma iteração e a seguinte, somente a máscara – e não a variável de interesse – é deslocada para extrair o próximo bit.

Código 15: Impressão da sequência de bits de uma variável `short` usando mascaramento.

```
1 void printbin_mask(unsigned short int a) {  
2     unsigned int i;          /* Contador */  
3     unsigned int mask = 0x8000; /* Mascara */  
4  
5     /* Imprime cada bit da variavel 'a' */  
6     printf("\nBinario:\n");
```

```

7   for (i = 16; i > 0; i--) {
8       printf("%u", (a & mask) >> (i-1));
9       mask = mask >> 1;
10  }
11 }

```

5.6.2 Impressão da Representação Hexadecimal de Ponto Flutuante

Como vimos na Seção 5.3, não é possível utilizar a diretiva %X para imprimirmos a representação hexadecimal de variáveis do tipo float ou double em C. O Código 16 contorna esse impedimento fazendo com que uma mesma área de memória seja interpretado ora como float, ora com unsigned. Isso se dá por meio da union definida nas linhas 4 a 7. Todos os componentes de uma union são armazenados em um mesmo local (endereço) de memória. Quando desejamos escrever um valor real em base decimal, usamos o componente float. Já quando queremos ler a representação hexadecimal, por meio da diretiva %X, utilizamos o componente unsigned.

Código 16: Impressão da representação hexadecimal de uma variável float usando union.

```

1  #include <stdio.h>
2  /* Permite multiplas interpretacoes
3  a partir de um mesmo espaco de memoria */
4  typedef union {
5      float    f;
6      unsigned u;
7  } tipoFloat;
8
9  int main() {
10     tipoFloat numero;
11
12     printf("Digite um numero real em base decimal: ");
13     scanf("%f", &numero.f);
14
15     printf("Numero real          : %f\n", numero.f);
16     printf("Representacao binaria: %u\n", numero.u);
17     printf("\n\n");
18
19     return 0;
20 }

```

O Código 17 apresenta uma solução mais elegante, pois usa ponteiros em C. Ponteiros são variáveis que armazenam endereços de memória. O Código 17 define uma variável do tipo float chamada var. Como sabemos, ela não pode ser impressa usando-se a diretiva de formatação %X. Para contornar esse impedimento, o Código 17 executa os seguintes passos na linha 8:

1. Toma-se o endereço (e não valor) da variável `var` por meio do operador `&`.
2. Converte-se o ponteiro de `float *` para `unsigned *`, pois este tipo suporta o uso da diretiva de formatação `%X`. Esse procedimento não altera o valor do endereço de memória. O que muda é o *tipo* de variável apontada.
3. Extrai-se o valor apontado pelo endereço convertido, por meio do operador `*`, à esquerda da palavra `unsigned`. Essa operação é conhecida como *derreferenciamento*.

Código 17: Impressão da representação hexadecimal de uma variável `float` usando ponteiro.

```

1 #include <stdio.h>
2 int main() {
3     float var;
4
5     printf("Digite o valor da variavel float: ");
6     scanf("%f", &var);
7
8     printf("Representacao binaria: %08X\n", *((unsigned *)&var));
9
10    return 0;
11 }
```

Por fim, apresentamos o Código 18, uma modificação do Código 16. Ele imprime os três campos da representação IEEE 754: sinal, expoente e fração. Para isso, definimos nas linhas 3 a 7 uma estrutura chamada `bitsIEEE754`, constituída por três componentes. Estes, por sua vez, têm seu tamanho em bits determinado pelo valor à direita do sinal de dois pontos.

Estruturas são úteis para agrupar valores em um só registro, que pode ser manipulado como um só tipo ao longo do código. Assim, na linha 14, acrescentamos mais uma interpretação da *union* `tipoFloat`. Podemos enxergar esse espaço de memória de três maneiras:

- como um número real em base decimal;
- como um bloco único de caracteres hexadecimal correspondentes aos 32 bits da notação IEEE 754; ou
- como um conjunto de 32 bits dividido nos três campos da notação IEEE 754.

Código 18: Disseca a representação de bits de uma variável `float` usando campos de bits e `union`.

```

1 #include <stdio.h>
```

```

2  /* Define campos de bits de um float */
3  typedef struct {
4      unsigned fracao: 23;
5      unsigned expoente: 8;
6      unsigned sinal: 1;
7  } tipoCamposIEEE754;
8
9  /* Permite multiplas interpretacoes
10 a partir de um mesmo espaco de memoria */
11 typedef union {
12     float f;
13     unsigned u;
14     tipoCamposIEEE754 bits;
15 } tipoFloat;
16
17 int main() {
18     tipoFloat numero;
19
20     printf("Digite um numero real em base decimal: ");
21     scanf("%f", &numero.f);
22
23     printf("\n\nCAMPOS DA NOTACAO IEEE 754:\n");
24     printf("-----\n");
25     printf("Numero      : %f\n", numero.f);
26     printf("Sinal        : %u\n", numero.bits.sinal);
27     printf("Expoente (b10): %d\n", numero.bits.expoente - 127);
28     printf("Fracao (hexa): %06X\n", numero.bits.fracao << 1);
29     printf("\n\n");
30
31     return 0;
32 }

```

As linhas 26, 27 e 28 imprimem os valores dos campos da sequência de bits da notação IEEE 754. Eis alguns detalhes importantes:

SINAL: este é o campo mais simples, pois assume um dos seguintes valores: *zero*, se o valor representado for positivo; ou *um*, se for negativo.

EXPOENTE: este campo está representado em notação deslocada (vide Seção 3.4, p. 62). Para encontrar seu valor em complemento de 2, basta subtrair o deslocamento, que é de $2^{n-1} - 1$, sendo n o tamanho do campo em bits. Para o tipo float, o expoente tem 8 bits. Logo, devemos subtrair 127 antes de imprimir o valor. Usando a diretiva %d, o valor do expoente é convertido da notação complemento de 2 para a base decimal.

FRAÇÃO: este campo tem 23 bits de tamanho na notação IEEE 754 de precisão simples. Por outro lado, o processador manipula informação organizada em bytes, e não em bits. Assim, ao operar um valor inteiro sem sinal de 23 bits de tamanho, como definido na linha 4, o processador inclui um zero à esquerda, para

completar 3 bytes (=24 bits). Sabemos que a inclusão de zeros à esquerda de valores inteiros não altera o seu valor. Contudo, o valor aqui manipulado não um inteiro. Ele representa uma *fração*. E em frações, zeros acrescentados à esquerda modificam o valor representado. Por conta disso, temos de deslocar a sequência de bits uma posição à esquerda, antes de imprimi-la, a fim de eliminar esse zero inconvenientemente acrescentado.

5.7 OVERFLOW E UNDERFLOW

5.7.1 Overflow de Inteiros em Linguagem C

5.7.2 Overflow e Underflow de Ponto Flutuante em Linguagem C

Código 19: Comparação entre a impressão de uma variável `double` e de outra `float` que armazenam valores iguais.

```

1 #include <stdio.h>
2 int main() {
3     float x = 0.66666666666666666666; /* precisao 32 bits */
4     double y = 0.66666666666666666666; /* precisao 64 bits */
5
6     printf("x = %.20f\n", x);
7     printf("y = %.20lf\n", y);
8
9     return 0;
10 }
```

Quando executado, o Código 19 produz a seguinte saída na tela:

```

x = 0.666666668653488159000
y = 0.6666666666666666663000
```

No Código 19, a dízima $0,\overline{6}$, truncada em vinte casas decimais, foi atribuída duas variáveis: `x`, do tipo `float`, e `y`, do tipo `double`. Sabemos que o tipo `float` ocupa 32 bits na memória, dos quais 23 são utilizados para representar a parte fracionária, o que correspondem a $\lfloor 23 \cdot \log_2 10 \rfloor = 6$ casas decimais. De modo semelhante, o tipo `double` ocupa 64 bits na memória, dos quais 52 bits representam a parte fracionária (15 casas decimais).

Se o tipo `float` possibilita representar valores com até 6 casas decimais de precisão, então de onde vieram os valores exibidos a partir da sétima casa decimal (0,000000686534882)?

Antes de tudo, esse valor *não é aleatório*! Se fosse aleatório, cada execução do programa forneceria um valor diferente. Também *não é lixo de memória*! Quando declaramos variáveis em um programa C, um espaço de memória é reservado para armazenar o valor delas.

Entre o momento da declaração de uma variável e o momento da atribuição de valor a essa variável, de fato, o valor da variável será aquele que já estava gravado naquela região de memória durante um processamento anterior que utilizava essa mesma região. Contudo, as linhas 3 e 4 do Código 19 atribuem um valor às variáveis *x* e *y*. Portanto, não podemos justificar os dígitos excedentes como sendo “lixo de memória”.

Pegue uma calculadora e verifique o valor de 2^{-23} :

$$2^{-23} = 0,000\ 000\ 119\ 209\ 289\ 550\ 781\ 250$$

Note que a representação do número 2^{-23} na base decimal ocupa 23 casas decimais. Algo semelhante acontece com outras potências de 2 próximas:

$$2^{-20} = 0,000\ 000\ 953\ 674\ 316\ 406\ 250\ 000\ 000$$

$$2^{-21} = 0,000\ 000\ 476\ 837\ 158\ 203\ 125\ 000\ 000$$

$$2^{-22} = 0,000\ 000\ 238\ 418\ 579\ 101\ 562\ 500\ 000$$

$$2^{-23} = 0,000\ 000\ 119\ 209\ 289\ 550\ 781\ 250\ 000$$

$$2^{-24} = 0,000\ 000\ 059\ 604\ 644\ 775\ 390\ 625\ 000$$

$$2^{-25} = 0,000\ 000\ 029\ 802\ 322\ 387\ 695\ 312\ 500$$

$$2^{-26} = 0,000\ 000\ 014\ 901\ 161\ 193\ 847\ 656\ 250$$

Quando convertemos um número fracionário de *n* bits para a base dez, realizamos uma *soma* das potências de 2 correspondentes às *n* casas fracionárias. Embora essa soma nos ofereça precisão de apenas $\lfloor n \cdot \log_2 10 \rfloor$ casas decimais, ela se estende por *n* casas decimais. Porém, o comando da linha 6 diz ao processador para imprimir as casas decimais excedentes além da sexta posição.

Como a variável *x* tem 32 bits de tamanho, o resultado mostrado a partir da sétima casa decimal é justamente o resíduo da soma das vinte e três potências de 2 disponíveis para o campo “significando”. Se desejássemos imprimir, por exemplo, o valor exato da sétima casa decimal, seria necessário estender o campo de 23 bits e, consequentemente, mais potências de 2 entrariam nessa soma, tornando o resultado da conversão mais próximo do valor real.

Já a variável *y* tem 64 bits de tamanho; por isso, sua impressão exibiu as 15 casas decimais com a precisão esperada. Por outro lado, por que não foi exibido um resíduo das somas de potências de 2 a partir da 16ª casa decimal, de modo semelhante ao que ocorreu na impressão da variável *x*?

Devemos lembrar que memória de um sistema computacional é um recurso finito. Portanto, existe algum limite de espaço a partir do qual não é possível armazenar os valores das potências de 2 que se estendem por muitas casas decimais.

Além disso, a função `printf()` promove os seus argumentos do tipo `float` para `double` no momento da impressão¹. É por isso que a soma das potências de 2 pôde ser expressa na saída do programa, mesmo que seu valor esteja além da capacidade de armazenamento de uma variável `float`. Já no caso da variável `double`, embora haja resíduos de soma de potências de 2, não há espaço para guardá-las. Desse modo, acrescentaram-se zeros à direita para satisfazer a formatação solicitada (`%.20`).

5.8 RESUMO DO CAPÍTULO

¹ Para mais detalhes, consultar o parágrafo 6 da seção 6.5.2.2 (*Function calls*) do padrão C18 [14].

EXERCÍCIOS DO CAPÍTULO 5

5.1 Que resultado será impresso na tela pelo seguinte programa? Explique.

```
1 #include <stdio.h>
2 int main() {
3     float a = 90.0625;    // 90 + 1/16
4
5     unsigned u = *(unsigned *) &a;
6     unsigned x = (u >> 23) & 0xFF;
7     unsigned y = (u & 0x007FFFFF) << 1;
8
9     printf("x = 0x%08X \n", x);
10    printf("y = 0x%08X \n", y);
11
12    return 0;
13 }
```

5.2 Que resultado será impresso na tela pelo seguinte programa? Explique.

```
1 #include <stdio.h>
2 int main() {
3     union {
4         float x;
5         unsigned y;
6     } numero;
7
8     numero.x = 41.4375;
9
10    printf("%#08X", numero.y);
11
12    return 0;
13 }
```

NÚMEROS BINÁRIOS NA LINGUAGEM PYTHON

CAPÍTULO EM CONSTRUÇÃO

PYTHON É UMA LINGUAGEM DE PROGRAMAÇÃO que vem ganhando popularidade nos últimos anos. Este capítulo apresenta algumas funções e métodos que manipulam números inteiros e ponto flutuante em Python.

6.1 NÚMEROS INTEIROS EM PYTHON

6.2 NÚMEROS PONTO FLUTUANTE EM PYTHON

```
print((1.0).hex())
```

```
0x1.0000000000000p+0
```

Retorna a representação de um número ponto flutuante como uma string hexadecimal.

6.3 RESUMO DO CAPÍTULO

EXERCÍCIOS DO CAPÍTULO 6

OUTRAS FORMAS DE REPRESENTAÇÃO

PALAVRAS DE BITS podem ser usadas para representar não apenas números, mas também caracteres. Adicionalmente, formas de representação numéricas podem facilitar a realização de operações lógicas e aritméticas no nível do processador, mas podem trazer complicações quando utilizadas no interfaceamento com o mundo físico. Por fim, os erros de transmissão e de armazenamento inerentes ao mundo físico podem exigir representações de dados mais robustas a tais problemas.

Neste capítulo, veremos...

7.1 BINARY CODED DECIMAL (BCD)

7.2 CÓDIGO GRAY

7.3 REPRESENTAÇÃO DE CARACTERES

7.3.1 *ASCII*

7.3.2 *Unicode*

7.4 CÓDIGOS DE DETECÇÃO E CORREÇÃO DE ERROS

7.4.1 *Código de Barras*

7.4.2 *QR Code*

7.5 RESUMO DO CAPÍTULO

Tabela 19: Correspondência entre algumas convenções BCD e seu valor em base decimal.

Dígito	BCD Natural	Excesso de 3	Aiken
0	0000	0011	0000
1	0001	0100	0001
2	0010	0101	0010
3	0011	0110	0011
4	0100	0111	0100
5	0101	1000	1011
6	0110	1001	1100
7	0111	1010	1101
8	1000	1011	1110
9	1001	1100	1111

EXERCÍCIOS DO CAPÍTULO 7

Tabela 20: Código Gray de 4 bits.

Decimal	Binário	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Tabela 21: Exemplos de códigos de correção de erro na transmissão de alguns valores numéricos.

Decimal	Paridade ímpar	Paridade par	2-de-5	Biquinário
0	00001	00000	01001	0100001
1	00010	00011	00011	0100010
2	00100	00101	00101	0100100
3	00111	00110	00110	0101000
4	01000	01001	01010	0110000
5	01011	01010	01100	1000001
6	01101	01100	10001	1000010
7	01110	01111	10010	1000100
8	10000	10001	10100	1001000
9	10011	10010	11000	1010000

Tabela 22: Código de barras UPC (*Universal Product Code*).

Dígito	Lado Esquerdo	Lado Direito
0	0001101	1110010
1	0011001	1100110
2	0010011	1101100
3	0111101	1000010
4	0100011	1011100
5	0110001	1001110
6	0101111	1010000
7	0111011	1000100
8	0110111	1001000
9	0001011	1110100

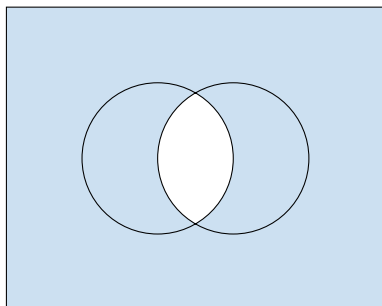
Parte II

CIRCUITOS COMBINATÓRIOS

Nesta parte veremos os Circuitos Combinatórios.

ÁLGEBRA BOOLEANA

8.1 RELAÇÃO ENTRE ÁLGEBRA BOOLEANA E CONJUNTOS



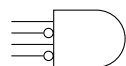
8.2 PROPRIEDADES

8.3 RESUMO DO CAPÍTULO

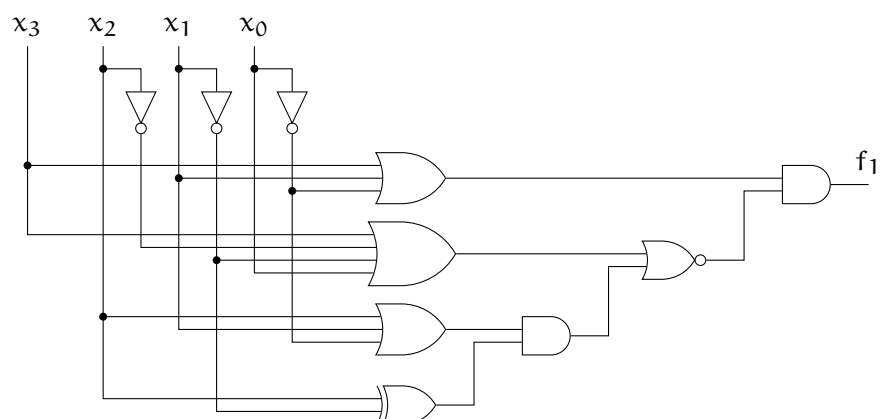
EXERCÍCIOS DO CAPÍTULO 8

PORTAS LÓGICAS

9.1 PORTA NOT



9.2 PORTA AND



9.3 PORTA OR

9.4 PORTA XOR

9.5 PORTA NAND

9.6 RESUMO DO CAPÍTULO

EXERCÍCIOS DO CAPÍTULO 9

MAPAS DE KARNAUGH

10.1 RESUMO DO CAPÍTULO

EXERCÍCIOS DO CAPÍTULO 10

CIRCUITOS COMBINATÓRIOS

11.1 RESUMO DO CAPÍTULO

EXERCÍCIOS DO CAPÍTULO 11

Parte III

CIRCUITOS SEQUENCIAIS

Nesta parte veremos circuitos sequenciais e uma das suas principais aplicações, os circuitos de memória.

CIRCUITOS SEQUENCIAIS

12.1 RESUMO DO CAPÍTULO

EXERCÍCIOS DO CAPÍTULO 12

13.1 RESUMO DO CAPÍTULO

EXERCÍCIOS DO CAPÍTULO 13

JUNTANDO TUDO: O PROCESSADOR

14.1 RESUMO DO CAPÍTULO

EXERCÍCIOS DO CAPÍTULO 14

REFERÊNCIAS BIBLIOGRÁFICAS

- [1] S. P. Bali. *2000 Solved Problems in Digital Electronics (Sigma Series)*. Tata McGraw-Hill, Nova Deli, 2005.
- [2] Alex Bellos. *Alex nos País dos Números: Uma viagem ao mundo maravilhoso da Matemática*. Companhia das Letras, São Paulo, 2011.
- [3] Kenneth J. Breeding. *Digital Design Fundamentals*. Prentice Hall, The Ohio State University, 2 edition, 1992.
- [4] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley, 3 edition, 2016.
- [5] Harold T. Davis. *História da Computação (Tópicos de história da matemática para uso em sala de aula)*, volume 2. Atual Editora, 1992.
- [6] Freeman Dyson. *Infinito em todas as direções: do gene à conquista do universo*. Editora Best Seller, São Paulo, 1988.
- [7] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991. ISSN 0360-0300.
- [8] John R. Gregg. *Ones and Zeros: Understanding Boolean Algebra, Digital Circuits, and the Logic of Sets (Understanding Science & Technology Series)*. Wiley-IEEE Press, New Jersey, 1998.
- [9] Bernard H. Gundlach. *História dos Números e Numerais (Tópicos de história da matemática para uso em sala de aula)*, volume 1. Atual Editora, 1992.
- [10] David Harris and Sarah Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann, 2 edition, 2012.
- [11] Geoffrey L. Herman, Craig Zilles, and Michael C. Loui. How do students misunderstand number representations? *Computer Science Education*, 21(3):289–312, 2011. doi: 10.1080/08993408.2011.611712.
- [12] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Computer Society, 2008. Disponível em <http://www.csee.umbc.edu/~tsimol/CMSC455/IEEE-754-2008.pdf> (Novembro 2016).
- [13] Georges Ifrah. *Os Números*. Globo, 5 edition, 1992.
- [14] ISO/IEC. *ISO/IEC 9899:2018 (C18 Standard)*, 2018.

- [15] Annibal Hetem Jr. *Fundamentos de Informática: Eletrônica Digital*. LTC, Rio de Janeiro, 2010.
- [16] K. N. King. *C Programming: A Modern Approach*. W W Norton Company, 2008.
- [17] Israel Koren. *Computer Arithmetic Algorithms*. A K Peters, Natick, Massachusetts, 2 edition, 2002.
- [18] Miles J. Murdocca and Vincent P. Heuring. *Computer Architecture and Organization: An Integrated Approach*. Wiley, 2007.
- [19] Linda Null and Julia Lobur. *Princípios Básicos de Arquitetura e Organização de Computadores*. Bookman, 2010.
- [20] Behrooz Parhami. *Arquitetura de Computadores: de microprocessadores a supercomputadores*. McGraw-Hill, São Paulo, 2007.
- [21] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, New York, 2 edition, 2010.
- [22] Tales Cleber Pimenta. *Circuitos Digitais: Análise e Síntese Lógica: aplicações em FPGA*. Elsevier, Rio de Janeiro, 2007.
- [23] Francisco A. C. Pinheiro. *Elementos de Programação em C*. Bookman, 2012.
- [24] Charles H. Roth and Larry L. Kinney. *Fundamentals of Logic Design*. Cengage Learning, 7 edition, 2014.
- [25] Márcia Regina Sawaya. *Dicionário de Informática e Internet*. Nobel, 1999.
- [26] William Stallings. *Arquitetura e Organização de Computadores*. Pearson Prentice Hall, São Paulo, 8 edition, 2010.
- [27] Andrew S. Tanenbaum and Todd Austin. *Organização Estruturada de Computadores*. Pearson Prentice Hall, São Paulo, 6 edition, 2013.
- [28] Roger Tokheim. *Fundamentos de Eletrônica Digital: sistemas combinacionais*, volume 1. Bookman, 7 edition, 2013.
- [29] John P. Uyemura. *Sistemas Digitais: Uma Abordagem Integrada*. Pioneira Thomson Learning, São Paulo, 2002.
- [30] Wikipedia, the free encyclopedia. Extended Precision, November 2016. Disponível em https://en.wikipedia.org/wiki/Extended_precision (Novembro 2016).