Tutorial - Representação de números positivos e negativos em base 2

Prof. Regis Rossi A. Faria
USP

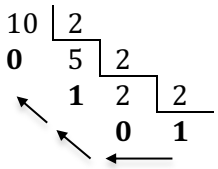
As bases numéricas são completas no sentido em que podem representar números positivos e negativos, sejam inteiros ou reais (incluindo os números irracionais ou fracionários). Além da base decimal a que estamos acostumados, a base binária assume grande importância na medida em que é utilizada largamente na conversão de sinais analógicos para os formatos digitais. Nos cursos de computação sonora e tecnologia musical se faz importante estudarmos como os valores das amplitudes de sinais acústicos podem ser convertidos em números binários.

Na base 10 (decimal) temos 10 símbolos diferentes possíveis para usar em cada casa, que representam os números de zero a nove, são eles: 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9. O número 10 já irá usar novamente o "0" na casa de unidades e leva "1" para a próxima casa (no caso, a casa das dezenas), e assim por diante para as casas de ordem superior.

Na base 2 (binária), com que os sistemas digitais trabalham, temos para cada casa 2 símbolos (ou dígitos) possíveis: *0* e *1*. Cada dígito na base 2 recebe o nome de *bit*.

Convertendo de decimal para binário e vice-versa

A *conversão de números decimais positivos inteiros* (ex: 0, 1, 2, ...) *para binário* pode ser feita dividindo-se o número X por 2 consecutivamente, enquanto o quociente Q for maior ou igual a 2 (isto é, $Q \geq 2$). Quando o quociente for menor que 2 então paramos a divisão e coletamos os números gerados no processo, começando do último *quociente* e coletando os *restos* de cada divisão da direita para esquerda. A sequência de bits resultante representará o número X convertido em base 2. Vejamos um pequeno exemplo: converta X=10 para binário.



O número binário obtido pela divisão consecutiva é "1010". De fato, podemos verificar se 1010_2 (isto é, 1010 na base 2) é realmente 10_{10} (isto é, 10 na base 10): para tanto fazemos a *conversão do número binário para decimal* multiplicando-se o valor da casa do dígito pelo próprio dígito e somando-se estas parcelas, assim:

valor da casa -->	2^3	2^2	2^1	2^0
número binário -->	1	0	1	0
bit x valor da casa -->	$1x2^3=8$	0	$1x2^1=2$	0
	8 +	0 +	2 +	0 = 10_{10}

Para números decimais positivos inteiros sempre vamos precisar de N bits para representar 2^N números. Por exemplo, para representar 4 números precisamos só de 2 bits (pois $2^2 = 4$) que vão gerar os números 00 ($=0_{10}$), 01 ($=1_{10}$), 10 ($=3_{10}$) e 11 ($=2_{10}$).

Para converter por exemplo o número 120_{10} em binário precisamos determinar *quantos bits serão necessários* para isso. Vejamos a tabela abaixo que mostra quantos números são representáveis com N bits para vários valores de N:

com N bits	*representamos 2^N números*	*o que dá p/ representar*
2	*4*	de 0 a 3 ou de 1 a 4
3	*8*	de 0 a 7 ou de 1 a 8
n	*2^n*	*de 0 a (2^n -1) ou de 1 a 2^n*
6	*64*	de 0 a 63 ou de 1 a 64
7	*128*	de 0 a 127 ou de 1 a 128
8	*256*	de 0 a 255 ou de 1 a 258

A forma mais usual é considerar o zero como o primeiro número, assim só sobrarão mais *(2^n - 1)* números para se representar com *n* bits. Por exemplo, com 3 bits representamos de 0 a 7 (oito números: 0, 1, 2, 3, 4, 5, 6, e 7), o número 8 não dará mais para representar. Para representarmos o 8 precisaremos de 4 bits, que tem a capacidade de representar até 16 números (isto é, de 0 a 15).

Por extensão deste raciocínio, percebemos que para representar o número 120_{10} somente 7 bits serão suficientes (6 bits não dariam conta do recado, pois só podemos representar até 64 números com esta quantidade de bits). Similarmente ao exemplo acima, para representar o número 127_{10} ainda são suficientes 7 bits, mas já para representar 128_{10} vamos precisar de um bit a mais, isto é, de 8 bits.

Representando números negativos

Vimos que, embora com 3 dígitos seja possível representar até $2^3 = 8$ valores, não temos como representar +8 com 3 dígitos. Os oito valores serão 000, 001, 010, 011, 100, 101, 110 e 111, isto é, o primeiro número será o zero e o oitavo número será o +7. Portanto precisamos de 4 bits, e assim +8 na base 2 será 1000_2.

Agora, para representar números negativos precisamos *reservar um bit para representar o sinal*. A convenção mais comum é usar o *bit mais significativo* para isso, isto é, o bit que está mais à esquerda. Para indicar valores positivos atribuímos ao bit de sinal o valor "0" e para indicar valores negativos fazemos este bit = 1. Embora a adição de um bit a mais (para o sinal) possa resolver o problema, a solução mais comum e elegante do ponto de vista aritmético é usar uma forma de codificação diferenciada chamada ***complemento de dois***, muito utilizada para representar números binários em computadores e sistemas digitais.

Para representar uma faixa de valores que vá de $+8_{10}$ até -8_{10} em complemento de dois precisaremos mais do que 4 dígitos! Isso porque em complemento de dois o bit mais à esquerda (o bit mais significativo) é usado para representar o sinal somente e *não representa valor*.

Portanto, em complemento de dois, se tivermos N bits, sabemos que 1 bit exprimirá o sinal, e sobram N-1 bits para representar a faixa de valores que queremos representar. Uma característica vantajosa desta representação é que no final das contas N bits ainda representam 2^N valores, entre números positivos e negativos, tendo a faixa de representação variando de

$$-(2^{N-1}) \text{ até } +(2^{N-1} - 1).$$

Por exemplo, com 4 bits teremos $2^4 = 16$ valores representáveis: o bit mais à esquerda será o bit de sinal e os 3 bits restantes são usados para representar 8 valores positivos (incluindo o zero) e 8 negativos. Representaremos portanto de -8 a +7 com 4 bits em complemento de dois. Os 8 valores positivos são (incluindo-se o bit 0 de sinal + à esquerda):

0000 (representando o zero),
0001 (representando 1_{10}),

0010 (2_{10}),
0011 (3_{10}),
0100 (4_{10})
....., até
0111 (+7)

Os 8 valores negativos são obtidos de uma forma muito simples: pegue o valor binário positivo, *inverta todos os bits*[1] e *some +1*, *o resultado será o valor negativo em complemento de dois*.

> Exemplo: $+1_{10}$ = 0001 --> invertendo dá 1110 e somando-se +1 dá 1111 (este será o equivalente binário de -1_{10}). Da mesma forma +2 = 0010 --> 1101 + 1 = 1110 (que equivale a -2_{10}), e assim por diante, até +7 = 0111 --> invertendo dá 1000 e somando-se 1 dá 1001 (-7_{10}).

Assim obtemos a faixa de números negativos representada por:

1111 (-1_{10}),
1110 (-2_{10}),
1101 (-3_{10})
1100 (-4_{10})
1011 (-5_{10})
1010 (-6_{10})
1001 (-7_{10})

Observe que temos 7 números negativos + 7 números positivos + o zero, um total de 15 valores representados desta forma. Mas com 4 bits podemos representar 16 valores!... Analisando de perto observamos que o número "1000" não foi ainda usado. Por ter o bit mais significativo "1" este é um número negativo, e ele será o número $-(2^{N-1}) = -(2^{4-1}) = -(2^3) = -8_{10}$. Para verificar, tomamos o número binário inteiro +8 sem complemento de 2 (que é 1000) e usamos a mesma operação para revelar qual seria o valor negativo em complemento de dois: invertemos o número (dá 0111) e somamos +1 (0111 + 1 = 1000) que leva exatamente ao número faltante. Assim completamos todos os 16 números incluindo este:

1000 (-8_{10})

Pelo nosso exemplo acima concluímos que com 4 bits podemos representar em complemento de dois a seguinte faixa de valores:

1000 (-8_{10}), 1001 (-7_{10}), ... , 1111 (-1_{10}), 0000, 0001 ($+1_{10}$), ... até 0111 ($+7_{10}$).

Não podemos portanto representar $+8_{10}$ ... Para representar 8 valores positivos + 8 valores negativos + o zero precisamos de no mínimo 5 dígitos binários! Isto porque 8+8+1=17 valores a representar, e 2^4 dão somente 16 valores representáveis. Com 5 bits temos que 2^5= 32 valores representáveis: de -16 a +15 (no total 32 valores).

Resumo da ópera: desta forma para representar um certo número X_{10} em complemento de dois temos que descobrir *quantos N bits são necessários para cobrir a faixa de valores que engloba X,* sabendo-se que *N bits engloba a faixa de* $-(2^{N-1})$ *a* $+(2^{N-1} - 1)$.

Exemplos:

a) X = +15

[1] Isto é chamado de obter o complemento do número.

p/ N=3 $\Rightarrow -(2^{3-1})$ a $+(2^{3-1}-1) \Rightarrow -(2^2)$ a $+(2^2-1) \Rightarrow -(4)$ a $+(3)$ (não dá)
p/ N=4 $\Rightarrow -(2^{4-1})$ a $+(2^{4-1}-1) \Rightarrow -(2^3)$ a $+(2^3-1) \Rightarrow -(8)$ a $+(7)$ (não dá)
p/ N=5 $\Rightarrow -(2^{5-1})$ a $+(2^{5-1}-1) \Rightarrow -(2^4)$ a $+(2^4-1) \Rightarrow -(16)$ a $+(15)$ (dá) √

Portanto com 5 bits dá para representar +15 em complemento de dois.

b) X = -32

p/ N=6 $\Rightarrow -(2^{6-1})$ a $+(2^{6-1}-1) \Rightarrow -(2^5)$ a $+(2^2-1) \Rightarrow -(32)$ a $+(31)$ (dá) √

Com 6 bits dá para representar -32 em complemento de dois.

c) X = +32

p/ N=6 $\Rightarrow -(2^{6-1})$ a $+(2^{6-1}-1) \Rightarrow -(2^5)$ a $+(2^2-1) \Rightarrow -(32)$ a $+(31)$ (não dá)
p/ N=7 $\Rightarrow -(2^{7-1})$ a $+(2^{7-1}-1) \Rightarrow -(2^6)$ a $+(2^6-1) \Rightarrow -(64)$ a $+(63)$ (dá) √

Com 7 bits dá para representar +32 em complemento de dois.

Convertendo números negativos em complemento de dois para decimal

É importante notar que em complemento de dois os números negativos não podem ser convertidos para a base 10 por aquele processo de pegar o bit e multiplicar pelo valor da casa, exatamente porque o bit mais significativo não mais representa valor (isto é, a casa mais à esquerda não tem o valor 2^N). Por exemplo o número em complemento de dois 1001 equivale ao decimal -7_{10}, mas se usarmos simplesmente aquele processo de conversão para este número negativo vamos obter que ele representa $1x2^3 + 0x2^2 + 0x2^1 + 1x2^0 = 8 + 1 = 9_{10}$ ...

Portanto, se o número está em complemento de dois e é negativo (começa com "1") precisamos primeiro obter seu recíproco positivo (que começa com "0") e aí sim poderemos aplicar o processo de conversão para decimal. Assim, obtemos o recíproco de 1001 complementando-o (invertendo-se os bits) e somando +1, o que dará 0110 +1 = 0111. Agora convertemos este número positivo diretamente para decimal: $0x2^3 + 1x2^2 + 1x2^1 + 1x2^0 = 4 + 2 + 1 = 7$. Portanto o número negativo era -7!

Com este conhecimento agora podemos então montar uma tabela com os valores binários em complemento de dois usando-se 5 bits, que dá para representar de -16 até +15:

no. na base 10	*na base 2*	*cálculo que leva o número base 2 à base 10*
+15	01111	$0.2^4 + 1.2^3 + 1.2^2 + 1.2^1 + 1.2^0 = 15$
...	...	...
+8	01000	$0.2^4 + 1.2^3 + 0.2^2 + 0.2^1 + 0.2^0 = 8$
+7	00111	$0.2^4 + 0.2^3 + 1.2^2 + 1.2^1 + 1.2^0 = 7$
...	...	...
1	00001	$0.2^4 + 0.2^3 + 0.2^2 + 0.2^1 + 1.2^0 = 1$
0	00000	$0.2^4 + 0.2^3 + 0.2^2 + 0.2^1 + 0.2^0 = 0$
-1	11111	$\overline{11111} = 00000 + 1 = 00001 = 0.2^4 + 0.2^3 + 0.2^2 + 0.2^1 + 1.2^0 = 1$
...	...	...
-7	11001	inverta e some +1, e aí converta como acima
-8	11000	inverta e some +1, e aí converta como acima
...	...	...
-15	10001	inverta e some +1, e aí converta como acima
-16	10000	inverta e some +1, e aí converta como acima

Vantagem de se representar números binários em complemento de dois

Você deve estar se perguntando qual é a grande vantagem em se usar esta representação. A resposta é que com ela podemos realizar somas e subtrações utilizando-se somente *uma simples operação de soma bit-a-bit*. Veja os exemplos abaixo:

1100 (-4_{10})	1100 (-4_{10})	0100 (4_{10})
+ 0111 ($+7_{10}$)	+ 1101 (-3_{10})	+ 1001 (-7_{10})
----------	----------	----------
0011 ($+3_{10}$)	1001 (-7_{10})	1101 (-3_{10})

No primeiro exemplo (à esquerda) somamos -4 a +7 (o que é subtrair 4 de 7) e a única coisa que fizemos foi uma soma bit-a-bit, levando o "vai um" para a próxima casa. Como não temos mais dígitos à esquerda, quando houver um "vai um" na soma do bit mais significativo ele será desprezado. Todavia o valor calculado (com o número de bits que a representação está usando, no caso 4 bits) ainda estará correto.

No segundo exemplo somamos dois números negativos (-4 e -3) pelo mesmo processo de simples soma bit-a-bit. O resultado será o número "1001" (que é -7, como esperado). No último exemplo subtraímos 7 de 4 (isto é, somamos +4 com -7). O resultado será negativo, como esperado ("1101" que é -3).

Representação em ponto flutuante

Até agora vimos exemplos de conversão de números inteiros e pequenos, mas e quando os valores são tão grandes que precisariam de um enorme número de bits que não dispomos? E como fazemos para codificar números fracionários, racionais e irracionais?

Para resolver esta demanda, lançamos mão de uma estratégia para codificar em base binária números grandes e valores fracionários que explora uma outra forma de representar qualquer número, em qualquer base, que é a ***notação científica***, usando-se uma *mantissa* e um *expoente*. Chamamos também esta estratégia de *representação em ponto flutuante*.

O quadro abaixo mostra exemplos da representação numérica em notação científica para números decimais e binários, mostrando que a estratégia é aplicável a qualquer base.

Número	*base*	*Representação em notação científica*	*mantissa*	*expoente*
1532	10	0.1532×10^{4}	0.1532	4
37,24	10	0.3724×10^{2}	0.3724	2
0,03724	10	0.3724×10^{-1}	0.3724	-1
1010	2	0.1010×2^{4}	0.1010	4
10,11	2	0.1011×2^{2}	0.1011	2
0,01011	2	0.1011×2^{-1}	0.1011	-1

Note que podemos sempre deslocar a vírgula (ponto decimal) do número, de tal maneira a representá-lo no formato 0,XXXX. Ao fazermos isso estamos mudando a ordem de grandeza do número. Este deslocamento equivale na prática a multiplicar o número pelo valor da base tantas vezes quanto forem o número de casas deslocadas. Por exemplo, o número 100 é igual ao número

1,00 x 10 x 10 = 1,00 x 10^2 = 100, isto é, multiplicamos 1,00 por 10 e de novo por 10 e o novo número será o original com o ponto decimal deslocado 2 casas para a direita.

O mesmo se aplica a deslocamentos à esquerda, mas neste caso dividimos o numeral pela base tantas vezes quanto deslocarmos a vírgula para a esquerda. Por exemplo o número 0,01 = 1,00 x 10^{-2}, isto é, deslocar o ponto decimal do número 1,00 duas casas à esquerda equivale a dividi-lo por 10 e de novo dividi-lo por 10, de forma que 0,01 = 1,00 x 10^{-2}. É por causa desta habilidade do ponto decimal em se deslocar que dizemos que ele é *flutuante*. Ele pode se deslocar para qualquer parte, e seu deslocamento é registrável no valor do expoente, que dirá quanto e para onde o ponto se deslocou:

$$1234 = 123{,}4 \times 10^1 = 12{,}34 \times 10^2 = 1{,}234 \times 10^3 = 0{,}1234 \times 10^4 = \ldots$$

Esta forma de representação na verdade tem 4 informações a guardar:

- ✓ o número no formato 0,XXX..., que é chamado de *mantissa*
- ✓ o seu sinal (se positivo ou negativo),
- ✓ a *base*,
- ✓ e o *expoente* que corresponde ao número de casas deslocadas.

Para representar o número decimal 37,24 em notação científica guardaremos a mantissa 0,3724, a base 10, o sinal + e o expoente 2. Para representar o número binário 0,001101= 0,1101 x 2^{-2} guardaremos sua mantissa 0,1101, a base que é 2, o sinal + e o expoente -2.

Para simplificar o processo, supondo que a base seja implicitamente conhecida, na verdade precisaremos guardar somente o *sinal*, a *mantissa* e o *expoente*. Também precisamos definir qual será o tamanho em bits de cada um destes blocos. Por exemplo, para guardar a mantissa (que tem a informação mais importante) com mais resolução, podemos usar por exemplo 8 bits. Já o expoente, podemos guardá-lo com 4 bits. Com estas escolhas, poderemos representar até 2^8 = 256 valores de mantissa, e até 2^4 = 16 valores de expoente. Para o sinal, basta um bit para guardá-lo, na forma já convencionada, em que 0 corresponde a + e 1 corresponde a -.

Não podemos esquecer que o expoente também pode ser negativo, portanto é uma boa idéia usarmos a representação em complemento de dois para guardar o expoente. Com 4 bits, poderemos guardar expoentes que variam de -8 a +7. Estas escolhas inevitavelmente vão determinar a *precisão máxima* com que podemos representar números, algo inerente a qualquer sistema de representação numérica. Se quisermos mais precisão, capacidade de representar mais números e ordens de grandeza mais finas ou maiores, mais bits serão necessários.

Para finalizar nosso exemplo, que objetiva armazenar o número 0,001101 em formato de ponto de flutuante, codificamos os valores da mantissa com 8 bits e do expoente em complemento de dois com 4 bits. Assim,

0,001101= 0,1101 x 2^{-2} --> sinal = 0 | expoente = -2 = 1110 | mantissa = 0,11010000

O nosso número codificado em ponto flutuante será 0 1110 1101 0000

Para recuperar um número codificado em formato de ponto flutuante é preciso portanto saber como os blocos de informações básicas (sinal, mantissa e expoente) estão organizados, isto é, quantos bits cada bloco tem, como estão codificados e qual é a ordem dos blocos de bits. No exemplo acima, o número terá no total 13 bits: 0 primeiro bit é o sinal, os 4 seguintes são o expoente em complemento de dois, e os 8 últimos são a mantissa (a parte fracionária dela).

Várias estratégias de organização são possíveis, a mais difundida no mundo digital é o formato IEEE 754, que especifica três níveis de precisão numérica: 24 bits, 53 bits e 63 bits. Os números

neste formato também são codificados em forma binária na seguinte ordem: sinal | expoente | mantissa.

Você pode saber mais sobre o formato IEEE 754 (*Standard for binary floating-point arithmetic*) buscando por tutoriais ou livros que tratam sobre este assunto em específico ou mesmo baixando o padrão do site do IEEE (*Institute of Electrical and Electronic Engineers*). O padrão especifica os formatos numéricos possíveis para as várias precisões e operações básicas de conversão, e foi proposto originalmente por volta de 1985, sofrendo algumas revisões ao longo das décadas de uso. A revisão mais recente de 2008 está disponível para compra ou *download* direto no link http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4610933 sob o título "*IEEE Standard for Floating-Point Arithmetic*".

Revisão

Neste tutorial você aprendeu como:

- ✓ converter um número decimal (inteiro e positivo) em binário
- ✓ converter um número binário (inteiro e positivo) em decimal
- ✓ representar números binários negativos utilizando-se complemento de dois
- ✓ converter um número decimal (inteiro positivo ou negativo) para binário em complemento de dois
- ✓ converter um número em complemento de dois para decimal (inteiro positivo ou negativo)
- ✓ fazer operações aritméticas de soma e subtração com números em complemento de dois
- ✓ representar um número real e fracionário em formato de ponto flutuante

Histórico de revisões deste tutorial:

Sem.01.2021	correções e atualização. release 2ª edição
Sem.01.2015	adicionada codificação em ponto flutuante
Sem.01.2012	release inicial