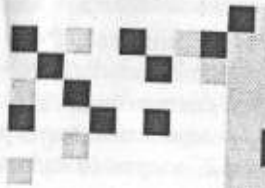


2 REPRESENTAÇÃO DE DADOS

2.1 Introdução

No início da era dos computadores existiam várias concepções erradas. Uma delas era a de que computadores eram nada mais do que uma gigantesca máquina de calcular. Mas mesmo os computadores dessa época podiam fazer muito mais do que isso. Outra era a de que, em contradição com a primeira, os computadores podiam fazer qualquer coisa. Hoje em dia sabemos que existem classes de problemas intratáveis mesmo com o mais poderoso dos computadores. A percepção correta, é claro, está entre essas duas.

Operações de computadores que não são aritméticas são conhecidas nossas: geração de gráficos, audiodigital, e até mesmo a manipulação do mouse. Mas independente do tipo de informação que está sendo manipulada pelo computador, essa informação tem de ser representada por padrões de zeros e uns (também conhecidos como códigos "liga-desliga"). Isso imediatamente chama a atenção para como a informação deve ser descrita ou representada na máquina – essa é a **representação** ou **codificação dos dados**. Imagens gráficas, audiodigital ou cliques de mouse devem ser todos codificados de uma forma sistemática.

Nós podemos pensar na representação decimal da informação como a mais natural porque a conhecemos melhor, mas o uso de códigos liga-desliga para representar informação existiu muitos anos antes dos computadores, na forma de código Morse.

Este capítulo introduz algumas das mais simples e importantes codificações: a codificação de números de ponto fixo com e sem sinal, números reais (chamados de **números de ponto flutuante** no jargão de computadores), e caracteres a serem impressos ou mostrados na tela. Veremos que em todos os casos existem várias maneiras de se codificar certos dados, algumas úteis em algum contexto, outras em outro contexto. Estudaremos também a parte básica de aritmética de computadores com o propósito de entender alguns esquemas de codificação, muito embora os detalhes serão vistos somente no Capítulo 3.

Uma questão é extremamente importante durante o processo de se decidir como representar dados em um computador: o quanto de espaço reservar para

cada valor. Por exemplo, um projetista pode decidir tratar inteiros com 32 bits e implementar uma ALU que implemente operações aritméticas em 32 bits e retorne resultados de 32 bits. Contudo, alguns números são muito grandes para serem representados com 32 bits. Em outros casos os operandos podem ser codificados com 32 bits, mas o resultado da operação não, causando um **overflow** (descrito no Capítulo 3). Portanto, precisamos entender os limites impostos pela precisão e limites dos cálculos numéricos impostos pela representação dos dados. Iremos investigar esses limites nas próximas seções.

2.2 Números de Ponto Fixo

Em um sistema que usa números de ponto fixo cada número tem exatamente o mesmo número de dígitos, e o "ponto" está sempre no mesmo lugar. Exemplos do sistema de números decimais são 0,23, 5,12 e 9,11. Nestes exemplos cada número tem três dígitos e o ponto decimal está localizado a duas posições a partir do lado direito. Exemplos do sistema de números **binários** (onde cada dígito pode ter somente um valor 0 ou 1) são 11,10, 01,10 e 00,11, onde existem quatro dígitos binários e o ponto está no meio dos dígitos. Uma diferença importante entre a maneira que representamos números de ponto fixo no papel e a maneira como os representamos no computador é que no computador o ponto não é armazenado em nenhum lugar, somente supõe-se que estará em certa posição. Pode-se dizer que o ponto binário existe somente na cabeça do programador.

Começaremos a estudar números de ponto fixo investigando o intervalo de representação e precisão de números de ponto fixo no sistema decimal. Depois discutiremos a natureza das bases de números tais como decimal e binária, e como converter entre essas bases. Com esse estudo, investigaremos diversas maneiras de se representar números negativos de ponto fixo, e também operações aritméticas simples que podem ser feitas com eles.

2.2.1 Intervalo de representação e precisão em números de ponto fixo

Uma representação de ponto fixo pode ser caracterizada pelo **intervalo** de representação de números que podem ser expressos (ou seja, a distância entre o menor e o maior número) e a **precisão** (a distância entre dois números adjacentes na seqüência). Para o exemplo citado acima, usando-se três dígitos e o ponto decimal colocado a duas posições da direita, o intervalo de representação vai de 0,00 a 9,99, incluindo os dois pontos extremos, denotado [0,00, 9,99]. A precisão é de 0,01 e o erro é de 0,5 da diferença entre dois números "adjacentes" na seqüência, tais como, por exemplo, 5,01 e 5,02, que têm uma diferença de 0,01. O erro é, portanto, de 0,01/2 = 0,005, ou seja, podemos representar qualquer número dentro do intervalo 0,00 e 9,99 com um erro máximo de 0,005 do seu valor real.

Note como o intervalo de representação e a precisão são conflitantes: com o ponto decimal no extremo direito o intervalo de representação é de [000, 999] e

a precisão é de 1,0. Com o ponto decimal no extremo esquerdo o intervalo é de [0,000, 0,999] e a precisão é de 0,001.

Nos dois casos, existem 10^3 "objetos" decimais diferentes, no intervalo entre 000 e 999 ou 0,000 e 0,999, sendo possível, portanto, representar somente mil valores diferentes, independente de como estabelecemos o intervalo de representação e a precisão.

Não existe necessidade de se começar a representar números em 0. Um número decimal de dois dígitos pode ter um intervalo de representação de [00, 99] ou [-50, 49] ou até mesmo [-99, 0]. A representação de números negativos é vista de forma mais detalhada na Seção 2.2.6.

Intervalo de representação e precisão são pontos importantes em arquitetura de computadores porque ambos são finitos na implementação da arquitetura, mas também são infinitos no mundo real; portanto o usuário precisa estar ciente das limitações de se tentar representar informação externa em formato interno.

2.2.2 A lei associativa da álgebra nem sempre funciona em computadores

Em matemática nós aprendemos a lei associativa da álgebra:

$$a + (b + c) = (a + b) + c$$

Como veremos, essa lei não é válida para números de ponto fixo que têm uma representação finita. Considere uma representação de ponto fixo de um dígito decimal, com o ponto decimal na direita e um intervalo [-9, 9], com $a = 7$, $b = 4$, e $c = -3$. Agora $a + (b + c) = 7 + (4 - 3) = 7 + 1 = 8$. Mas $(a + b) + c = (7 + 4) - 3 = 11 - 3$. Contudo o resultado intermediário 11 não pode ser representado dentro do nosso sistema! Temos uma condição de overflow em um cálculo intermediário, mas o resultado final estaria dentro do nosso sistema de numeração. Mas isso é tão ruim quanto o resultado final não poder ser representado porque o valor final estará errado se um valor intermediário estiver.

Portanto podemos ver nesse exemplo que a lei associativa da álgebra não vale para números de ponto fixo com número de dígitos finitos. Essa é uma conseqüência inevitável dessa forma de representação, e não existe nada a ser feito na prática, exceto detectar overflows quando eles ocorrerem, terminar o cálculo imediatamente e notificar o usuário da condição ou então, havendo detectado um overflow, repetir o cálculo com números com maior intervalo de representação (essa técnica é muito raramente usada, a não ser em aplicações críticas).

2.2.3 Raiz de sistemas numéricos

Nesta seção aprenderemos como trabalhar com números com bases arbitrárias, muito embora nos concentremos nas bases mais utilizadas em computadores digitais, tais como base 2 (binário) e seus parentes próximos base 8 (octal) e base 16 (hexadecimal).

A **base**, ou **raiz**, de um sistema numérico define o intervalo de valores que um dígito pode ter. No sistema em base 10 (decimal) um dos dez valores 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 é usado para cada dígito de um número. O sistema mais natural para se representar números em um computador é o de base 2, no qual os dados são representados por uma coleção de zeros e uns.

A forma geral de se determinar o valor decimal de um número na base k de um sistema numérico de ponto fixo é:

$$\text{Valor} = \sum_{i=-m}^{n-1} b_i \cdot k^i$$

O valor de um dígito na posição i é dado por b_i. Existem n dígitos para a esquerda do ponto e m dígitos à direita do ponto. Essa forma de representação na qual cada posição tem um valor é chamada de **código de posição ponderada** (weighted position code). Como exemplo, vamos determinar o valor de $(541{,}25)_{10}$, no qual o subscrito 10 representa a base. Temos $n = 3$, $m = 2$ e $k = 10$:

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

$$(500)_{10} + (40)_{10} + (1)_{10} + (2/10)_{10} + (5/100)_{10} = (541{,}25)_{10}$$

Agora consideremos na base 2 o número $(1010{,}01)_2$ no qual $n = 4$, $m = 2$ e $k = 2$:

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

$$(8)_{10} + (0)_{10} + (2)_{10} + (0)_{10} + (0/2)_{10} + (1/4)_{10} = (10.25)_{10}$$

Isso sugere uma maneira de se converter um número de uma base arbitrária para um número na base 10 usando o **método polinomial**. A idéia é multiplicar cada dígito pelo peso associado a ele por sua posição (potências de dois nesse exemplo) e então somar os termos para obter um número convertido. Muito embora conversões possam ser feitas entre todas as bases da mesma forma, algumas bases causam problemas especiais, como veremos na próxima seção.

Nota: Nesses sistemas de números ponderados nós definimos o bit que carrega o maior peso de **bit mais significativo** (MSB – Most Significant Bit), e o bit que carrega o menor valor de **bit menos significativo** (LSB – Least Significant Bit). De forma convencional, o MSB é o bit mais à esquerda, e o LSB é o bit mais à direita.

2.2.4 Conversões entre as raízes

Na seção anterior vimos um exemplo de como um número na base 2 pode ser convertido em um número na base 10. Uma conversão na direção reversa é mais complexa. A maneira mais simples de se converter números de ponto fixo contendo partes inteiras e fracionais é converter cada parte separadamente. Por exemplo, vamos converter $(23{,}375)_{10}$ para a base 2. Começamos separando o número nas suas partes inteira e fracional.

$$(23{,}375)_{10} = (23)_{10} + (0{,}375)_{10}$$

Convertendo a Parte Inteira de um Número de Ponto Fixo – O Método do Resto

Como sugerido na seção anterior, a forma polinomial geral para se representar um inteiro binário é

$$b_i \times 2^i + b_{i-1} \times 2^{i-1} + \ldots + b_1 \times 2^1 + b_0 \times 2^0$$

Se dividirmos o inteiro por 2 obteremos

$$b_i \times 2^{i-1} + b_{i-1} \times 2^{i-2} + \ldots + b_1 \times 2^0$$

com um resto de b_0. Como resultado de se dividir o inteiro original por 2, descobrimos que o valor do primeiro coeficiente binário é b_0. Podemos repetir esse processo no polinomial restante e determinar o valor de b_1. Então, podemos continuar iterando esse processo no polinomial restante e obter todos os b_i. Esse processo forma a base do método do resto de se converter inteiros entre bases.

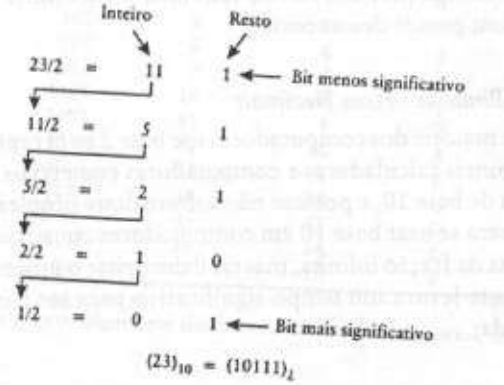


Figura 2.1 Uma conversão de um inteiro na base 10 para um inteiro na base 2 usando o método do resto.

Podemos agora aplicar o método do resto para converter $(23)_{10}$ para a base 2. Como mostrado na Figura 2.1, o inteiro é inicialmente dividido por 2, o que deixa um resto de 0 ou 1. Neste caso, 23/2 tem como resultado um quociente de 11 e um resto de 1. O primeiro resto é o least significant binary digit (bit) do número convertido (o bit mais à direita). No próximo passo 11 é dividido por 2, o que resulta em 5 e um resto de 1. O processo continua até que tenhamos um quociente de 0. Se continuarmos o processo após obtermos um quociente de 0, vamos obter somente zeros para o quociente e o resto, o que não vai modificar o valor do número convertido. Os restos são armazenados em um número na base 2 na ordem mostrada na Figura 2.1 para gerar os resultados $(23)_{10} = (10111)_2$. Em geral, podemos converter qualquer inteiro na base 10 para qualquer outra base simplesmente dividindo o inteiro pela base para a qual estamos convertendo.

Podemos checar o resultado convertendo o resultado da base 2 de volta para a base 10 usando o método polinomial:

$$\begin{aligned}(10111)_2 &= 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 16 + 0 + 4 + 2 + 1 \\ &= (23)_{10}\end{aligned}$$

Dessa forma conseguimos converter a parte inteira de $(23.375)_{10}$ para a base 2.

Convertendo a Parte Fracional de um Número de Ponto Fixo – O Método da Multiplicação

A conversão da parte fracional pode ser obtida por multiplicações sucessivas da fração por 2 como descrito a seguir.

A fração binária é representada na forma geral

$$b_{.1} \times 2^{-1} + b_{.2} \times 2^{-2} + b_{.3} \times 2^{-3} + \ldots$$

Se multiplicarmos a fração por 2, obteremos

$$b_{.1} + b_{.2} \times 2^{-1} + b_{.3} \times 2^{-2} + \ldots$$

Dessa forma determinamos o coeficiente $b_{.1}$. Se iterarmos esse processo na fração restante, obteremos os próximos b_i. Esse processo forma a base da multiplicação que converte frações entre as bases. No exemplo anterior (Figura 2.2), a fração inicial $(0{,}375)_{10}$ é menor que 1. Se a multiplicarmos por 2, o resultado final vai ser menor que dois. O dígito à esquerda do ponto será então 0 ou 1. Este é o primeiro dígito à direita do ponto no número convertido para a base 2, como mostrado na figura. Repetimos o processo na parte fracional até que reste somente uma fração de 0. Nesse ponto somente zeros adicionais serão criados por

novas iterações, ou então chegaremos ao limite da precisão usada na nossa representação. Os dígitos são armazenados e o resultado obtido é $(0,375)_{10} = (0,11)_2$.

Nesse processo, o multiplicador é o mesmo do valor da base-alvo. O multiplicador é 2 nesse caso, mas se quiséssemos converter para outra base, tal como 3, por exemplo, então utilizaríamos 3 como multiplicador[1].

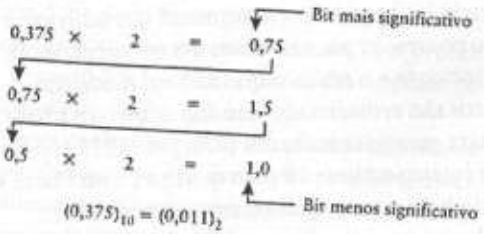


Figura 2.2 Uma conversão de uma fração na base 10 para a base 2 usando o método da multiplicação.

Mais uma vez podemos checar o resultado da conversão convertendo da base 2 de volta para a base 10 usando o método polinomial:

$$(0,011)_2 = 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 0 + 1/4 + 1/8 = (0,375)_{10}$$

Finalmente podemos combinar as partes inteira e fracional do número no resultado final:

$$(23,375)_{10} = (10111,011)_2$$

Frações Infinitas

Muito embora esse método de conversão funcione entre todas as bases, alguma precisão pode ser perdida no processo. Por exemplo, nem todas as frações infinitas na base 10 têm um formato finito na base 2. Por exemplo, vamos converter $(0,2)_{10}$ para a base 2, como mostrado na Figura 2.3. Na última linha da conversão a fração $(0,2)_{10}$ reaparece e o processo se repete *ad infinitum*. Para ver por que isso acontece, notemos que qualquer fração que não se repete na base 2 pode ser representada como $i/2^k$ para inteiros i e k. (Frações que se repetem na base 2 não podem ser representadas nesse formato.) De forma algébrica:

[1] Alternativamente, podemos usar o sistema numérico na base 10 e evitar a conversão de uma representação na base 2, no qual combinações de uns e zeros representam os dígitos na base 10. Essa representação é conhecida como decimal codificado em binário (BCD – Binary Coded Decimal), que veremos em maior detalhe mais tarde.

$$2^k = i \times 5^k/(2^k \times 5^k) = i \times 5^k/10^k = j/10^k$$

onde j é o inteiro $i \times 5^k$. A fração portanto não se repete na base 10. Isso se baseia no fato de que somente frações que não se repetem na base b podem ser representadas como i/b^k para inteiros i e k. A condição que tem que ser satisfeita para uma fração que não se repete na base 10 ter um equivalente (que não se repete) na base 2 é:

$$i/10^k = i/(5^k \times 2^k) = j/2^k$$

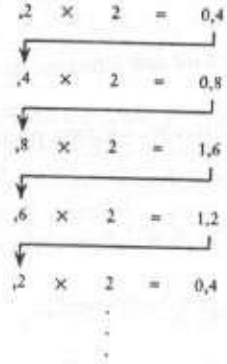


Figura 2.3 Uma fração finita em base 10 que não tem uma forma finita na base 2.

onde $j = i/5^k$, e 5^k deve ser um fator de i. Para frações decimais de um dígito somente $(0,0)_{10}$ e $(0,5)_{10}$ não se repetem na base 2 (20% de todas as frações possíveis); para frações decimais de dois dígitos somente $(0,00)_{10}$, $(0,25)_{10}$, $(0,5)_{10}$ e $(0,75)_{10}$ não se repetem (4% das frações possíveis) etc. Existe uma ligação entre números primos entre si e frações que se repetem, o que ajuda a entender por que algumas frações finitas em base 10 não têm uma forma finita na base 2. Knuth (1981) explica um pouco desta teoria.

Representações Binárias versus *Decimais*

Muito embora a maioria dos computadores use base 2 para representação inteira e aritmética, algumas calculadoras e computadores comerciais usam uma representação interna de base 10, e por isto não sofrem deste problema de representação. O motivo para se usar base 10 em computadores comerciais não é somente evitar o problema da fração infinita, mas também evitar o processo de conversão, que historicamente levava um tempo significativo para ser feita pelas unidades de entrada e saída.

Representações Binárias, Octais e Hexadecimais

Os números binários refletem a representação interna real usada pela maioria das máquinas. Contudo eles sofrem da desvantagem que números representados

na base 2 normalmente necessitam de mais dígitos que números em outras bases (porque?), e é mais fácil cometer erros quando escrevendo-os devido às longas seqüências de zeros e uns. Já mencionamos que a base 8, **raiz octal**, e a base 16, **raiz hexadecimal**, são relacionadas com a base 2. Isto, é devido às três raízes serem divisíveis por 2, a menor. Vamos agora ver que converter entre as três bases 2, 8 e 16 é trivial e existem vantagens significativas em se representar números nestas bases.

Números binários podem ser consideravelmente mais longos que seus equivalentes na base 10. Como uma conveniência notacional, podemos às vezes usar bases maiores do que 2 que são múltiplas de 2. Converter entre as bases 2, 8 ou 16 é mais simples que converter para a base 10. Os valores usados para os dígitos na base 8 são familiares para nós, usuários da base 10, mas para a base 16 (hexadecimal) necessitamos de seis dígitos adicionais além dos usados para a base 10. As letras A, B, C, D, E e F ou os seus equivalentes em letras minúsculas são comumente usados para representar os valores 10, 11, 12, 13, 14 e 15 respectivamente em hexadecimal. Os dígitos usados normalmente para as bases 2, 8, 10 e 16 são mostrados na Figura 2.4. Ao comparar a coluna da base 2 com as bases 8 e 16 necessitamos de 3 bits para representar cada dígito na base 8 em binário, e necessitamos de quatro bits para representar cada dígito da base 16 em binário. Em geral, k bits são necessários para representar cada dígito na base 2^k, k inteiro, então a base $2^3 = 8$ usa três bits e a base $2^4 = 16$ usa 4 bits.

Binário (base 2)	Octal (base 8)	Decimal (base 10)	Hexadecimal (base 16)
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

Figura 2.4 Valores para os dígitos em sistemas numéricos binário, octal, decimal e hexadecimal.

Para se converter um número da base 2 para a base 8 temos que particionar o número da base 2 em grupos de três começando do ponto, e preencher os grupos mais externos com zeros quando preciso. Então convertemos cada triplo para o

octal equivalente. Para converter da base 2 para a base 16 usamos grupos de quatro. Por exemplo, vamos converter $(10110)_2$ para a base 8:

$$(10110)_2 = (010)_2\,(110)_2 = (2)_8\,(6)_8 = (26)_8$$

Note que os dois bits mais à esquerda foram preenchidos com zero à esquerda para criar uma tripla completa.

Para converter $(10110110)_2$ para a base 16:

$$(10110110)_2 = (1011)_2\,(0110)_2 = (B)_{16}\,(6)_{16} = (B6)_{16}$$

(Note que B é o dígito na base 16 correspondente a $(11)_{10}$. B não é uma variável.)

Os métodos de conversão podem ser usados para converter um número de qualquer base para qualquer outra base, mas pode não ser muito intuitivo converter algo como $(513.03)_6$ para a base 7. Para nos ajudar em uma conversão não comum podemos converter para a mais familiar base 10 como um passo intermediário e então continuar a conversão da base 10 para a base-alvo. Como regra geral, o método polinomial deve ser usado quando convertendo para a base 10, e os métodos do resto e multiplicação quando convertendo a partir da base 10.

2.2.5 Um primeiro estudo de aritmética computacional

Vamos ver aritmética computacional em detalhe no Capítulo 3, mas no momento precisamos aprender como fazer uma simples adição binária porque ela é usada na representação de números binários com sinais. Adição binária é feita de forma similar à usada para se somar números decimais manualmente, como ilustrado na Figura 2.5. Dois números binários A e B são somados da direita para a esquerda, criando uma soma e um carry (vai um) em cada posição de bit. Uma vez que os bits mais à direita de A e B podem cada um assumir um de dois valores, quatro casos têm de ser considerados: 0 + 0, 0 + 1, 1 + 0 e 1 + 1, com um carry de 0, como mostrado na figura. O carry para o bit mais à direita é 0. Para as outras posições de bit, o carry pode ser 0 ou 1, de tal forma que um total de oito combinações de entradas têm de ser consideradas como mostrado na figura.

Note que o maior número que pode ser representado usando o formato de oito bits mostrado na Figura 2.5 é $(11111111)_2 = (255)_{10}$ e que o menor número que pode ser representado é $(00000000)_2 = (0)_{10}$. Os padrões de bits 11111111 e 00000000 e todos os padrões intermediários de bits representam números no intervalo fechado de 0 a 255, todos números positivos. Até este momento consideramos somente números sem sinal, mas precisamos representar números com sinal também. Desta forma (aproximadamente) metade dos padrões de bits é reservada para números positivos e a outra metade para números negativos. Quatro representações comuns para números com sinal na base 2 são discutidas na próxima seção.

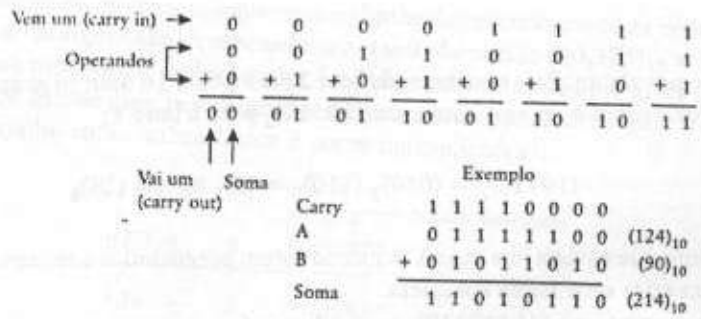


Figura 2.5 Exemplo de adição binária.

2.2.6 Números de ponto fixo com sinal

Até agora estudamos somente a representação de números de ponto fixo sem sinal. A situação é diferente quando representamos números de ponto fixo *com sinal*. Existem quatro maneiras de se representar números com sinal que são comumente usadas: sinal magnitude, complemento de um, complemento de dois e notação de excesso. Vamos ver cada uma, usando inteiros em nossos exemplos. Durante esta discussão o leitor pode se referir à Tabela 2.1, que mostra como números de 3 bits são armazenados nas várias representações.

Tabela 2.1 Representações de Inteiros de Três Bits.

Decimal	Sem sinal	Sinal magnitude	Complemento de 1	Complemento de 2	Excesso 4
7	111	–	–	–	–
6	110	–	–	–	–
5	101	–	–	–	–
4	100	–	–	–	–
3	011	011	011	011	111
2	010	010	010	010	110
1	001	001	001	001	101
+0	000	000	000	000	100
–0	–	100	111	000	100
–1	–	101	110	111	011
–2	–	110	101	110	010
–3	–	111	100	101	001
–4	–	–	–	100	000

Sinal Magnitude

A representação de **sinal magnitude** (às vezes chamada de **sinal e magnitude**) é a mais familiar a nós acostumados com o sistema numérico de base 10. Um sinal positivo ou negativo à esquerda do número indica se o número é positivo ou negativo, como por exemplo em $+12_{10}$ ou -12_{10}. Na representação de sinal magnitude binária, o bit mais à esquerda é usado para o sinal, que tem o valor de 0 ou 1 para + ou –, respectivamente. Os bits restantes contêm a magnitude absoluta do valor. Representando-se $(+12)_{10}$ e $(-12)_{10}$ em um formato com 8 bits:

$$(+12)_{10} = (00001100)_2$$

$$(-12)_{10} = (10001100)_2$$

O número negativo é formado simplesmente trocando o bit de sinal do número positivo de 0 para 1. Note que existem duas representações para zero, +0 (00000000) e –0 (10000000).

Existem 8 bits neste exemplo, e todos os padrões de bits representam números válidos; portanto existem $2^8 = 256$ padrões válidos. Contudo, somente $2^8 - 1 = 255$ números diferentes podem ser representados porque +0 e –0 representam o mesmo número.

Nós vamos usar a representação de sinal magnitude quando virmos números de ponto flutuante na Seção 2.3

Complemento de Um

A operação de **complemento de um** é trivial de se fazer: troque todos os uns do número por zeros e todos os zeros por uns. Veja a quarta coluna da Tabela 2.1 para exemplos. Podemos observar na tabela que na representação de **complemento de um** o bit mais à esquerda é 0 para números positivos e 1 para números negativos, assim como na representação de sinal magnitude. Esta negação, trocar uns e zeros, é conhecida como **complementação** de bits. Vejamos, por exemplo, a representação de $(+12)_{10}$ e $(-12)_{10}$ em um formato de 8 bits, agora usando a representação de complemento de um:

$$(+12)_{10} = (00001100)_2$$

$$(-12)_{10} = (11110011)_2$$

Note novamente que existem representações para +0 e –0, que são 00000000 e 11111111, respectivamente. Como resultado, existem somente $2^8 - 1 = 255$ números diferentes que podem ser representados, muito embora existam 2^8 padrões de bits diferentes.

A representação em complemento de um não é comumente utilizada. Em parte isso se explica pela dificuldade de se efetuar comparações quando existem duas representações para 0. Existe uma dificuldade adicional relacionada com a adição de números, que é discutida no Capítulo 3.

Complemento de Dois

O complemento de dois é formado de maneira similar a do complemento de um: complementando todos os bits do número, mas em seguida soma-se 1 ao resultado, e se esta adição resultar em "vai um" (carry) do bit mais significativo, ignore o carry. Examinando-se a quinta coluna da Tabela 2.1 podemos ver que na representação do **complemento de dois** o bit mais à esquerda é novamente 0 para números positivos e 1 para números negativos. Contudo, este formato não tem a desafortunada característica de ambos, sinal, magnitude e complemento de um: existe somente uma representação para zero. Para ver que isso é verdade, tente representar $(+0)_{10}$, que tem o padrão de bits:

$$(+0)_{10} = (00000000)_2$$

Formar o complemento de um de $(00000000)_2$ gera $(11111111)_2$, e somando um a este resultado obtemos $(00000000)_2$, ou seja $(-0)_{10} = (00000000)_2$. O carry da posição mais à esquerda é ignorado na adição em complemento de dois (exceto para detecção de condições de overflow). Como existe somente uma representação para 0, e como todos os padrões de bits são válidos, existem $2^8 = 256$ números diferentes que podem ser representados.

Vejamos novamente a representação de $(+12)_{10}$ e $(-12)_{10}$ em um formato de 8 bits, desta vez usando a representação em complemento de dois. Começando com $(+12)_{10} = (00001100)_2$, vamos complementar, ou negar, o número gerando $(11110011)_2$. Somando 1 temos $(11110100)_2$, e portanto $(-12)_{10} = (11110100)_2$:

$$(+12)_{10} = (00001100)_2$$

$$(-12)_{10} = (11110100)_2$$

Existe um número igual de números positivos e negativos se considerarmos zero como um número positivo, o que é razoável uma vez que seu bit de sinal é 0. Os números positivos começam em 0, mas os números negativos começam em –1, então a magnitude do número mais negativo é um maior que a magnitude do número mais positivo. O número positivo com maior magnitude é +127, e o número negativo com a maior magnitude é –128. Não existe, portanto, um número positivo que corresponda ao negativo –128 que possa ser representado. Se tentarmos formar o complemento de dois de –128 chegamos a um número negativo:

$$(-128)_{10} = (10000000)_2$$

$$\Downarrow$$

$$\begin{array}{rr} & 01111111 \\ + & 1 \\ \hline & (10000000)_2 \end{array}$$

A representação em complemento de dois é a representação mais comumente usada em computadores convencionais, e a usaremos freqüentemente neste livro.

Representação de Excesso

Na representação de **excesso**, ou **biased**, o número é tratado como sem sinal, mas é "shifted" (deslocado) em valor subtraindo-se um fator (bias, em inglês) do mesmo. A idéia é atribuir o menor padrão numérico de bits, todos zero, ao negativo do fator, e atribuir aos números restantes na seqüência os padrões de bits que crescem em magnitude. Uma maneira conveniente de se pensar na representação em excesso é que o número é representado pela soma do seu complemento de dois e outro número, conhecido como "excesso", ou "bias". Mais uma vez, veja a coluna mais à direita da Tabela 2.1 para exemplos.

Vamos representar mais uma vez $(+12)_{10}$ e $(-12)_{10}$ em um formato de oito bits mas agora usando uma representação em excesso de 128. Um número em excesso de 128 é formado adicionando-se 128 ao número original e então criando uma versão binária sem sinal. Para $(+12)_{10}$ calculamos $(128 + 12 = 140)_{10}$ e usamos o padrão $(10001100)_2$. Para $(-12)_{10}$, calculamos $(128 + -12 = 116)_{10}$ e usamos o padrão de bits $(01110100)_2$:

$$(+12)_{10} = (10001100)_2$$

$$(-12)_{10} = (01110100)_2$$

Note que não existe um significado numérico para o valor de excesso: ele simplesmente tem o efeito de deslocar a representação dos números em complemento de dois.

Existe somente uma representação em excesso para 0, uma vez que a representação em excesso é simplesmente uma versão deslocada da representação em complemento de dois. Para o caso anterior, o valor do excesso foi escolhido de forma a ter o mesmo padrão de bits do maior número negativo, o que tem o efeito de fazer com que os números apareçam ordenados numericamente quando vistos como números binários sem sinal. Portanto, o número mais negativo é $(-128)_{10} = (00000000)_2$ e o mais positivo é $(+127)_{10} = (11111111)_2$. Essa re-

presentação simplifica a comparação entre números, porque os padrões de bits para números negativos têm valores numericamente menores do que os padrões de bits para números positivos. Isso é importante para representar os expoentes de números em ponto flutuante, nos quais os expoentes de dois números são comparados a fim de torná-los iguais para adição e subtração. Representações de números de ponto flutuante serão vistas na Seção 2.3.

2.2.7 Decimal codificado em binário

Números podem ser representados na base 10 mesmo usando codificação binária. Cada dígito na base 10 ocupa quatro bits, o que é conhecido como **decimal codificado em binário** (Binary Coded Decimal – BCD). Cada dígito BCD pode ter um entre dez valores. Existem $2^4 = 16$ padrões possíveis para cada dígito na base 10, e como resultado seis padrões de bits não são usados para cada dígito. Na Figura 2.6 existem quatro dígitos significativos, portanto $10^4 = 10.000$ padrões de bits são válidos, muito embora $2^{16} = 65.536$ padrões de bits são possíveis com 16 bits.

Muito embora alguns padrões de bits não sejam usados, o formato BCD é comumente usado em calculadoras e em aplicações comerciais. Existem menos problemas em representar frações finitas na base 10 nesse formato, ao contrário da representação em base 2. Dados entrados na base 10 (como em uma calculadora, por exemplo) não precisam ser convertidos para uma representação interna na base 2, ou convertidos de uma representação interna na base 2 para uma saída na base 10.

Efetuar cálculos aritméticos em números BCD com sinal pode não ser óbvio. Muito embora estejamos acostumados a usar a representação sinal magnitude na base 10, um método diferente de representar números na base 10 com sinal é usado no computador. No sistema numérico em **complemento de nove**, números positivos são representados como em BCD comum, mas o dígito mais à esquerda é menor que 5 para números positivos e 5 ou maior para números negativos. O negativo em complemento de nove é formado subtraindo-se cada dígito de 9. Por exemplo, o número na base 10 +301 é representado como 0301 (ou simplesmente 301) nos complementos de nove e de dez conforme mostrado na Figura 2.6a. O negativo em complemento de nove é 9698 (Figura 2.6b), que é obtido subtraindo-se cada dígito de 0301 de 9.

O negativo em **complemento de dez** é formado adicionando-se 1 ao negativo em complemento de nove, o que faz com que a representação em complemento de dez de –301 seja 9698 + 1 = 9699, como mostrado na Figura 2.6c. Nesse exemplo, os números positivos variam de 0 a 4999 e os números negativos variam de 5000 a 9999.

2.3 Números de Ponto Flutuante

A representação de números de ponto fixo, que estudamos na Seção 2.2 tem uma posição fixa para o ponto e um número fixo de dígitos à esquerda e à direita do

ponto. Uma representação de ponto fixo pode precisar de um número muito grande de dígitos para representar um intervalo prático de valores. Por exemplo, um computador pode representar números tão grandes como um trilhão mantendo pelo menos 40 bits à esquerda do ponto, uma vez que $2^{40} \approx 10^{12}$. Se no mesmo computador for necessário representar um trilhonésimo, então 40 bits devem também ser mantidos à direita do ponto, resultando num total de 80 bits por número.

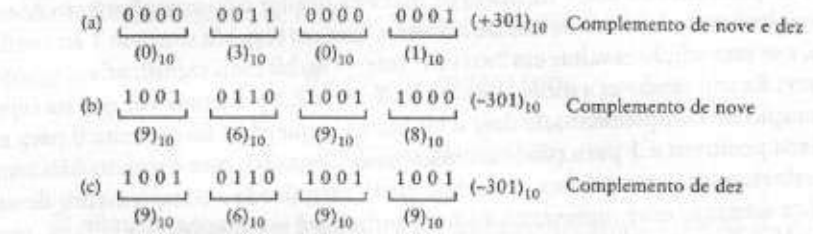


Figura 2.6 Representações BCD de 301 (a) e –301 em complemento de nove (b) e em complemento de dez (c).

Na prática, números muito maiores e muito menores aparecem freqüentemente, exigindo ainda mais do computador. Um hardware muito sofisticado é necessário para se armazenar e manipular números com 80 ou mais bits de precisão, e a computação procede mais lentamente para um grande número de dígitos do que para um pequeno número de dígitos. Alta precisão, contudo, não é geralmente necessária quando números grandes são usados, e, da mesma forma, números grandes em geral não precisam ser representados quando cálculos são feitos com pequenos números. Um computador mais eficiente pode ser construído quando se guarda somente a precisão que será efetivamente usada.

2.3.1 Intervalo de representação e precisão em números de ponto flutuante

Uma representação de **ponto flutuante** permite um grande intervalo de representação usando um número pequeno de dígitos através da separação dos dígitos usados para a *precisão* dos dígitos usados para o *intervalo*. O número de ponto flutuante na base 10 que representa o número de Avogrado é:

$$+6.023 \times 10^{23}$$

Aqui o intervalo é representado por uma potência de 10, 10^{23} neste caso, e a precisão é representada pelos dígitos no número de ponto fixo, 6,023 neste caso. Na discussão a respeito de números de ponto flutuante, a parte em ponto fixo é denominada **mantissa** do número. Logo, um número de ponto flutuante pode ser caracterizado por três números: **sinal, expoente e mantissa.**

Figura 2.7 Representação de um número de ponto flutuante na base 10.

O intervalo de representação é determinado principalmente pelo número de dígitos no expoente (dois dígitos são usados aqui) e pela base à qual o expoente é elevado (aqui a base é 10). A precisão é determinada principalmente pelo número de dígitos na mantissa (quatro dígitos são usados aqui). Conseqüentemente, o número completo pode ser representado por um sinal e seis dígitos, dois para o expoente, e quatro para a mantissa. A Figura 2.7 mostra como o triplo sinal, expoente e mantissa podem ser formatados em um computador. Note como os dígitos são compactados com o sinal primeiro, seguido pelo expoente, seguido pela mantissa. Essa ordenação foi escolhida por simplificar a comparação de números de ponto flutuante. O leitor não deve esquecer de que o ponto decimal não necessita ser armazenado com o número, conquanto que ele esteja sempre na mesma posição da mantissa. (Isso será visto em mais detalhes na Seção 2.3.2.)

Se um intervalo maior de representação for necessário, e se estamos dispostos a sacrificar a precisão, podemos simplesmente usar três dígitos para a mantissa e reservar três dígitos para o expoente sem aumentar o número de dígitos usados na representação. Um método alternativo de aumentar o intervalo é aumentar a base, o que tem o efeito de aumentar a precisão dos números menores, mas diminuir a precisão dos maiores. O compromisso entre intervalo de representação/precisão é uma grande vantagem de se usar uma representação de ponto flutuante, mas a precisão reduzida pode causar problemas, às vezes levando a desastres, como, por exemplo, o que é descrito na Seção 2.4.

2.3.2 Normalização e o bit escondido

Um problema em potencial com a representação de números em ponto flutuante é que o mesmo número pode ser representado de diversas maneiras, o que faz com que comparações e operações aritméticas tornem-se complexas. Por exemplo, considere as formas numéricas equivalentes abaixo:

$$3584.1 \times 10^0 = 3.5841 \times 10^3 = 0{,}35841 \times 10^4$$

A fim de se evitar representações múltiplas para o mesmo número, números de ponto flutuante são mantidos em uma forma **normalizada**, ou seja, o ponto é deslocado para a esquerda ou para a direita (e o expoente ajustado apropriadamente) até que o ponto esteja à esquerda do dígito diferente de zero mais à esquerda. Dessa forma o número mais à direita da equação anterior é a representação normalizada. Infelizmente, o número zero não pode ser representado dessa maneira, portanto a representação do zero não segue essa regra. A exceção é que o zero é representado com a mantissa consistindo somente em zeros.

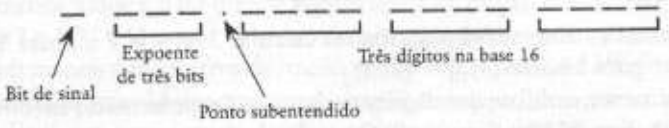
Se a mantissa for representada em binário, ou seja, na base 2, e se a condição de normalização exigir que exista um 1 inicial na mantissa normalizada, então não existe necessidade de se armazenar este 1. A maioria dos formatos de ponto flutuante não o armazena. Em vez disso, ele é eliminado antes de se compactar o número para armazenamento, e é recolocado quando descompactando o número de volta em expoente e mantissa. Isso resulta em ter disponível um bit adicional de precisão à direita do número, uma vez que removemos o bit mais à esquerda. Esse bit removido é chamado de **bit escondido**, também conhecido como **1 escondido**. Por exemplo, se a mantissa em um dado formato for 0,11010 após normalização, então o padrão de bits armazenado é 1010 – o bit mais à esquerda é truncado, ou escondido. Veremos que o formato de números de ponto flutuante IEEE[2] 754 usa o bit escondido.

2.3.3 Representando números de ponto flutuante em computadores – preliminares

Projetaremos um formato simples para representação de números de ponto flutuante para ilustrar os pontos mais importantes dessa representação em computadores. Nosso formato pode parecer desnecessariamente complexo à primeira vista. Vamos representar a mantissa em um formato sinal magnitude com um bit para o sinal e três dígitos hexadecimais para a magnitude. O expoente será um número de três bits em excesso de 4, com uma raiz de 16. A forma normalizada do número terá o ponto hexadecimal à esquerda dos três dígitos hexadecimais.

Os bits serão compactados da seguinte forma: o bit de sinal à esquerda, seguido pelo expoente de 3 bits, seguido pelos três bits hexadecimais da mantissa. Nem a raiz nem o ponto hexadecimal serão armazenados na forma compactada.

O motivo pelo qual escolhemos esse formato aparentemente estranho é que números nesse formato podem ser comparados para $=, \neq \geq$ e $\leq$ em sua forma "compactada", que é mostrada na seguinte ilustração:



Representemos o número $(358)_{10}$ nesse formato.

[2] Institute of Electrical and Electronics Engineers.

O primeiro passo é converter o número de ponto fixo da sua base original para um número de ponto fixo na base destino. Usando o método descrito na Seção 2.1.3, podemos converter números na base 10 para a base 16 da seguinte maneira:

		Quociente	Resto
358/16	=	22	6
22/16	=	1	6
1/16	=	0	1

Portanto, $(358)_{10} = (166)_{16}$. O próximo passo é converter o número de ponto fixo em um número de ponto flutuante:

$$(166)_{16} = (166.)_{16} \times 16^0$$

Note que a forma 16^0 reflete a base 16 com um expoente de 0 e que o número 16 como aparece na página usa a forma em base 10, ou seja, $(16^0)_{10} = (10^0)_{16}$. Isso é apenas uma conveniência notacional para se descrever um número em ponto flutuante.

O próximo passo é normalizar o número:

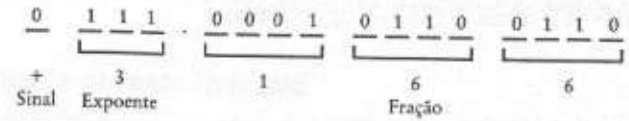
$$(166.)_{16} \times 16^0 = (0{,}166)_{16} \times 16^3$$

Finalmente, precisamos preencher os campos restantes do formato. O número é positivo, portanto colocamos um 0 na posição do bit de sinal. O expoente é 3, mas é representado em excesso de 4, o que faz com que o padrão de bits para o expoente seja calculado como:

	0 1 1	$(+3)_{10}$
Excesso de 4	+ 1 0 0	$(+4)_{10}$
Expoente em excesso de 4	1 1 1	

De maneira alternativa poderíamos calcular 3 + 4 = 7 na base 10, e então converter para binário $(7)_{10} = (111)_2$

Finalmente, cada um dos dígitos em base 16 é representado em binário como 1 = 0001, 6 = 0110 e 6 = 0110. O resultado é:



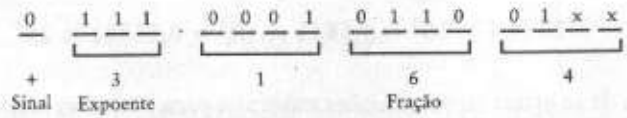
Note mais uma vez que o ponto não é representado explicitamente no padrão de bits resultante, mas sua presença é assumida. Os espaços entre os dígitos são somente para clareza de visualização, os bits não são armazenados com espaços entre eles. O padrão de bits da maneira que seria armazenado na memória do computador é:

0111000101100110

O uso de um expoente em excesso de 4 em vez de uma representação em complemento de dois ou sinal magnitude simplifica a adição e a subtração de números de ponto flutuante (que será estudado em detalhe no Capítulo 3). A fim de se somar ou subtrair dois números de ponto flutuante normalizados, o menor expoente (menor em grau, não em magnitude) deve ser aumentado até se igualar ao expoente maior (este mantém seu intervalo), o que tem o efeito de "desnormalizar" o número menor. Para se determinar qual expoente é maior, precisamos somente de tratar dos padrões de bits como números sem sinal e compará-los, ou seja, usando uma representação em excesso de 4, o menor expoente é –4, representado como 000. O maior expoente é +3, representado como 111. Os padrões de bits restantes para –3, –2, –1, 0, +1 e + 2 correspondem aos padrões na sua ordem 001, 010, 011, 100, 101, 110.

Dessa maneira, dado o padrão de bits mostrado previamente para $(358)_{10}$ junto com uma descrição da representação de ponto flutuante, podemos facilmente determinar o número. O bit de sinal é 0, significando que o número é positivo. O expoente na forma sem sinal é $(+7)_{10}$, mas uma vez que estamos usando excesso de 4, devemos subtrair 4 do mesmo, resultando no expoente real de $(+7 - 4 = +3)_{10}$. A fração é agrupada em quatro bits hexadecimais, o que dá uma fração de $(+.166)_{16}$. O resultado final é $(+0{,}166 \times 16^3)_{16} = (358)_{10}$.

Suponhamos agora que somente 10 bits são permitidos para a fração no exemplo anterior, em vez de 12 bits que agrupam naturalmente em grupos de quatro cada um correspondendo a um dígito hexadecimal. Como isso afetaria a representação? Uma maneira de se resolver o problema seria arredondar a fração e ajustar o expoente de maneira apropriada. Uma outra maneira, usada aqui, seria simplesmente truncar os bits menos significativos e evitar quaisquer ajustes no expoente, de tal forma que o número realmente representado é



Se tratarmos os bits eliminados como zeros, então esse padrão de bits representa $(.164 \times 16^3)_{16}$. Esse método produz um erro não-aleatório, uma vez que os valores 00, 01, 10 e 11 dos bits eliminados são todos tratados como 0, e, dessa

forma, o erro fica no intervalo de 0 a $(.003)_{16}$. O fator não-aleatório existe porque o erro não é simétrico ao redor de 0. Esse problema não será explorado aqui, mas um tratamento completo pode ser encontrado em Hamacher *et al.* (1990).

Mais uma vez lembramos que qualquer que seja o formato usado para números de ponto flutuante, é necessário notar que todos devem armazenar e restaurar números no mesmo formato. A IEEE tem tomado a frente na padronização de formatos de ponto flutuante. O formato de ponto flutuante IEEE 754, que é usado quase que universalmente, é discutido na Seção 2.3.5.

2.3.4 Erro em representações de ponto flutuante

É inevitável que uma representação com precisão finita introduza erros, o que significa que devemos considerar quão grande o erro pode ser (por "erro" queremos dizer a distância entre dois valores representáveis adjacentes) e se esse erro é aceitável para nossa aplicação. Como um exemplo de um problema em potencial, consideremos representar um milhão em ponto flutuante, e então subtrair 1 deste número um milhão de vezes. Podemos ao final ainda ter um milhão se o erro for maior que 1.[3]

Para caracterizarmos o erro, intervalo de representação e precisão usaremos a seguinte notação:

b	Base
s	Número de *dígitos* significativos (não bits) na fração
M	Maior expoente
m	Menor expoente

O número de dígitos significativos na fração é representado por s, que é diferente do número de bits na fração se a base for diferente de 2 (por exemplo, base 16 usa quatro bits para cada dígito). Em geral, se a base for 2^k, com k inteiro, então k bits serão necessários para representar cada dígito. O uso de um 1 escondido aumenta s por um bit, mesmo que não aumente o total de números representáveis. No exemplo anterior, existem três dígitos significativos na fração da base 16 e 12 bits que formam os três dígitos. Existem três bits em excesso de 4 no expoente, o que determina um intervalo para o expoente de $[-2^2$ a $2^2 - 1]$. Para esse caso, $b = 16$, $s = 3$, $M = 3$, e $m = -4$.

Na análise da representação de ponto flutuante, existem cinco características que devemos considerar: o número de números representáveis, os números que terão a maior e a menor magnitude (diferente de zero) e os tamanhos dos maiores e menores intervalos entre números sucessivos.

O número de números representáveis pode ser determinado como mostrado na Figura 2.8. O bit de sinal pode ter dois valores, como indicado pela posição

[3] A maioria dos computadores hoje em dia permite um limite superior tão alto quanto 8 milhões usando a precisão default.

marcada com "A". O número total de expoentes é indicado na posição B. Note que nem todos os padrões de bits para o expoente são válidos em todas as representações. O padrão de ponto flutuante IEEE 754, que estudaremos em breve, tem o menor expoente de −126, muito embora o expoente de oito bits possa representar um número tão pequeno quanto −128. Os expoentes proibidos são reservados para números especiais, tais como zero e infinito.

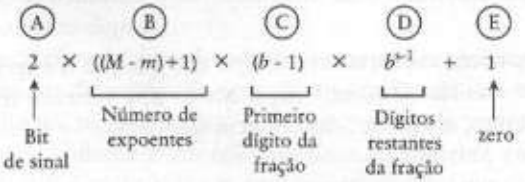


Figura 2.8 Cálculo do número de números representáveis em uma representação de ponto flutuante.

O primeiro dígito da fração é analisado a seguir, e pode assumir qualquer valor, exceto 0 em uma representação normalizada (menos quando um 1 escondido é usado), conforme o indicado por $(b - 1)$ na posição C. Os dígitos restantes da fração também podem assumir quaisquer dos b valores para a base, como indicado por b^{s-1} na posição D. Se um 1 escondido for usado, então a posição C é removida e a posição 4 é substituída por b^s. Finalmente, deve existir uma representação para 0, o que é levado em conta na posição E.

Consideremos agora os números com a menor e a maior magnitude. O número com a menor magnitude tem o menor expoente e a menor fração normalizada diferente de zero. Deve existir um valor diferente de zero para o primeiro dígito, e uma vez que um 1 é o menor valor que podemos armazenar aqui, a menor fração é b^{-1}. O número com a menor magnitude então é $b^m \cdot b^{-1} = b^{m-1}$. Finalmente, deve existir uma representação para 0, que é levada em conta na posição E.

Os menores e maiores intervalos entre valores representáveis são determinados de maneira semelhante. O menor intervalo ocorre quando o expoente está com seu menor valor e o bit menos significativo da fração muda. Este intervalo é de $b^m \cdot b^{-s} = b^{m-s}$. O maior intervalo ocorre quando o expoente tem seu maior valor e o bit menos significativo da fração muda. Este intervalo é $b^M \cdot b^{-s} = b^{M-s}$.

Por exemplo, considere uma representação de ponto flutuante na qual existe um bit de sinal, um expoente de dois bits representado como excesso de 2, e uma fração normalizada na base 2 de três bits na qual o primeiro 1 é visível; ou seja, não se usa o 1 escondido. A representação para 0 é o padrão de bits 00000000. Uma linha numérica mostrando todos os possíveis números que podem ser representados neste formato é mostrada na Figura 2.9. Note que existe um intervalo relativamente grande entre 0 e o primeiro número representável, porque a representação normalizada não aceita padrões de bits que correspondam para números entre 0 e o primeiro número representável.

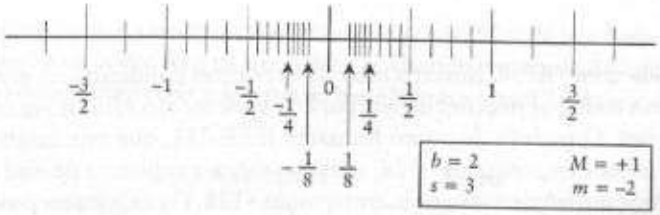


Figura 2.9 Uma linha numérica mostrando todos os números representáveis em um formato simples de ponto flutuante.

O menor número representável aparece quando o expoente e a fração têm seus menores valores. O menor expoente é –2, e a menor fração normalizada é $(.100)_2$. Portanto, o menor número representável é

$$b^m \times b^{-1} = b^{m-1} = 2^{-2-1} = 1/8$$

De forma similar, o maior número representável aparece quando o expoente e a fração estão nos seus maiores valores. A maior fração acontece quando a fração é somente uns, o que representa um número que é 2^{-3} menor que 1 porque existem três dígitos significativos na fração. O maior número representável então é

$$b^M \times (1 - b^{-s}) = 2^1 \times (1 - 2^{-3}) = 7/4$$

O menor intervalo acontece quando o expoente tem seu menor valor, e o bit menos significativo da fração muda, o que é

$$b^m \times b^{-s} = b^{m-s} = 2^{-2-3} = 1/32$$

De forma similar, o maior intervalo acontece quando o expoente tem seu maior valor e o bit menos significativo da fração muda, o que é

$$b^M \times b^{-s} = b^{M-s} = 2^{1-3} = 1/4$$

O número de padrões de bits que representam números válidos é menor que o número existente de padrões de bits, por causa da normalização. Como discutido anteriormente, o número de números representáveis consiste em cinco partes, que acontecem com o bit de sinal, os expoentes, o primeiro dígito significativo, os dígitos restantes e o padrão de bits para 0. Isso é calculado como se vê a seguir.

$$2 \times ((M - m) + 1) \times (b - 1) \times b^{s-1} + 1$$

$$= 2 \times ((1 - (-2)) + 1) \times (2 - 1) - 2^{3-1} + 1$$

$$= 33$$

Note como os intervalos são pequenos para números pequenos e grandes para números grandes. Na verdade, o erro relativo é aproximadamente o mesmo para todos os números. Se calcularmos a razão entre um intervalo grande e um número grande e comparar com a razão de um intervalo pequeno e um número pequeno, veremos que as razões são as mesmas.

$$\text{Um intervalo grande} \longrightarrow \qquad \text{Um número grande} \longrightarrow \qquad \frac{b^{M-s}}{b^M \times (1 - b^{-s})} = \frac{b^{-s}}{1 - b^{-s}} = \frac{1}{b^s - 1}$$

e

$$\text{Um intervalo pequeno} \longrightarrow \qquad \text{Um número pequeno} \longrightarrow \qquad \frac{b^{m-s}}{b^m \times (1 - b^{-s})} = \frac{b^{-s}}{1 - b^{-s}} = \frac{1}{b^s - 1}$$

A representação para "números pequenos" é usada aqui em vez do menor número, porque o intervalo grande entre zero e o primeiro número representável é um caso especial.

EXEMPLO

Consideremos o problema de converter $(9{,}375 \times 10^{-2})$ para notação científica na base 2. O resultado deve ter a forma $x.yy \times 2^E$. Nós começaremos convertendo a notação em ponto flutuante na base 10 para notação em ponto fixo na base 10 movendo o ponto decimal duas posições para a esquerda, o que corresponde ao expoente –2: 0,09375. Depois convertemos de ponto fixo na base 10 para um ponto fixo na base 2 usando o método da multiplicação:

$$\begin{aligned}
0{,}09375 &\times 2 = 0{,}1875 \\
0{,}1875 &\times 2 = 0{,}375 \\
0{,}375 &\times 2 = 0{,}75 \\
0{,}75 &\times 2 = 1{,}5 \\
0{,}5 &\times 2 = 1{,}0
\end{aligned}$$

desta forma $(0{,}09375)_{10} = (0{,}00011)_2$. Finalmente convertemos a um formato de ponto flutuante normalizado na base 2: $0{,}00011 = 0{,}00011 \times 2^0 = 1{,}1 \times 2^{-4}$. ■

2.3.5 O padrão para ponto flutuante IEEE 754

Existem muitas maneiras de se representar números de ponto flutuante, algumas que já foram vistas. Cada representação tem suas características em termos de intervalo de representação, precisão e o número de números representáveis. Em um esforço para melhorar a portabilidade de software e garantir uma precisão uniforme em cálculos de ponto flutuante, foi desenvolvido o padrão IEEE 754 para ponto flutuante para números binários (IEEE, 1985). Existem algumas linhas de produtos que não usam esse formato, normalmente antigas tais como IBM/370, DEC VAX ou Cray, mas praticamente todas as novas arquiteturas normalmente provêem algum nível de compatibilidade com o formato IEEE 754.

O padrão IEEE 754 da maneira que é descrito nesta seção tem que ser implementado no sistema computacional, não necessariamente somente via hardware. Ou seja, uma mistura de hardware e software pode ser usada sem perder compatibilidade com o padrão.

2.3.5.1 Formatos

Existem dois formatos primários em IEEE 754: **precisão simples** e **precisão dupla**. A Figura 2.10 sumariza os *layouts* dos dois formatos. O formato em precisão simples ocupa 32 bits, enquanto o formato de precisão dupla ocupa 64 bits. O formato em precisão dupla é simplesmente uma versão mais ampla do formato em precisão simples.

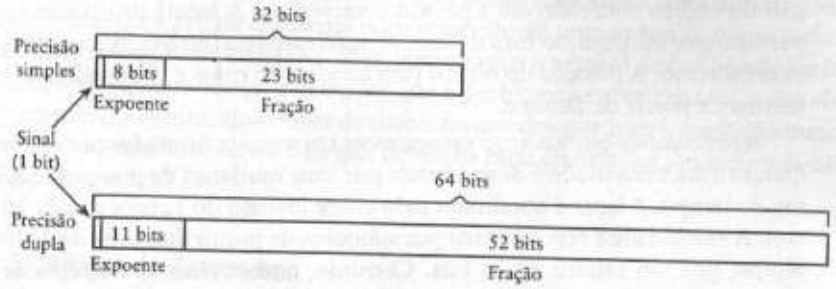


Figura 2.10 Precisão simples e dupla no formato de ponto flutuante IEEE 754.

O bit de sinal está na posição mais à esquerda, e indica um número positivo (0) ou negativo (1). Em seguida vem o expoente de oito bits em excesso de 127 (*não* 128). Neste caso os padrões de bits 00000000 e 11111111 são reservados para casos especiais, a serem descritos depois. Para precisão dupla, o expoente de 11 bits é representado em excesso de 1023, com os padrões 00000000000 e

11111111111 reservados. A fração de 23 bits na base 2 vem em seguida. Existe um bit escondido *à esquerda* do ponto binário, que quando considerado junto com a fração de precisão simples forma uma mantissa de 23 + 1 = 24 bits na forma 1.fff...f, em que o padrão fff...f representa a parte fracionária de 23 bits que é armazenada. O formato de precisão dupla também usa um bit escondido à esquerda do ponto binário, que permite uma mantissa de 52 + 1 = 53 bits. Para ambos os formatos o número é normalmente **normalizado**, sendo que exceções a esta regra serão vistas depois.

Existem cinco tipos básicos de números que podem ser representados. Números normalizados diferentes de zero são representados como descrito anteriormente. Um chamado "zero limpo" é representado pelo padrão de bits reservado 00000000 no expoente e a fração consistindo somente em zeros. O bit de sinal pode ser 0 ou 1, e então existem duas representações para zero: +0 e –0.

Infinito tem uma representação na qual o expoente contém o padrão reservado 11111111, a fração contém somente zeros, e o bit de sinal 0 ou 1. Infinito é útil no tratamento de overflow ou para dar uma representação válida para um número (diferente de zero) dividido por zero. Se zero é dividido por zero ou infinito é dividido por infinito, então o resultado é indefinido. Isto é representado pelo formato **NaN** (não é um número) em que o expoente contém o padrão de bits reservado 11111111, a fração é diferente de zero, e o bit de sinal é 0 ou 1. Um NaN pode ser também gerado ao se tentar calcular a raiz quadrada de –1.

Como em todas as representações normalizadas, existe um intervalo grande entre zero e o primeiro número representável. A representação chamada "zero sujo", não-normalizada, permite que números neste intervalo sejam representados. O bit de sinal pode ser 0 ou 1, o expoente contém o padrão de bits reservado 00000000 que representa –126 para precisão simples (–1022 para precisão dupla), e a fração contém o padrão de bits real para a magnitude do número. Deste modo, não existe nenhum 1 escondido para este formato. Note que a representação *não-normalizada* não é uma representação sem *normalização*. A diferença principal é que só existe uma representação não-normalizada para cada número, enquanto existem infinitas representações para números sem normalização.

A Figura 2.11 ilustra alguns exemplos de números de ponto flutuante IEEE 754. Os exemplos de (a) a (h) estão em formato de precisão simples e o exemplo (i) está em formato de precisão dupla. O exemplo (a) mostra a um número ordinários em precisão simples. Note que a mantissa é 1.101, mas que só a fração (101) é explicitamente representada. Exemplo (b) usa o menor expoente de precisão simples (–126) e o exemplo (c) usa o maior expoente de precisão simples (127).

Os exemplos (d) e (e) ilustram as duas representações para zero. O exemplo (f) ilustra o padrão de bits para $+\infty$. Existe também um padrão de bits correspondente para $-\infty$. O exemplo (g) mostra um número não-normalizado. Note que apesar de o número ser 2^{-128}, o menor expoente representável ainda é –126. O expoente para números não-normalizados é sempre –126, representado pelo pa-

drão de bits 00000000 e uma fração diferente de zero. A fração representa a magnitude do número em vez da mantissa. Portanto temos $+2^{-128} = +.01 \times 2^{-126}$, o que é representado pelo padrão de bits mostrado na Figura 2.11g.

	Valor	Padrão de bits		
		Sinal	Expoente	Fração
(a)	$+1.101 \times 2^5$	0	1000 0100	101 0000 0000 0000 0000 0000
(b)	-1.01011×2^{-126}	1	0000 0001	010 1100 0000 0000 0000 0000
(c)	$+1.0 \times 2^{127}$	0	1111 1110	000 0000 0000 0000 0000 0000
(d)	+0	0	0000 0000	000 0000 0000 0000 0000 0000
(e)	-0	1	0000 0000	000 0000 0000 0000 0000 0000
(f)	$+\infty$	0	1111 1111	000 0000 0000 0000 0000 0000
(g)	$+2^{-128}$	0	0000 0000	010 0000 0000 0000 0000 0000
(h)	+NaN	0	1111 1111	011 0111 0000 0000 0000 0000
(i)	$+2^{-128}$	0	011 0111 1111	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Figura 2.11 Exemplos de números de ponto flutuante IEEE 754 em precisão simples (a-h) e formato de precisão dupla. Os espaços são mostrados apesar por clareza, eles são parte da distribuição.

O exemplo (h) mostra um NaN de precisão simples. Um NaN pode ser positivo ou negativo. Finalmente, o exemplo (i) revisita a representação de 2^{-128}, desta vez usando precisão dupla. A representação é de um número ordinário em precisão dupla. Note que 2^{-128} tem uma mantissa de 1.0, o que explica por que a fração é composta somente por zeros.

Além dos formatos de precisão simples e dupla existem os formatos **simples estendido e duplo estendido**. Os formatos estendidos não são visíveis ao usuário, mas são usados para reter uma grande precisão interna durante cálculos a fim de reduzir os efeitos de erro de arredondamento. Os formatos estendidos aumentam a largura dos expoentes e frações por um número de bits que pode variar dependendo da implementação. Por exemplo, o formato simples estendido adiciona pelo menos três bits ao expoente e oito bits à fração. O formato duplo estendido tipicamente tem 80 bits de largura, com um expoente de 15 bits e uma fração de 64 bits.

2.3.5.2 Arredondamento

Uma implementação do padrão IEEE 754 deve implementar ao menos precisão simples, os outros formatos são opcionais. Além disso, o resultado de qualquer operação simples em números de ponto flutuante deve ter um erro menor que meio bit menos significativo da fração. Isto significa que bits adicionais de precisão devem ser armazenados durante o cálculo, e que existe necessidade de um método apropriado de se arredondar os resultados intermediários para o número de bits na fração.

Existem quatro modos de arredondamento no padrão IEEE 754. O primeiro arredonda para zero; o segundo, para $+\infty$; e o terceiro, para $-\infty$. O modo default arredonda para número representável mais próximo. Nos casos eqüidistantes de dois números representáveis arredonda-se para o número cujo dígito de ordem mais baixa é par. Por exemplo, 1.01101 arredonda para 1.01110, enquanto 1.01111 arredonda para 1.1000.

2.4 Estudo de Caso: A Falha do Sistema Mísseis Patriot Causado por Perda de Precisão

Durante a operação Tempestade no Deserto em 1991-1992 entre as forças de coalizão e o Iraque, uma base militar em Dhahran, na Arábia Saudita foi usada pela Coalizão. Esta base era protegida por seis baterias de mísseis Patriot. O sistema Patriot foi projetado originalmente para ser móvel e para operar continuamente somente por poucas horas, uma vez que ele deveria se mover a fim de evitar detecção.

O sistema Patriot rastreia e intercepta certos tipos de objetos, tais como mísseis Cruise ou mísseis balísticos Scud, um dos quais atingiu uma posição do exército americano em Dhahran em cinco de fevereiro de 1991, matanto 28 americanos. O sistema Patriot falhou em rastrear e interceptar o míssil Scud que se aproximava devido a uma perda de precisão na conversão entre representações inteiras e de ponto flutuante.

Um sistema de radar opera enviando uma seqüência de pulsos eletromagnéticos em diversas direções e depois ouvindo os sinais de retorno que são refletidos pelos objetos no caminho do pulso de radar. Se um objeto aéreo de interesse tal como um míssil Scud é detectado pelo sistema de radar Patriot, então a posição de uma **janela de alcance** é determinada (veja Figura 2.12), que estima a posição do objeto rastreado até a próxima varredura. A janela de alcance também permite que informação fora da área de interesse seja filtrada, o que simplifica o rastreamento. A posição do objeto (um Scud neste caso) é confirmada se estiver dentro da janela de alcance.

A previsão de onde o Scud vai aparecer em seguida é uma função da sua velocidade. Esta velocidade é determinada por uma mudança de posição com o passar do tempo. A hora é atualizada pelo clock interno do Patriot a cada 100 metros. A velocidade é representada por números de ponto flutuante de 24 bits; e o tempo, por um inteiro de 24 bits. Contudo, ambos têm de ser representados como números de ponto flutuante de 24 bits a fim de se prever onde o Scud vai aparecer.

A conversão do tempo de inteiro para ponto flutuante resulta em perda de precisão que aumenta à medida que o tempo representado internamente aumenta. O erro introduzido pela conversão resulta em um erro no cálculo da janela de alcance, que é proporcional à velocidade do alvo e o tempo total que o sistema está executando. A causa do incidente de Dhahran, ocorrida depois que o siste-

ma estava operando continuamente por mais de 100 horas é que a janela de alcance deslocou 687 metros, resultando na interceptação falha do Scud.

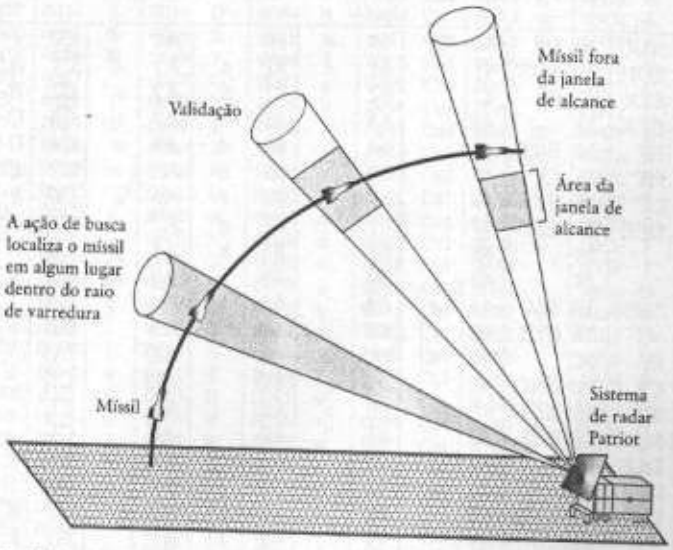


Figura 2.12 Efeito do erro de conversão no cálculo da janela de alcance.

O problema de conversão foi descoberto duas semanas antes do incidente em Dhahran como um resultado de dados gerados por Israel, mas o novo software que o corrigiu chegou somente no dia seguinte ao ataque devido à dificuldade de se distribuir correções de software em um ambiente de guerra. Uma solução para o problema, enquanto o novo software não estava disponível, teria sido simplesmente desligar e religar o sistema num intervalo de poucas horas, o que teria o efeito de resetar o clock interno. Mas, uma vez que o pessoal de campo não foi informado de quanto tempo gastava até que o problema se manifestasse, o que efetivamente era conhecido através de dados fornecidos por Israel, a solução nunca foi implementada. A lição é a de que devemos estar cientes das limitações de cálculos de precisão finita.

2.5 Códigos de Caracteres

Ao contrário dos números reais que têm um domínio infinito, existe somente um número finito de caracteres. Um conjunto completo de caracteres pode ser representado com um número pequeno de bits por caracter. Três das mais comuns representações de caracteres, ASCII, EBCDIC e Unicode são descritas a seguir.

2.5.1 O conjunto de caracteres ASCII

A representação denominada American Standard Code for Information Interchange (ASCII) é resumida na Figura 2.13, usando índices hexadecimais. A representação para cada caracter consiste em 7 bits, e todos os 2^7 padrões de bits representam caracteres válidos. Os caracteres nas posições 00-1F e posição 7F são caracteres especiais de controle que são usados para transmissão, controle de impressão e outros objetivos não-textuais. Os caracteres restantes podem todos ser impressos e incluem letras, números, pontuação e um espaço. Os dígitos 0-9 aparecem em seqüência, assim como as letras minúsculas e maiúsculas.[4] Esta organização simplifica a manipulação de caracteres. A fim de mudar a representação dos caracteres de um dígito em seu numérico equivalente, devemos subtrair $(30)_{16}$ do número. A fim de se converter o caracter ASCII "5", que está na posição $(35)_{16}$ no número 5, calculamos $(35\text{-}30\text{=}5)_{16}$. A fim de se converter uma letra maiúscula em uma letra minúscula somamos $(20)_{16}$. Por exemplo, para converter a letra "H" que está na localização $(48)_{16}$ na tabela ACII na letra "h" que está na posição $(68)_{16}$, calculamos $(48 + 20 = 68)_{16}$.

2.5.2 O conjunto de caracteres EBCDIC

Um problema com o código ASCII é que somente 128 caracteres podem ser representados, o que é uma limitação para muitos teclados que têm muitos caracteres especiais além das letras maiúsculas e minúsculas. O código Extended Binary Coded Decimal Interchange Code (EBCDIC) é um código de oito bits que é usado extensivamente pela IBM que não sofre desta restrição. Uma vez que caracteres ASCII de sete bits são freqüentemente representados em um formato modificado de oito bits (um caracter por byte), no qual um 0 ou um 1 é adicionado à esquerda do padrão de sete bits, o uso do EBCDIC não causa um aumento no espaço de armazenamento de caracteres em um computador. Para transmissão serial, contudo, um código de oito bits gasta mais tempo para ser transmitido que um código de sete bits, e neste caso o código mais amplo faz diferença.

O código EBCDIC é sumarizado na Figura 2.14. Existem espaços vazios na tabela, que podem ser usados para caracteres específicos de cada aplicação. O fato de que existem espaços vazios em seqüências de letras maiúsculas e minúsculas não causa uma grande desvantagem porque as manipulações de caracteres podem ser feitas da mesma forma que para ASCII, mas usando deslocamentos diferentes.

2.5.3 O conjunto de caracteres Unicode

Os códigos ASCII e EBCDIC representam conjuntos de caracteres historicamente dominantes (latinos). Existem muitos outros conjuntos de caracteres no mun-

[4] Note que o caracter "a" e o caracter "A" são diferentes e têm códigos diferentes na tabela ASCII. As letras minúsculas como "a" são chamadas **caixa-baixa** (*lower case*) e as letras maiúsculas como "A" são chamadas de **caixa-alta** (*upper case*). A nomenclatura vem das posições dos caracteres em uma máquina de escrever. As letras maiúsculas aparecem acima das letras minúsculas, o que resultou na nomenclatura **caixa-alta** e caixa-baixa. Hoje em dia, a tipografia é quase sempre feita eletronicamente, mas a nomenclatura ainda é usada.

do, e um simples mapeamento ASCII-para-linguagem-X não funciona no caso geral, e portanto um novo padrão universal de caracteres foi desenvolvido para representar um número maior de conjuntos de caracteres chamado **Unicode**.

00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

NUL	Null	FF	Form feed	CAN	Cancel
SOH	Start of heading	CR	Carriage return	EM	End of medium
STX	Start of text	SO	Shift out	SUB	Substitute
ETX	End of text	SI	Shift in	ESC	Escape
EOT	End of transmission	DLE	Data link escape	FS	File separator
ENQ	Enquiry	DC1	Device control 1	GS	Group separator
ACK	Acknowledge	DC2	Device control 2	RS	Record separator
BEL	Bell	DC3	Device control 3	US	Unit separator
BS	Backspace	DC4	Device control 4	SP	Space
HT	Horizontal tab	NAK	Negative acknowledge	DEL	Delete
LF	Line feed	SYN	Synchronous idle		
VT	Vertical tab	ETB	End of transmission block		

Figura 2.13 O código de caracteres ASCII mostrado com índices hexadecimais.

Unicode é um padrão em desenvolvimento. Ele é modificado à medida que novos conjuntos de caracteres são acrescentados, e à medida que conjuntos de caracteres existentes evoluem e suas representações são refinadas. Na versão 2.0 do padrão Unicode existem 38.885 caracteres distintos codificados que abordam as principais linguagens escritas da América, Europa, Oriente Médio, África, Índia, Ásia e Oceania.

O padrão Unicode usa um conjunto de códigos de 16 bits no qual existe uma correspondência de um para um entre os códigos de 16 bits e caracteres. De forma semelhante ao ASCII, não existem modos complexos ou códigos de escape. Muito embora o Unicode represente mais caracteres do que ASCII ou EBCDIC, ele não é a solução para todos os problemas. Na verdade, o padrão Unicode de 16 bits é um subconjunto do padrão de 32 bits 32-bit ISO 10646 Universal Character Set (UCS –4).

Os primeiros 256 caracteres Unicode são mostrados na Figura 2.15, de acordo com a versão Unicode 2.1. Note que os primeiros 128 caracteres são os mesmos que para ASCII.

00	NUL	20	DS	40	SP	60	–	80		A0		C0	{	E0	\
01	SOH	21	SOS	41		61	/	81	a	A1	~	C1	A	E1	
02	STX	22	FS	42		62		82	b	A2	s	C2	B	E2	S
03	ETX	23		43		63		83	c	A3	t	C3	C	E3	T
04	PF	24	BYP	44		64		84	d	A4	u	C4	D	E4	U
05	HT	25	LF	45		65		85	e	A5	v	C5	E	E5	V
06	LC	26	ETB	46		66		86	f	A6	w	C6	F	E6	W
07	DEL	27	ESC	47		67		87	g	A7	x	C7	G	E7	X
08		28		48		68		88	h	A8	y	C8	H	E8	Y
09		29		49		69		89	i	A9	z	C9	I	E9	Z
0A	SMM	2A	SM	4A	¢	6A	`	8A		AA		CA		EA	
0B	VT	2B	CU2	4B		6B	,	8B		AB		CB		EB	
0C	FF	2C		4C	<	6C	%	8C		AC		CC		EC	
0D	CR	2D	ENQ	4D	(	6D	_	8D		AD		CD		ED	
0E	SO	2E	ACK	4E	+	6E	>	8E		AE		CE		EE	
0F	SI	2F	BEL	4F	|	6F	?	8F		AF		CF		EF	
10	DLE	30		50	&	70		90		B0		D0	}	F0	0
11	DC1	31		51		71		91	j	B1		D1	J	F1	1
12	DC2	32	SYN	52		72		92	k	B2		D2	K	F2	2
13	TM	33		53		73		93	l	B3		D3	L	F3	3
14	RES	34	PN	54		74		94	m	B4		D4	M	F4	4
15	NL	35	RS	55		75		95	n	B5		D5	N	F5	5
16	BS	36	UC	56		76		96	o	B6		D6	O	F6	6
17	IL	37	EOT	57		77		97	p	B7		D7	P	F7	7
18	CAN	38		58		78		98	q	B8		D8	Q	F8	8
19	EM	39		59		79		99	r	B9		D9	R	F9	9
1A	CC	3A		5A	!	7A	:	9A		BA		DA		FA	¦
1B	CU1	3B	CU3	5B	$	7B	#	9B		BB		DB		FB	
1C	IFS	3C	DC4	5C	*	7C	@	9C		BC		DC		FC	
1D	IGS	3D	NAK	5D	)	7D	'	9D		BD		DD		FD	
1E	IRS	3E		5E	;	7E	=	9E		BE		DE		FE	
1F	IUS	3F	SUB	5F	¬	7F	"	9F		BF		DF		FF	

STX	Start of text	RS	Reader Stop	DC1	Device Control 1	BEL	Bell
DLE	Data Link Escape	PF	Punch Off	DC2	Device Control 2	SP	Space
BS	Backspace	DS	Digit Select	DC4	Device Control 4	IL	Idle
ACK	Acknowledge	PN	Punch On	CU1	Customer Use 1	NUL	Null
SOH	Start of Heading	SM	Set Mode	CU2	Customer Use 2		
ENQ	Enquiry	LC	Lower Case	CU3	Customer Use 3		
ESC	Escape	CC	Cursor Control	SYN	Synchronous Idle		
BYP	Bypass	CR	Carriage Return	IFS	Interchange File Separator		
CAN	Cancel	EM	End of Medium	EOT	End of Transmission		
RES	Restore	FF	Form Feed	ETB	End of Transmission Block		
SI	Shift In	TM	Tape Mark	NAK	Negative Acknowledge		
SO	Shift Out	UC	Upper Case	SMM	Start of Manual Message		
DEL	Delete	FS	Field Separator	SOS	Start of Significance		
SUB	Substitute	HT	Horizontal Tab	IGS	Interchange Group Separator		
NL	New Line	VT	Vertical Tab	IRS	Interchange Record Separator		
LF	Line Feed	UC	Upper Case	IUS	Interchange Unit Separator		

Figura 2.14 O código de caracteres EBCDIC mostrado com índices hexadecimais.

0000	NUL	0020	SP	0040	@	0060	`	0080	Ctrl	00A0	NBS	00C0	À	00E0	à
0001	SOH	0021	!	0041	A	0061	a	0081	Ctrl	00A1	¡	00C1	Á	00E1	á
0002	STX	0022	"	0042	B	0062	b	0082	Ctrl	00A2	¢	00C2	Â	00E2	â
0003	ETX	0023	#	0043	C	0063	c	0083	Ctrl	00A3	£	00C3	Ã	00E3	ã
0004	EOT	0024	$	0044	D	0064	d	0084	Ctrl	00A4	¤	00C4	Ä	00E4	ä
0005	ENQ	0025	%	0045	E	0065	e	0085	Ctrl	00A5	¥	00C5	Å	00E5	å
0006	ACK	0026	&	0046	F	0066	f	0086	Ctrl	00A6	¦	00C6	Æ	00E6	æ
0007	BEL	0027	'	0047	G	0067	g	0087	Ctrl	00A7	§	00C7	Ç	00E7	ç
0008	BS	0028	(	0048	H	0068	h	0088	Ctrl	00A8	¨	00C8	È	00E8	è
0009	HT	0029	)	0049	I	0069	i	0089	Ctrl	00A9	©	00C9	É	00E9	é
000A	LF	002A	*	004A	J	006A	j	008A	Ctrl	00AA	ª	00CA	Ê	00EA	ê
000B	VT	002B	+	004B	K	006B	k	008B	Ctrl	00AB	«	00CB	Ë	00EB	ë
000C	FF	002C	,	004C	L	006C	l	008C	Ctrl	00AC	¬	00CC	Ì	00EC	ì
000D	CR	002D	-	004D	M	006D	m	008D	Ctrl	00AD	-	00CD	Í	00ED	í
000E	SO	002E	.	004E	N	006E	n	008E	Ctrl	00AE	®	00CE	Î	00EE	î
000F	SI	002F	/	004F	O	006F	o	008F	Ctrl	00AF	¯	00CF	Ï	00EF	ï
0010	DLE	0030	0	0050	P	0070	p	0090	Ctrl	00B0	°	00D0	Ð	00F0	ð
0011	DC1	0031	1	0051	Q	0071	q	0091	Ctrl	00B1	±	00D1	Ñ	00F1	ñ
0012	DC2	0032	2	0052	R	0072	r	0092	Ctrl	00B2	²	00D2	Ò	00F2	ò
0013	DC3	0033	3	0053	S	0073	s	0093	Ctrl	00B3	³	00D3	Ó	00F3	ó
0014	DC4	0034	4	0054	T	0074	t	0094	Ctrl	00B4	´	00D4	Ô	00F4	ô
0015	NAK	0035	5	0055	U	0075	u	0095	Ctrl	00B5	µ	00D5	Õ	00F5	õ
0016	SYN	0036	6	0056	V	0076	v	0096	Ctrl	00B6	¶	00D6	Ö	00F6	ö
0017	ETB	0037	7	0057	W	0077	w	0097	Ctrl	00B7	·	00D7	×	00F7	÷
0018	CAN	0038	8	0058	X	0078	x	0098	Ctrl	00B8	¸	00D8	Ø	00F8	ø
0019	EM	0039	9	0059	Y	0079	y	0099	Ctrl	00B9	¹	00D9	Ù	00F9	ù
001A	SUB	003A	:	005A	Z	007A	z	009A	Ctrl	00BA	º	00DA	Ú	00FA	ú
001B	ESC	003B	;	005B	[	007B	{	009B	Ctrl	00BB	»	00DB	Û	00FB	û
001C	FS	003C	<	005C	\	007C	|	009C	Ctrl	00BC	1/4	00DC	Ü	00FC	ü
001D	GS	003D	=	005D	]	007D	}	009D	Ctrl	00BD	1/2	00DD	Ý	00FD	ý
001E	RS	003E	>	005E	^	007E	~	009E	Ctrl	00BE	3/4	00DE	Þ	00FE	þ
001F	US	003F	?	005F	_	007F	DEL	009F	Ctrl	00BF	¿	00DF	ß	00FF	ÿ

NUL	Null	SOH	Start of heading	CAN	Cancel	SP	Space
STX	Start of text	EOT	End of transmission	EM	End of medium	DEL	Delete
ETX	End of text	DC1	Device control 1	SUB	Substitute	Ctrl	Control
ENQ	Enquiry	DC2	Device control 2	ESC	Escape	FF	Form feed
ACK	Acknowledge	DC3	Device control 3	FS	File separator	CR	Carriage return
BEL	Bell	DC4	Device control 4	GS	Group separator	SO	Shift out
BS	Backspace	NAK	Negative acknowledge	RS	Record separator	SI	Shift in
HT	Horizontal tab	NBS	Non-breaking space	US	Unit separator	DLE	Data link escape
LF	Line feed	ETB	End of transmission block	SYN	Synchronous idle	VT	Vertical tab

Figura 2.15 Os primeiros 256 caracteres Unicode mostrado com índices hexadecimais.

■ RESUMO

Todos os dados em um computador são representados em termos de bits, os quais podem ser organizados e interpretados como inteiros, números de ponto fixo, números de ponto flutuante ou caracteres. Códigos de caracteres tais como ASCII, EBCDIC e Unicode têm tamanhos finitos e podem portanto ser representados completamente em um número finito de bits. O número de bits usado para representar números também é finito, e como resultado somente um subconjunto dos números reais podem ser representados. Isto leva a noções de intervalo de representação, precisão e erro. O intervalo de representação define a maior e a menor magnitudes que podem ser representadas e é totalmente determinada pela base e pelo número de bits no expoente da representação de ponto flutuante. A precisão é determinada pelo número de bits usado na representação da magnitude (excluindo os bits de expoente em uma representação de ponto flutuante). Erros aparecem em representações de ponto flutuante porque existem números reais que estão localizados em espaços vazios entre números representáveis adjacentes.

■ LEITURAS ADICIONAIS

Knuth (1981) apresenta um tratamento completo de aritmética de computadores e seus algoritmos. Hamacher *et al.* (1990) tem uma boa discussão sobre erros não-aleatórios em representações de ponto flutuante. O padrão IEEE 754 para ponto flutuante é descrito em IEEE (1985). A análise de intervalo, erro e precisão da Seção 2.3 foi influenciada por Forsythe (1970). O relatório GAO (U.S. GAO report GAO/IMTEC-92-26) apresenta um relato detalhado do problema de software que levou à falha de Dhahran. Veja http://www.unicode.org para informação no padrão Unicode.

Forsythe, G. E., *"Pitfalls in Computation, or Why a Math Book Isn't Enough"*, The American Mathematical Monthly, 77, n. 9, pp. 931-956 (novembro de 1970).

Hamacher, V. C., Z. G. Vranesic e S. G. Zaky, *Computer Organization*, 3ª ed., McGraw Hill (1990).

IEEE, *"IEEE Standard for Binary Floating Point Arithmetic"*, ANSI/IEEE Standard 754-1985. Com uma versão anterior em IEEE COMPUTER, 14, pp. 51-62 (março de 1981).

Knuth, D. E. *The Art of Computer Programming, vol. 2, Seminumerical algorithms*, 2ª ed., Addison-Wesley-Longman (1981).

U.S. General Accounting Office report GAO/IMTEC-92-26, *"Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia"*, U.S. General Accounting Office, P.O. Box 6015, Gaithersburg, Maryland, 20877 (fevereiro de 1992).

■ PROBLEMAS

2.1 Dada uma representação com sinal e em ponto fixo na base 10, com três dígitos à esquerda e à direita do ponto decimal,

(a) Qual seu intervalo de representação? (Calcule o maior número positivo e o menor número negativo.)

(b) Qual é a precisão? (Calcule a diferença entre dois números adjacentes. Lembre-se de que o erro é de metade da precisão.)

2.2 Converta os seguintes números como indicado, usando o menor número de dígitos possível para o resultado

(a) $(47)_{10}$ para um binário sem sinal

(b) $(-27)_{10}$ para binário em sinal magnitude

(c) $(213)_{16}$ para base 10

(d) $(10110.101)_2$ para base 10

(e) $(34.625)_{10}$ para base 4

2.3 Converta os seguintes números como indicado, usando o menor número de dígitos possível para o resultado.

(a) $(011011)_2$ para base 10

(b) $(-27)_{10}$ para binário em excesso de 32

(c) $(011011)_2$ para base 16

(d) $(55.875)_{10}$ para binário sem sinal

(e) $(132.2)_4$ para base 16

2.4 Converta $.201_3$ para decimal

2.5 Converta $(43.3)_7$ para base 8 usando não mais do que um dígito octal à direita do ponto. Trunque qualquer resto eliminando dígitos excedentes. Use uma representação octal comum sem sinal.

2.6 Represente $(17.5)_{10}$ na base 3, e então converta o resultado de volta à base 10. Use dois dígitos de precisão à direita do ponto para a representação intermediária na base 3.

2.7 Encontre o equivalente decimal do número de quatro bits em complemento de dois 1000.

2.8 Ache o equivalente decimal do número de quatro bits em complemento de um 1111.

2.9 Encontre a representação para $(305)_{10}$ usando três dígitos BCD.

2.10 Demonstre o complemento de dez de $(-305)_{10}$ usando três dígitos BCD.

2.11 Para um dado número de bits, existem mais números inteiros representáveis em complemento de um ou de dois? Ou o número é o mesmo?

2.12 Complete a tabela a seguir para representações de 5 bits (incluindo bits de sinal) como indicado. Mostre suas respostas como inteiros com sinal na base 10.

	5 Bits em Sinal Magnitude	5 Bits em Excesso de 16
Maior número		
Número mais negativo		
Número de números distintos		

2.13 Complete a tabela a seguir usando notação científica de base 2 e uma representação de oito bits de ponto flutuante na qual existe um expoente de três bits em excesso de 3 (não excesso de 4) e uma fração normalizada de quatro bits com um 1 escondido. Nesta representação o 1 escondido está à esquerda do ponto. Isto significa que o número 1,0101 está em forma normalizada, enquanto 0,101 não está.

Notação Científica de Base 2	Representação de Ponto Flutuante		
	Sinal	Expoente	Fração
$-1{,}0101 \times 2^{-2}$			
$+1{,}1 \times 2^{2}$			
	0	001	0000
	1	110	1111

2.14 A representação em ponto flutuante curta da IBM usa base 16, um bit de sinal, um expoente de sete bits em excesso de 64 e uma fração normalizada de 24 bits.

(a) Qual número é representado pelo seguinte padrão de bits?

1 0111111 01110000 00000000 00000000

Mostre sua resposta em decimal. Note que os espaços são incluídos no número somente para melhorar a legibilidade.

(b) Represente $(14.3)_6$ nesta notação

2.15 Para uma representação normalizada de ponto flutuante, mantendo todo o resto sem modificações mas

(a) diminuindo a base vai aumentar/diminuir/não alterar o número de números representáveis.

(b) aumentando o número de dígitos significativos vai aumentar/diminuir/não alterar o menor número positivo representável.

(c) aumentando o número de bits no expoente vai aumentar/diminuir/não alterar o intervalo de representação.

(d) modificando a representação do expoente de excesso de 64 para complemento de dois vai aumentar/diminuir/não alterar o intervalo de representação.

2.16 Para as partes (a) até (e) use uma representação de ponto flutuante com bit de sinal na posição mais à esquerda, seguido de um expoente de dois bits em complemento de dois, seguido por uma fração normalizada de três bits na base 2. O zero é representado pelo padrão de bits 0 0 0 0 0 0. Não existe um 1 escondido.

(a) Qual o número decimal representado pelo padrão de bits 100100?

(b) Mantendo todo o resto sem alterações mas mudando a base para 4 vai aumentar/diminuir/não alterar o menor número positivo representável?

(c) Qual é o menor espaço vazio entre números sucessivos?

(d) Qual é o maior espaço vazio entre números sucessivos?

(e) Existe um total de seis bits nesta representação em ponto flutuante e existem $2^6 = 64$ padrões de bits diferentes. Quantos destes padrões de bits são válidos?

2.17 Represente $(107.15)_{10}$ em uma representação de ponto flutuante com bit de sinal, um expoente de sete bits em excesso de 64 e uma fração normalizada de 24 bits na base 2. Não existe um 1 escondido. Trunque a fração eliminando bits quando necessário.

2.18 Para os seguintes padrões de bit IEEE 754 em precisão simples mostre o valor numérico com uma mantissa na base 2 e um expoente (por exemplo, $f1.11 \times 2^5$)

(a) 0 10000011 01100000000000000000000

(b) 1 10000000 00000000000000000000000

(c) 1 00000000 00000000000000000000000

(d) 1 11111111 00000000000000000000000

(e) 0 11111111 11010000000000000000000

(f) 0 00000001 10010000000000000000000

(g) 0 00000011 01101000000000000000000

2.19 Mostre os padrões de bits IEEE 754 para os seguintes números:

(a) $+1.1011 \times 2^5$ (precisão simples)

(b) +0 (precisão simples)

(c) -1.00111×2^{-1} (precisão dupla)

(d) –NaN (precisão simples)

2.20 Usando o formato IEEE 754 em precisão simples, mostre o valor (não o padrão de bits) de:

(a) O maior número positivo representável (nota: ∞ não é um número).

(b) O menor número positivo diferente de zero em forma normalizada

(c) O menor número positivo diferente de zero em forma não-normalizada

(d) O menor espaço vazio normalizado

(e) O maior espaço vazio normalizado

(f) O número de números normalizados representáveis (incluindo 0; note que ∞ e NaN não são números)

2.21 Dois programadores escreveram geradores de números aleatórios para números normalizados em ponto flutuante usando o mesmo método. O gerador do programador A cria números aleatórios no intervalo fechado de 0 a 0.5 e o gerador do programador *B* no intervalo fechado de 0.5 a 1. O gerador do programador *B* funciona corretamente, mas o gerador do programador A produz uma distribuição tendenciosa dos números. O que pode ser o problema com o enfoque do programador A?

2.22 Um 1 escondido não funciona na base 16. Por quê?

2.23 Com uma representação com um 1 escondido pode-se representar 0 se todos os padrões de bits possíveis no expoente e na fração são usados para números diferentes de zero?

2.24 Dado um número de ponto flutuante na base 10 (por exemplo, $0{,}583 \times 10^3$), este número pode ser convertido em uma forma equivalente na base 2: $0{,}x \times 2^y$ separando a conversão da fração (0,583) do expoente (3) na base 2?