Sistemas de Numeração

Alexsandro Santos Soares
prof.asoares@gmail.com

Universidade Federal de Uberlândia
Faculdade de Computação

Introdução

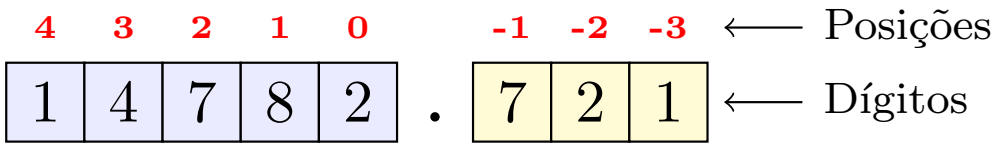
- A computação utiliza vários sistemas de numeração.
- Os computadores usam o binário, base 2, que somente possui dois símbolos para cada posição numérica: 0 ou 1.
- Programadores usam uma notação abreviada para representar números binários, a hexadecimal (base 16).
- Programadores também usam o sistema decimal, base 10.
- De vez em quando, também encontramos aplicações que usam a base 256.
- Como todos estes sistemas são usados em C, precimos de uma compreensão básica de cada um para entender verdadeiramente a linguagem.

Sistema posicional

- Todos os sistemas de numeração do slide anterior são posicionais:
  - A posição de um símbolo em relação aos outros símbolos determina seu valor.
- Nos inteiros e na parte inteira de um número real, a posição inicia com 0 e segue até $n - 1$, com n sendo o número de símbolos na parte inteira.
- Na parte fracionária dos números reais, a posição inicia com -1 e vai até m, com m sendo o número de símbolos na parte fracionária.
- A cada posição é atribuído um peso que varia de acordo com a base usada.

Números decimais (base 10)

- Todos já conhecemos os números decimais.
- O sistema decimal utiliza 10 símbolos para representar valores numéricos: 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9.
- Na figura abaixo, o número decimal 14782.721 possui oito dígitos nas posições de -3 a 4.



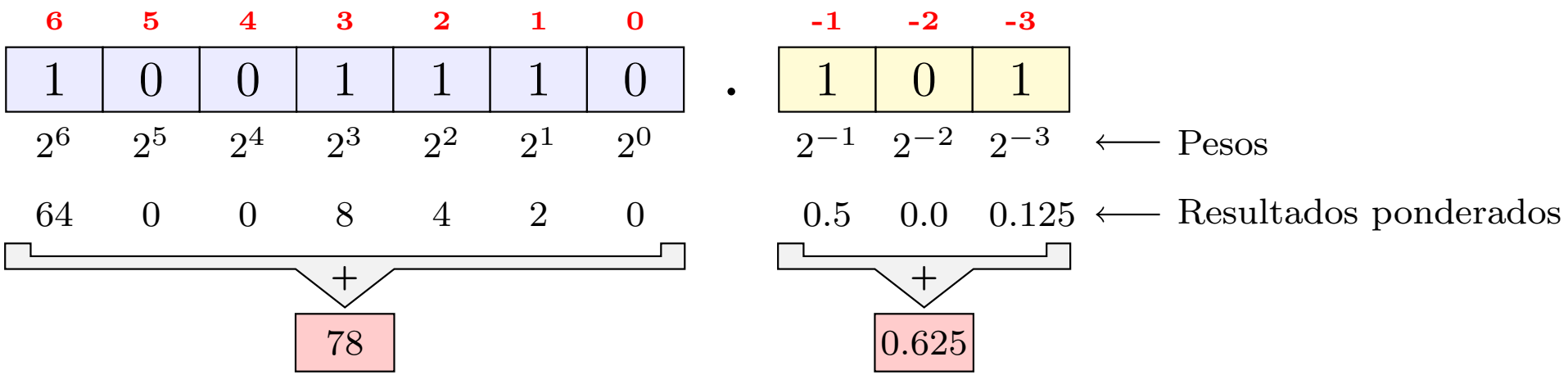
Note que

$$14782.721 = 1\times10^{4}+4\times10^{3}+7\times10^{2}+8\times10^{1}+2\times10^{0}+7\times10^{-1}+2\times10^{-2}+1\times10^{-3}$$

- No sistema decimal, cada peso é 10 elevado à potência de sua posição.

Conversão de binário para decimal

- No sistema numérico binário, base 2, cada peso é igual a 2 elevado à potência de sua posição.
- Para converter um número binário para decimal, multiplicamos cada dígito por seu peso e somamos todos os resultados ponderados.
- Na figura a seguir é ilustrado o processo de converter o binário 1001110.101 para o seu equivalente decimal 78.625.

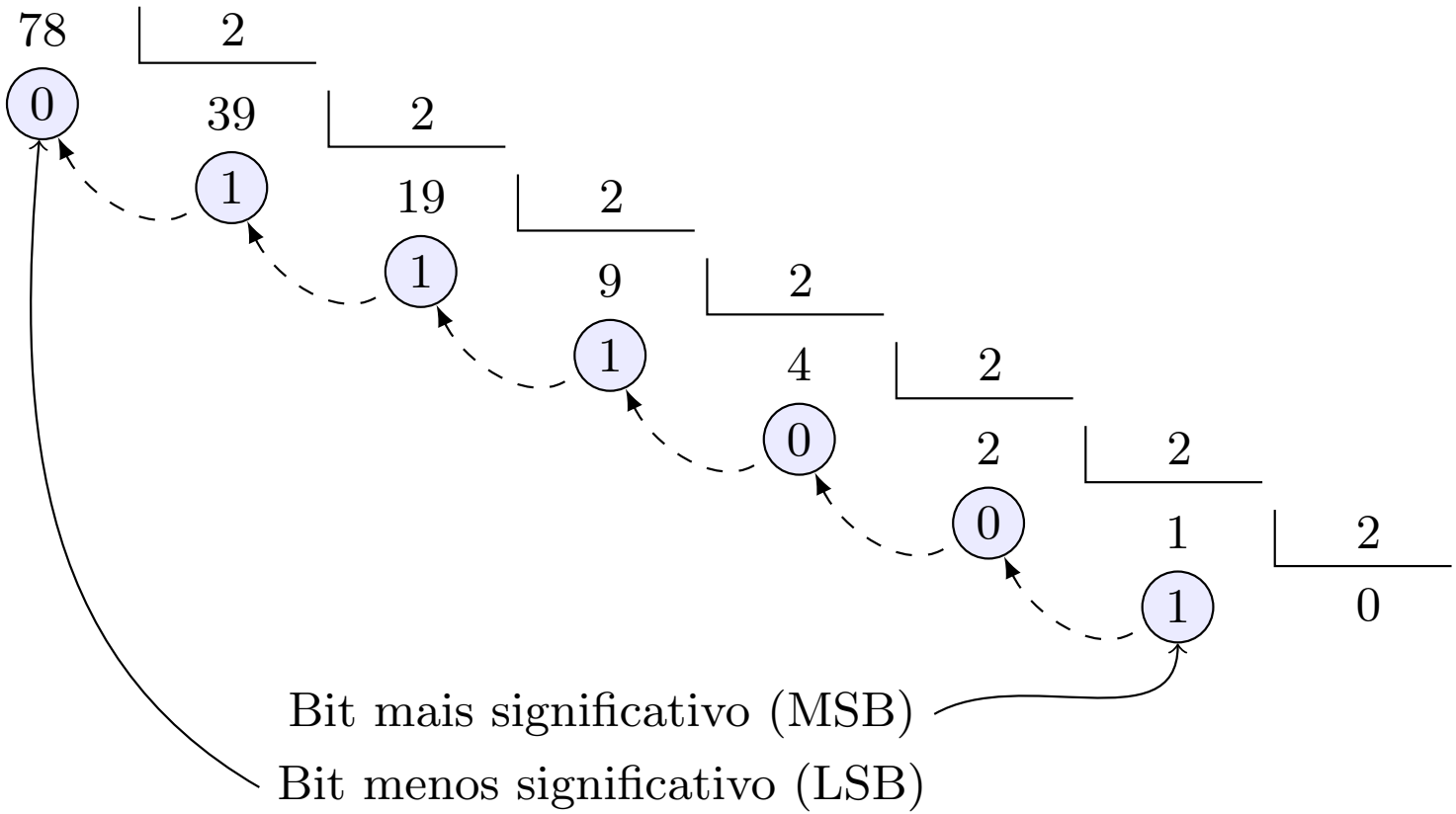


Conversão de decimal para binário – parte inteira

- Para converter de decimal para binário, separamos a parte inteira da fracionária.
- Para converter a parte inteira usamos o método das divisões sucessivas:
  1. Dividimos o número por 2 e escrevemos o resto, que deve ser 0 ou 1. O primeiro resto torna-se o *dígito binário menos significativo*.
  2. Tomamos o quociente da divisão e novamente o dividimos por 2 e escrevemos o novo resto na segunda posição.
  3. Repetimos esse processo até que o quociente torne-se zero.

Exemplo de conversão da parte inteira para binário

Imagine que se deseje saber a representação binária do número decimal 78. Usando as divisões sucessivas, teremos



Logo,

$$(78)_{10} = (1001110)_2$$

Conversão de decimal para binário – parte fracionária

- Para converter a parte fracionária usamos as multiplicações sucessivas:
  1. Multiplicamos o número por 2 e escrevemos a parte inteira desse produto, que deve ser 0 ou 1. Essa parte inteira torna-se um dígito binário.
  2. Tomamos a parte fracionária da multiplicação e novamente a multiplicamos por 2, pegamos a parte inteira e a usamos como o próximo dígito binário.
  3. Repetimos esse processo até que o produto torne-se zero.
- Precisamos limitar o processo de multiplicações sucessivas pois o produto pode nunca se tornar zero. Assim, decidimos quantos dígitos precisamos no lado direito do número binário e paramos quando tivermos alcançado essa quantidade.

Exemplo de conversão da parte fracionária para binário

Imagine que se deseje saber a representação binária do número decimal 0.625. Usando as multiplicações sucessivas, teremos

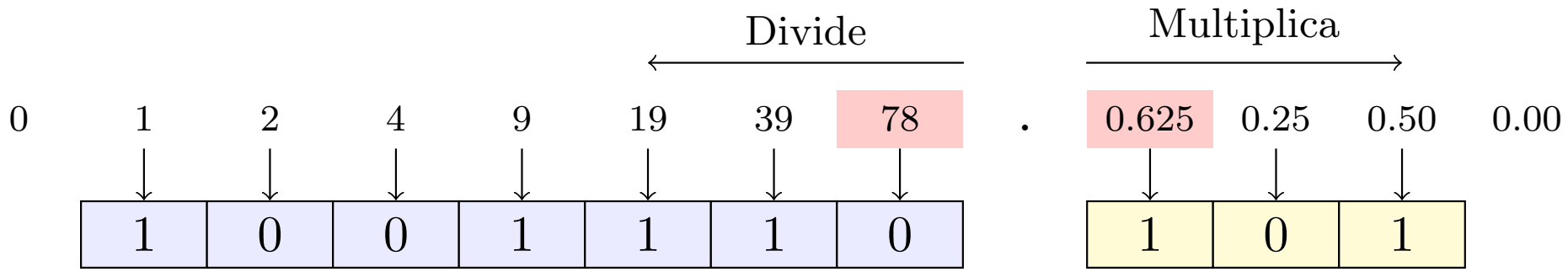
$$\begin{array}{lcl|l} 0.625 \times 2 & = & 1.25 & 1 \\ 0.25 \times 2 & = & 0.50 & 0 \\ 0.50 \times 2 & = & 1.0 & 1 \\ 0.00 \times 2 & = & 0 & \end{array} \quad \downarrow$$

Desta forma,

$$(0.625)_{10} = (0.101)_2$$

Exemplo de conversão de decimal para binário

Na figura abaixo é esquematizado o processo para a obtenção da representação binária do número decimal 78.625.

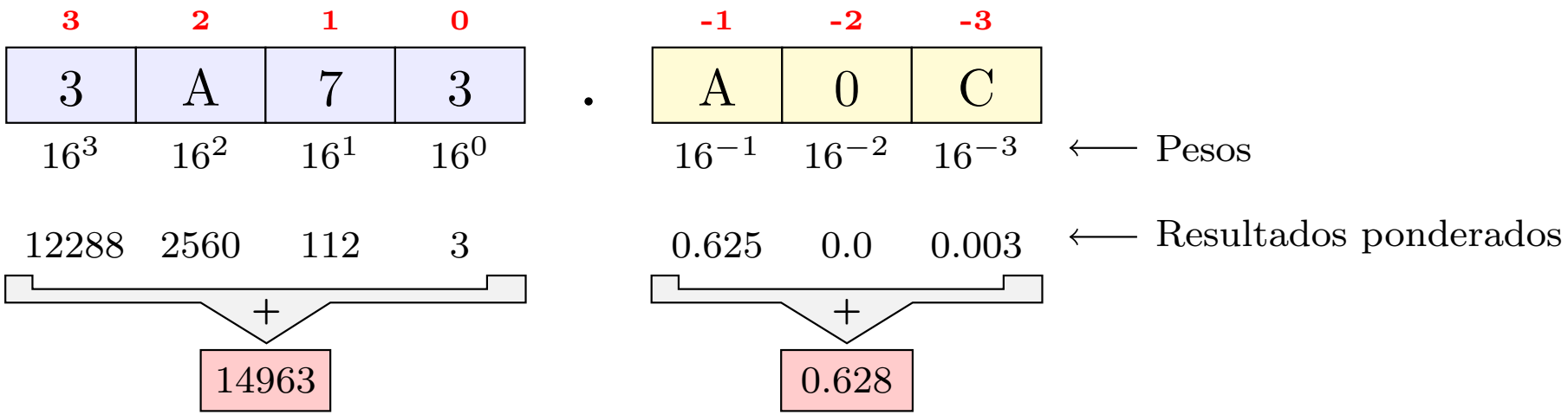


Portanto,

$$(78.625)_{10} = (1001110.101)_2$$

Conversão de hexadecimal para decimal

- No sistema numérico hexadecimal, base 16, usamos 16 símbolos $0, 1, \ldots, 9, A, B, C, D, E, F$ para os dígitos.
  - Os mesmos símbolos que a representação decimal mas, ao invés de 10, 11, 12, 13, 14 e 15, usamos A, B, C, D, E, F, respectivamente.
- Cada peso é igual a 16 elevado à potência de sua posição.
- Para converter um número hexadecimal para decimal, multiplicamos cada dígito por seu peso e somamos todos os resultados ponderados.
- Na figura a seguir é ilustrado o processo de converter o hexadecimal $3A73.A0C$ para o seu equivalente decimal 14 963.628.

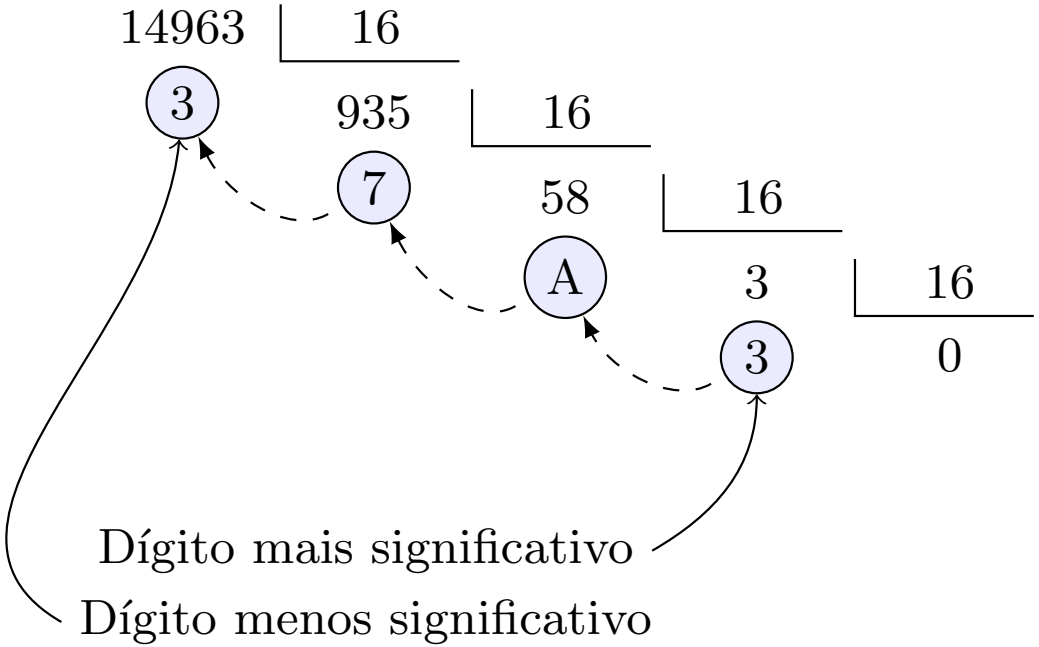


Conversão de decimal para hexadecimal

- Para converter um número decimal em sua representação hexadecimal, procedemos de forma análoga ao que fizemos com a representação binária:
  - Separamos a parte inteira da fracionária.
  - Convertemos a parte inteira usando divisões sucessivas por 16.
  - Convertermos a parte fracionária usando multiplicações sucessivas por 16.

Exemplo de conversão da parte inteira para hexadecimal

Imagine que se deseje saber a representação hexadecimal do número decimal 14 963. Usando as divisões sucessivas, teremos



Logo,

$$(14963)_{10} = (3A73)_{16}$$

Exemplo de conversão da parte fracionária para hexadecimal

Imagine que se deseje saber a representação hexadecimal do número decimal 0.628. Usando as multiplicações sucessivas, teremos

$$\begin{array}{lcl|l} 0.628 \times 16 & = & 10.048 & \mathrm{A} \\ 0.048 \times 16 & = & 0.768 & 0 \\ 0.768 \times 16 & = & 12.288 & \mathrm{C} \\ 0.288 \times 16 & = & 4.608 & 4 \\ \ldots & = & \ldots & \ldots \end{array} \quad \downarrow$$

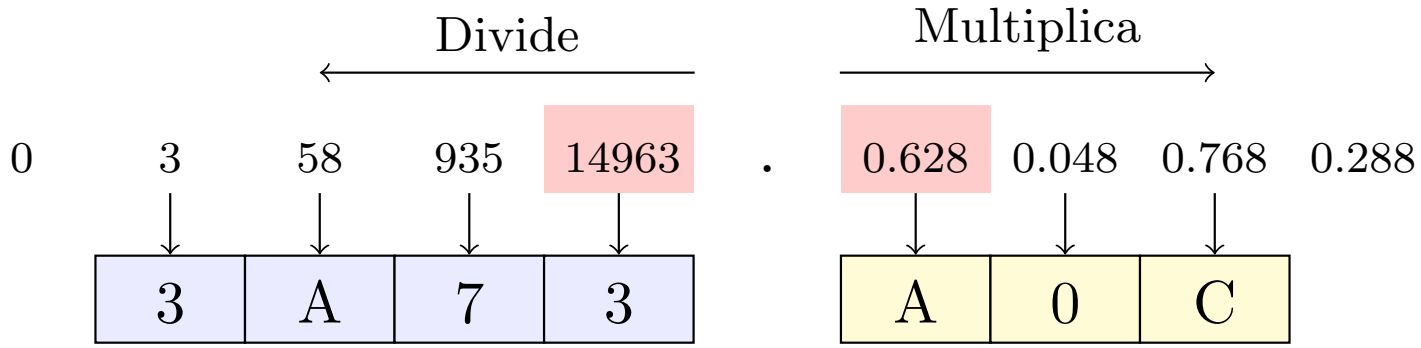
Desta forma,

$$(0.628)_{10} \approx (0.AOC)_{16}$$

Note que limitamos o número de dígitos hexadecimais em 3 na parte fracionária.

Exemplo de conversão de decimal para hexadecimal

Na figura abaixo é esquematizado o processo para a obtenção da representação hexadecimal do número decimal 14 963.628.



Portanto,

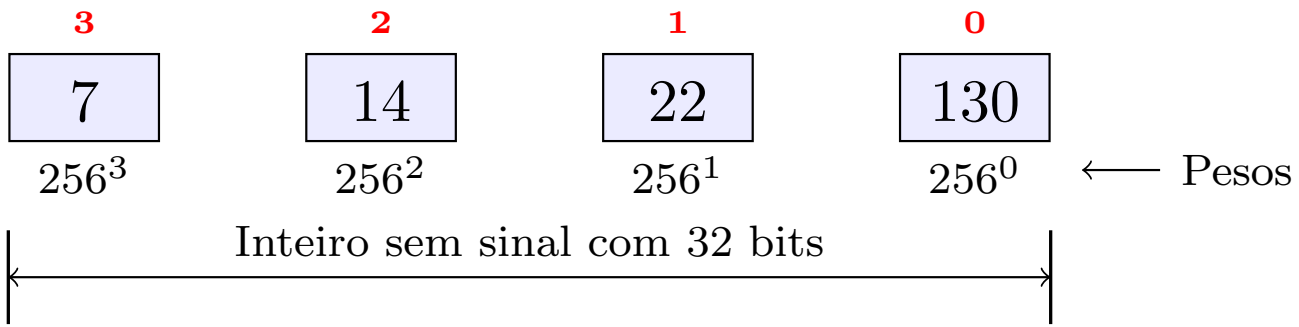
$$(14963.628)_{10} \approx (3A73.A0C)_{16}$$

Base 256

- A base 256 é normalmente usada em duas situações:
  - Criar um número a partir de bytes individuais.
  - Endereços IP da Internet.
- A conversão de um número na base 256 para decimal é similar à conversão de números binários e hexadecimais mas, agora, os pesos são potências de 256.
- Também, a conversão de números decimais para a base 256, segue a lógica das divisões sucessivas por 256, usando o resto como dígito.

Conversão de byte para decimal

Abaixo temos quatro bytes individuais, cada um contendo um número inteiro sem sinal entre 0 e 255. Às vezes, precisamos considerar vários destes bytes como um número.



O valor do número inteiro contido nestes quatro bytes é então

$$7 \times 256^3 + 14 \times 256^2 + 22 \times 256^1 + 130 \times 256^0 = 118\ 363\ 778$$

Comparação de três sistemas numéricos

- Na tabela abaixo é mostrado como os três sistemas representam os números decimais de 0 a 15.

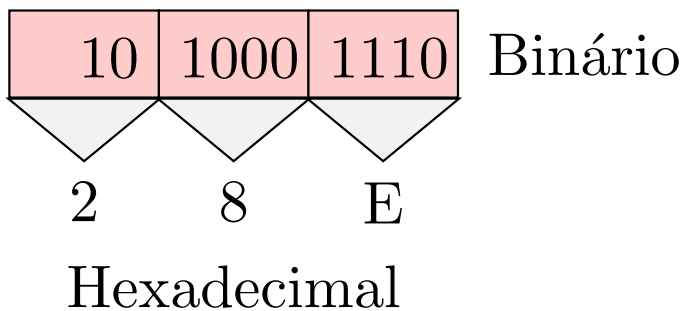
Decimal	Binário	Hexadecimal
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Outras conversões

- Ao convertermos números da base 2 para a 16 ou da base 16 para a 256 podemos usar a base 10 como intermediária.
  - Para mudar um número de binário para hexadecimal, primeiro convertemos o número de binário para decimal e depois de decimal para hexadecimal.
- Entretanto, existem maneiras mais fáceis de realizar estas conversões sempre que as bases em questão forem potências de 2.

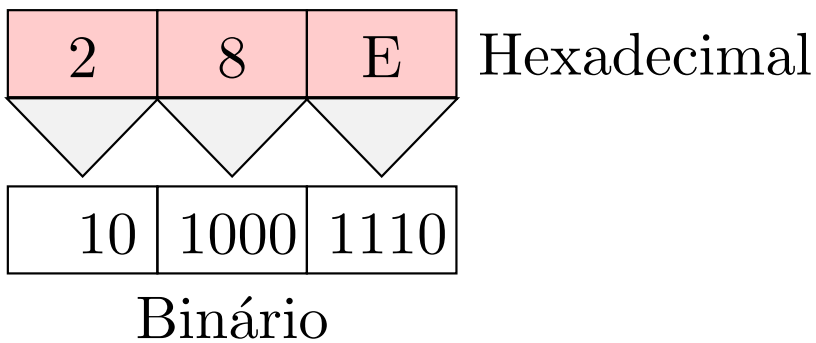
Conversão de binário para hexadecimal

Para converter um número de binário para hexadecimal, agrupamos os dígitos binários da direita para a esquerda em grupos de quatro.
Depois convertemos cada grupo de quatro bits em seu equivalente hexadecimal usando a tabela vista no slide anterior.
No exemplo a seguir é ilustrada a técnica para converter o binário 1010001110 para hexadecimal.



Conversão de hexadecimal para binário

Para converter de hexadecimal para binário, convertemos cada dígito hexadecimal em seu equivalente binário usando a tabela vista anteriormente e concatenamos os resultados.
No exemplo a seguir é ilustrada a técnica para converter o hexadecimal $28E$ para binário.

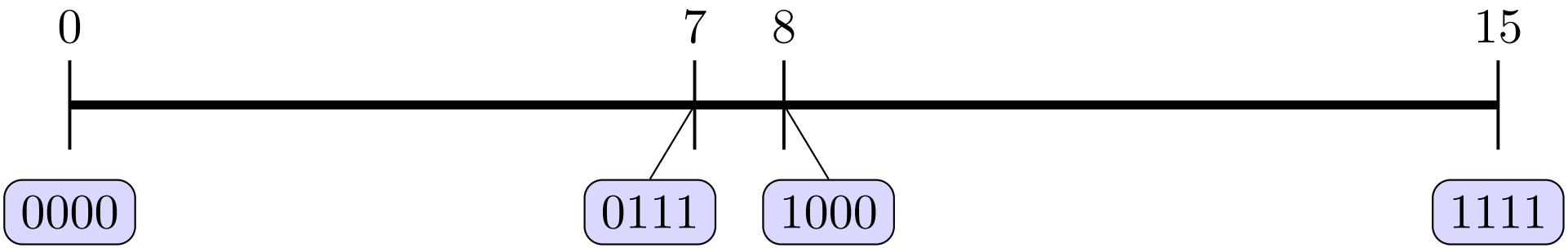


Armazenando inteiros

- Discutimos como inteiros e números reais são representados em diferentes bases.
- Embora as bases 16 e 256 sejam usadas na ciência da computação, os dados são armazenados no computador em binário.
  - Números devem ser convertidos para a base 2 antes de serem armazenados.
- Ignoramos, até agora, o sinal do número.
- Vamos nos concentrar agora em como números inteiros são armazenados no computador e depois veremos a situação para os reais.

Inteiros sem sinal

- Armazenar inteiros sem sinal é direto:
  - O número é convertido em sua representação binária e depois armazenado.
- Na figura abaixo vemos como inteiros sem sinal de 0 a 15 podem ser armazenados como inteiros de 4 bits.



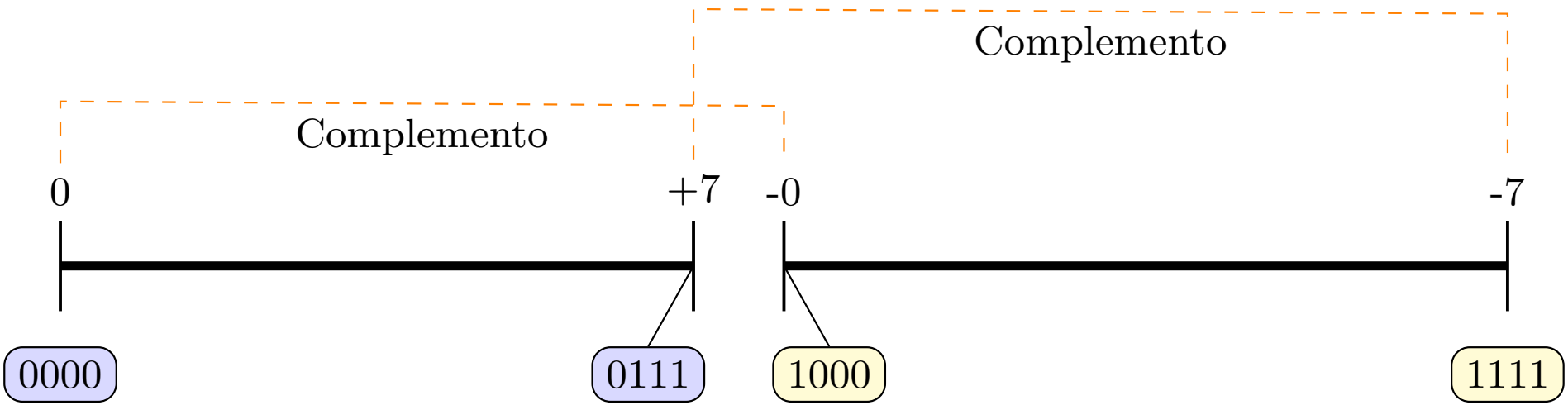
- A adição de inteiros sem sinal é simples desde que não haja *overflow*.
- Já a subtração deve ser realizada com cautela pois, se o resultado for negativo o número não será um inteiro sem sinal.
  - Em geral, os computadores promovem o resultado para um inteiro com sinal.

Inteiros com sinal

- Armazenar inteiros com sinal é diferente de armazenar inteiros sem sinal pois devemos converter tantos os números positivos quanto os negativos.
- Discutiremos quadro métodos para armazenar inteiros com sinal em um computador:
  - Sinal e magnitude.
  - Complemento para um.
  - Complementos para dois.
  - Sistema excesso-N.

Sinal e magnitude

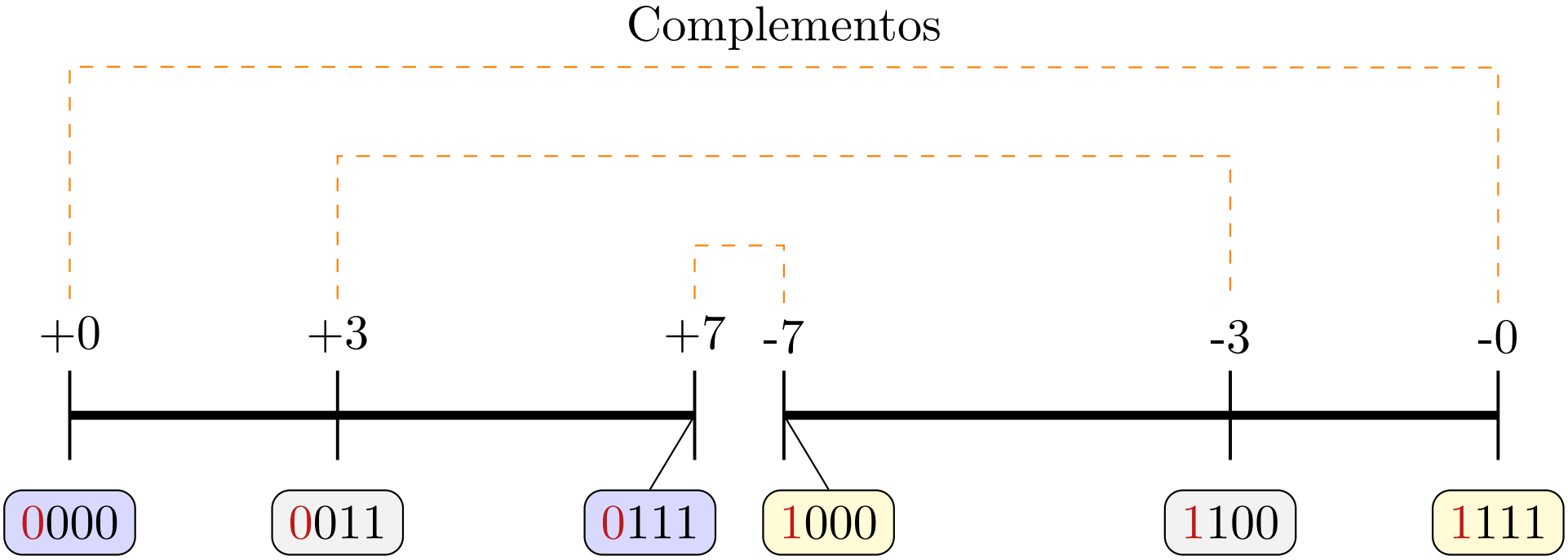
- No sinal e magnitude dividimos o intervalo disponível entre os números positivos e negativos.
  - A parte inferior do intervalo fica com os números positivos.
  - A parte superior, fica com os negativos.
- Para conseguirmos isso, usamos o bit mais à esquerda para representar o sinal e os bits restantes para representar o valor absoluto do número (a magnitude).
- No exemplo abaixo vemos como um inteiro de 4 bits pode manter os números positivos e negativos do intervalo $[-7, +7]$.



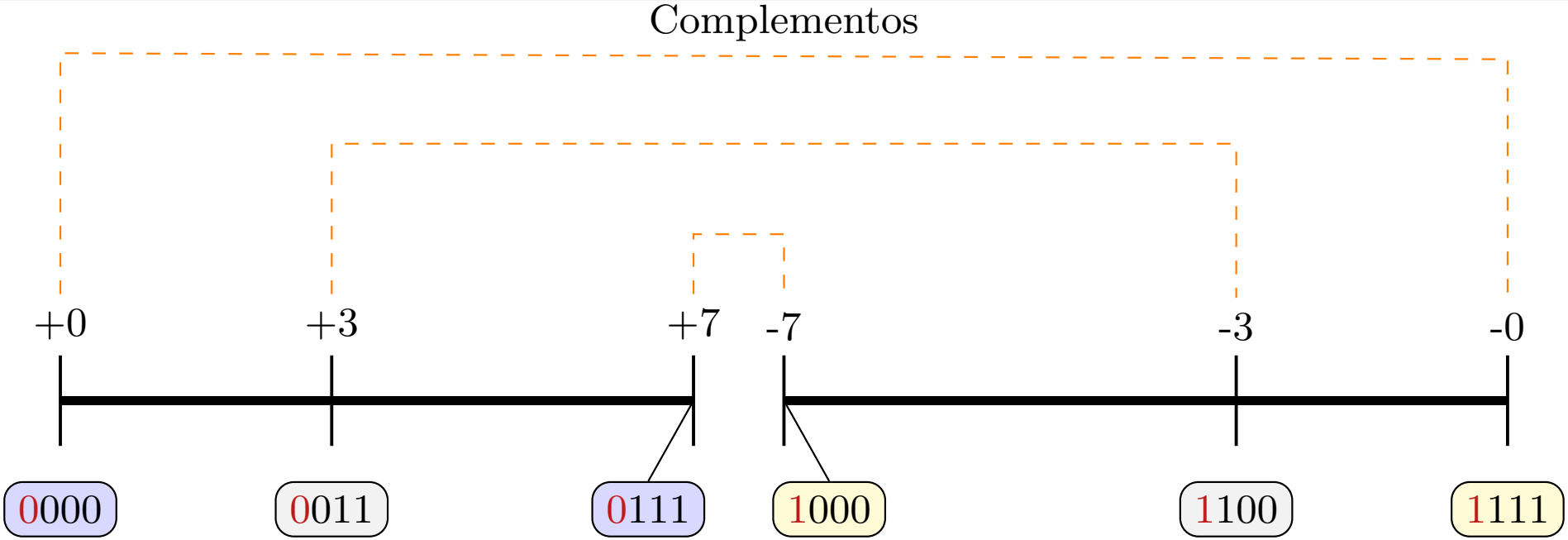
- Note que o bit mais à esquerda contém o sinal: 0 é positivo e 1 é negativo.
- Existem dois zeros: +0 (0000) e -0 (1000).

Complemento para um

- O complemento para um é similar ao sinal e magnitude mas, a partição do intervalo entre números positivos e negativos é diferente.
  - Os números são arranjados simetricamente.
- Um número positivo e um negativo são simétricos em relação ao meio do intervalo.



Propriedades do complemento para um



As propriedades do complemento para um são:

1. O bit mais à esquerda contém o sinal.
2. Existem dois zeros.
3. Para inverter o sinal de um número, inverte-se cada bit individualmente.
4. Se somarmos $A + (-A)$ obtemos -0 (todos os bits são 1).
5. Para somar $A + B$, somamos os números bit a bit. Para subtrair $A - B$, somamos A ao complemento de B, ou seja, $A - B = A + (-B)$. A única coisa a considerar é adicionar o vai-um produzido na última coluna do resultado.

Exemplos de somas e subtrações com complemento para um

Somas

$$\begin{array}{r} (+3) \\ +(+2) \\ \hline (+5) \end{array} \qquad \begin{array}{cccc} & 1 & & \\ 0 & 0 & 1 & 1 \\ + \; 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 1 \end{array}$$

$$\begin{array}{r} (+3) \\ +(-2) \\ \hline (+1) \end{array} \qquad \begin{array}{cccc} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ + \; 1 & 1 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 \\ & & & 1 \\ \hline 0 & 0 & 0 & 1 \end{array}$$

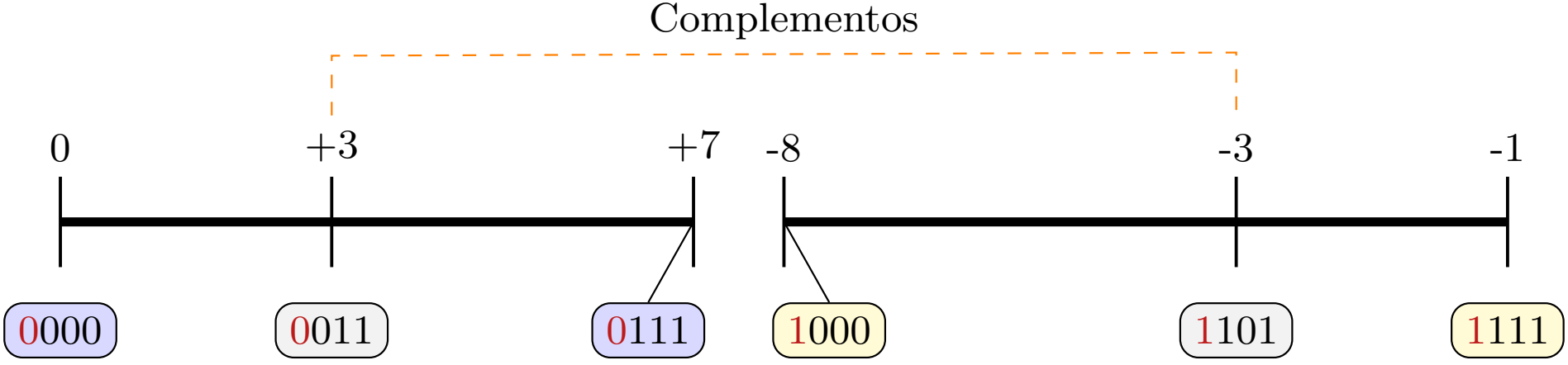
Subtrações (adicione o primeiro com o complemento do segundo)

$$\begin{array}{r} (+3) \\ -(+4) \\ \hline (-1) \end{array} \qquad \begin{array}{cccc} & 1 & 1 & \\ 0 & 0 & 1 & 1 \\ + \; 1 & 0 & 1 & 1 \\ \hline 1 & 1 & 1 & 0 \end{array}$$

$$\begin{array}{r} (-3) \\ -(-2) \\ \hline (-1) \end{array} \qquad \begin{array}{cccc} & 1 & & \\ 1 & 1 & 0 & 0 \\ + \; 0 & 0 & 1 & 0 \\ \hline 1 & 1 & 1 & 0 \end{array}$$

Complemento para dois

- O complemento para dois de um número com N bits é definido como o complemento em relação a 2^N.
  - Por exemplo, para o número de 4 bits 0010 (2_{10}), o complemento para dois é 0110, pois $0010 + 0110 = 10000$.
  - Note que 2^N em binário é representado por 1 seguido de N zeros.
- O bit mais significativo (MSB) é o sinal: 0 para positivo e 1 para negativo.
- Os números positivos são representados na sua forma binária direta.
- Os números negativos são representados na forma de complemento para 2.
  - Por exemplo, para encontrar a representação em 4 bits de -5, calculamos $2^4 - 5_{10} = (16 - 5)_{10} = 11_{10} = 1011_2$. Aqui operamos na base 10 e apenas convertemos para binário no final.

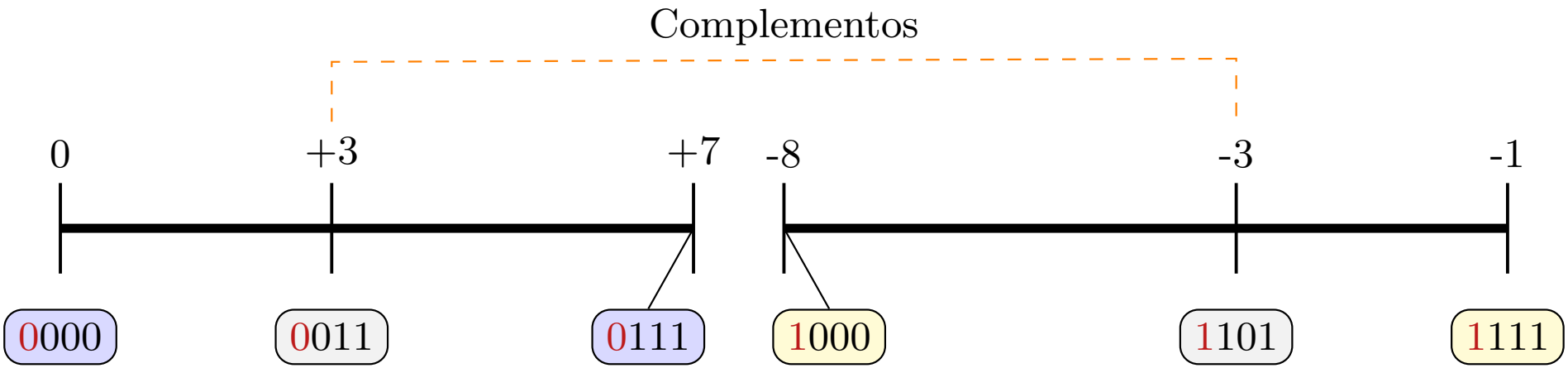


Propriedades do complemento para dois

- O bit mais à esquerda contém o sinal: 0 para positivo e 1 para negativo.
- Existe somente um zero neste método: 0 (0000).
- Para mudar o sinal de um número precisamos de duas operaçãoes. Primeiro precisamos inverter cada bit individualmente e depois adicionar 1.

```
   -3   ⟶   ~ (0 0 1 1) + 1   ⟶   1 1 0 1
-(-3)   ⟶   ~ (1 1 0 0) + 1   ⟶   0 0 1 1
```

- Se somarmos $A + (-A)$ obteremos 0.
- Para somar $A + B$ basta somar os números bit a bit. Para subtrair $(A - B)$, somamos A com o complemento de B. Se houver um vai um, basta descartá-lo.



Exemplos de somas e subtrações com complemento para dois

Somas

$$\begin{array}{r} (+3) \\ +(+2) \\ \hline (+5) \end{array} \qquad \begin{array}{cccc} & 1 & & \\ 0 & 0 & 1 & 1 \\ + \; 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 1 \end{array}$$

$$\begin{array}{r} (+3) \\ +(-2) \\ \hline (+1) \end{array} \qquad \begin{array}{cccc} 1 & 1 & & \\ 0 & 0 & 1 & 1 \\ + \; 1 & 1 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}$$

Subtrações (adicione o primeiro com o complemento do segundo)

$$\begin{array}{r} (+3) \\ -(+4) \\ \hline (-1) \end{array} \qquad \begin{array}{cccc} 0 & 0 & 1 & 1 \\ + \; 1 & 1 & 0 & 0 \\ \hline 1 & 1 & 1 & 1 \end{array}$$

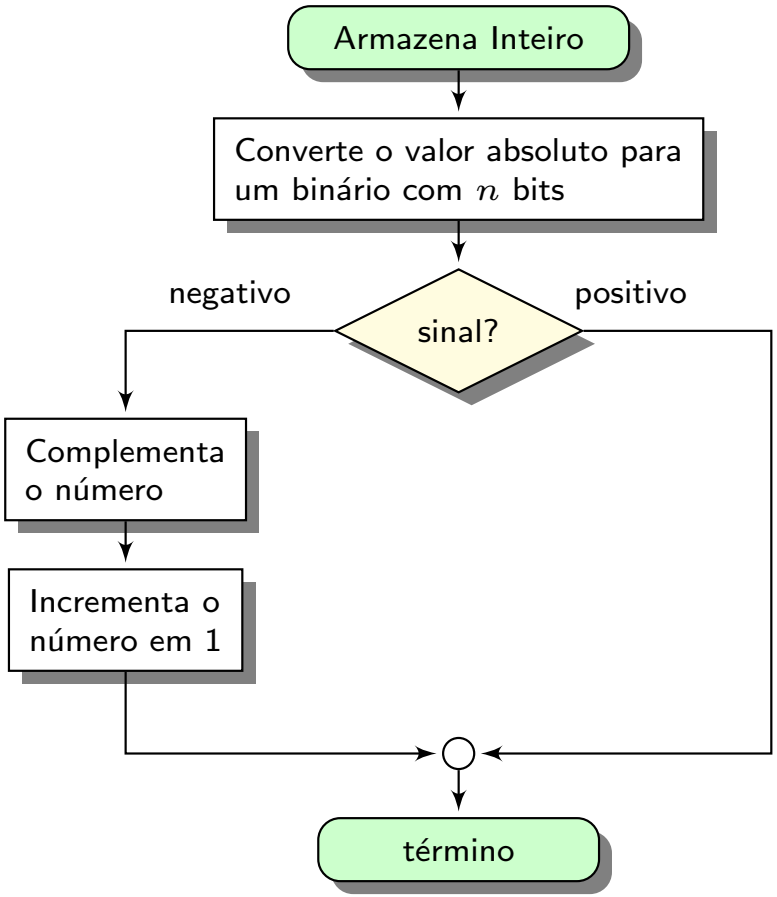
$$\begin{array}{r} (-3) \\ -(-2) \\ \hline (-1) \end{array} \qquad \begin{array}{cccc} 1 & 1 & 0 & 1 \\ + \; 0 & 0 & 1 & 0 \\ \hline 1 & 1 & 1 & 1 \end{array}$$

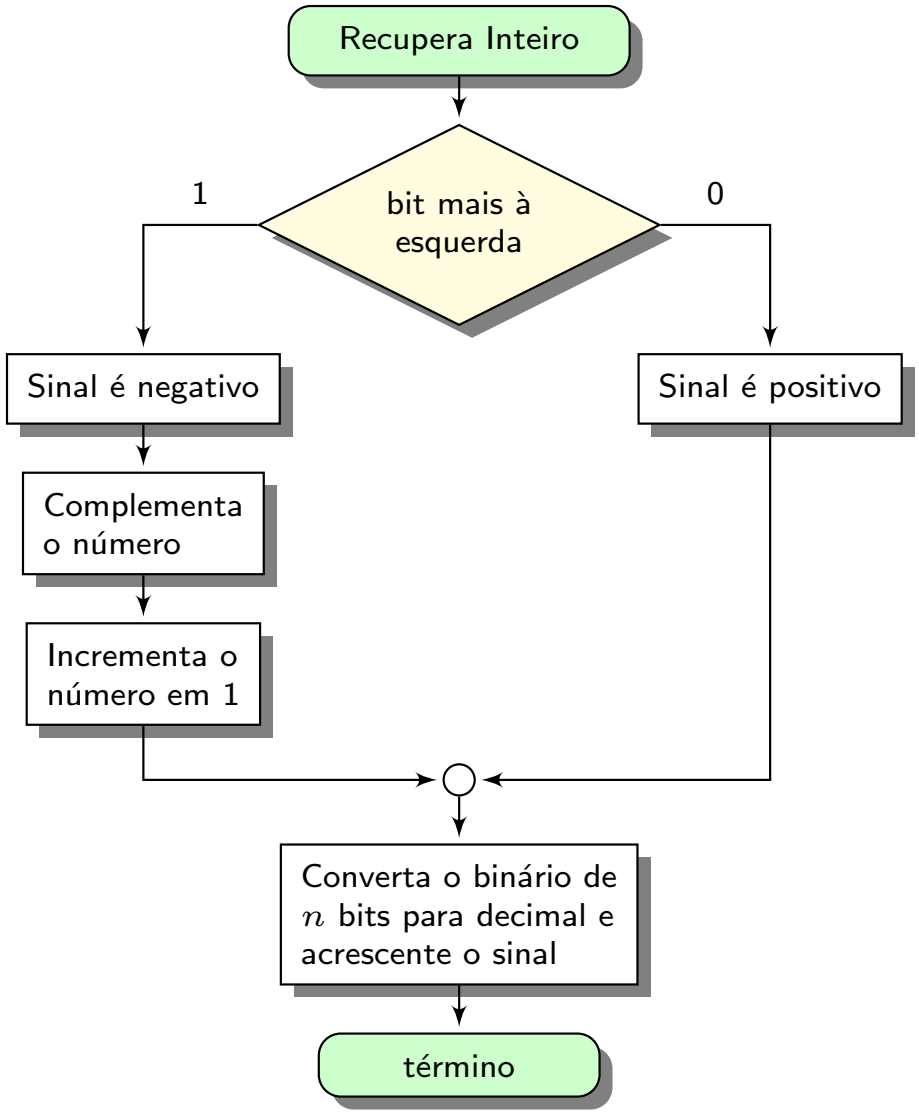
A maioria dos computadores modernos utiliza complemento para dois para armazenar inteiros com sinal.

Armazenando e recuperando em complemento para dois

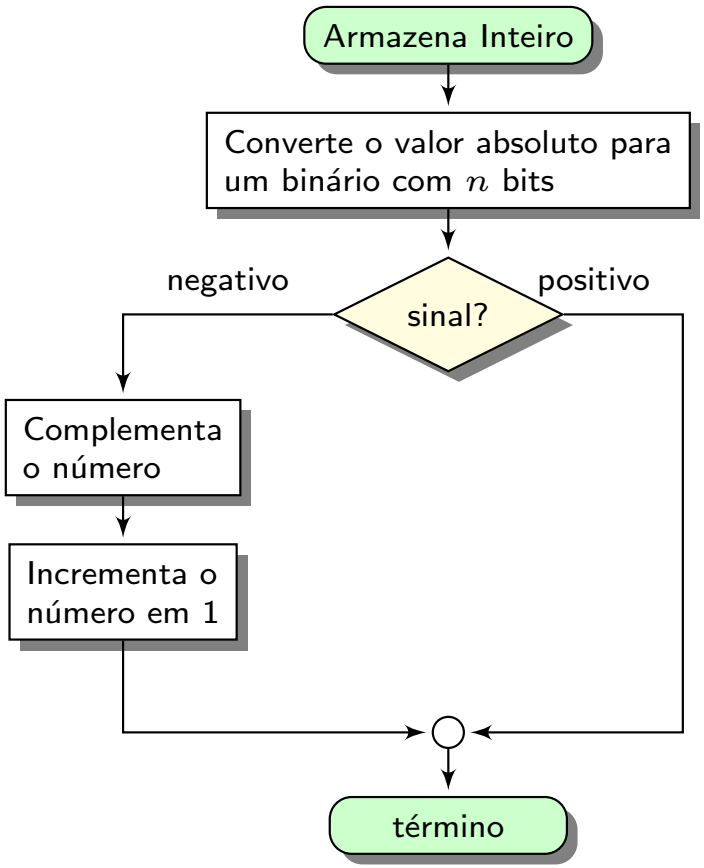
- Vamos entender como uma função de leitura, tal como `scanf`, converte um número decimal inteiro com sinal em um complemento para dois.
- Veremos também o inverso: como uma função de saída, tal como `printf`, converte um número em complemento para dois em um decimal inteiro com sinal.
- No próximo slide há dois algoritmos de alto nível, expressos como fluxogramas, que poderiam ser usados por estas funções.

Armazenando e recuperando em complemento para dois





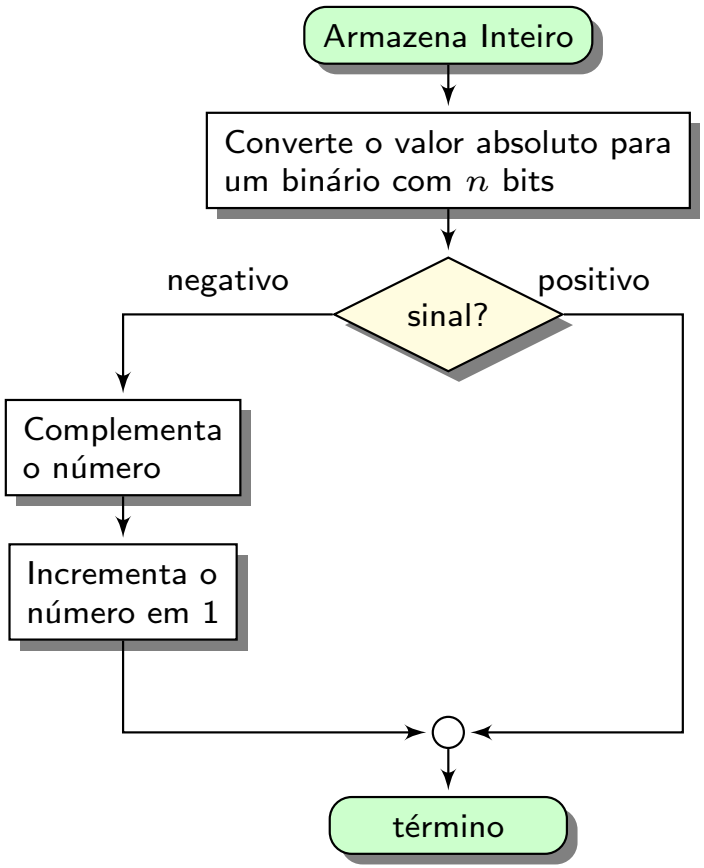
Exemplo 1 de armazenamento em complemento para dois



Seguiremos o algoritmo para ver como $+76$ é armazenado como um inteiro de 16 bits.
O valor absoluto do número (76) é convertido para um número binário de 16 dígitos.
O sinal é positivo, assim o número é armazenado na memória como está.

Número decimal:	$+76$
Converte o valor absoluto:	0000000001001100
Valor armazenado:	0000000001001100

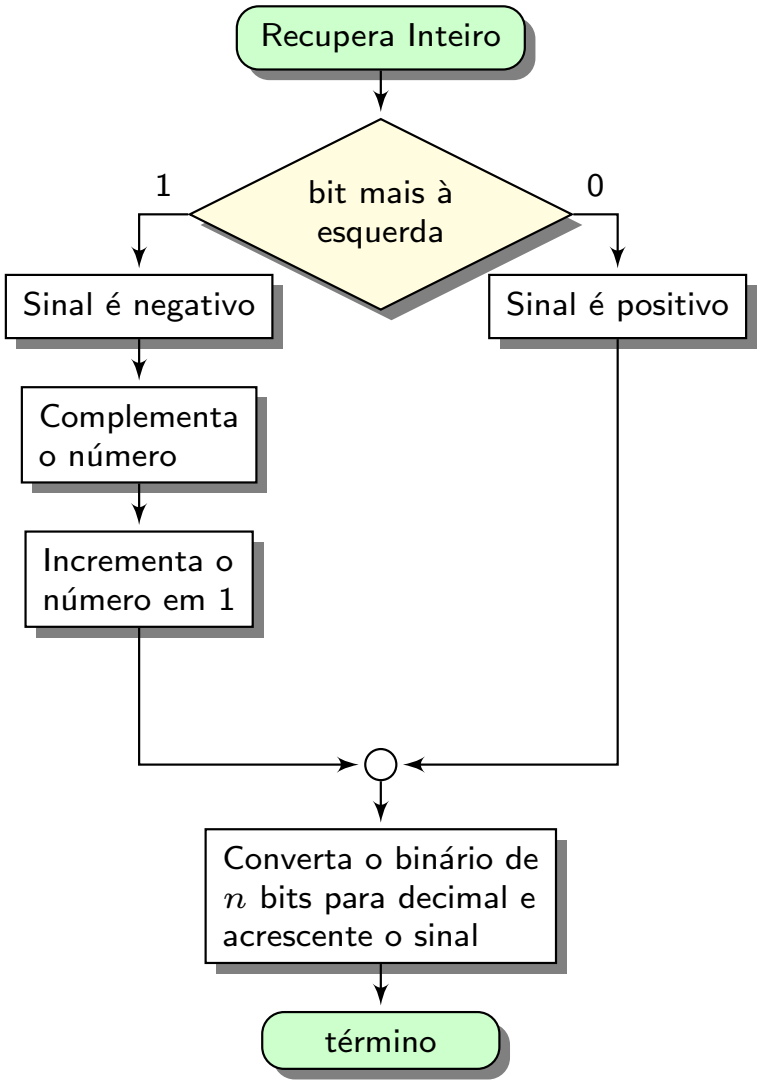
Exemplo 2 de armazenamento em complemento para dois



Seguiremos o algoritmo para ver como -76 é armazenado como um inteiro de 16 bits.
O valor absoluto do número (76) é convertido para um número binário de 16 dígitos.
O sinal é negativo, assim precisamos do complemento do número e depois adicionar 1 ao resultado antes de armazená-lo na memória.

Número decimal:	-76
Converte o valor absoluto:	0000000001001100
Complemento:	1111111110110011
Soma 1:	1111111110110100
Valor armazenado:	1111111110110100

Exemplo de recuperação em complemento para dois



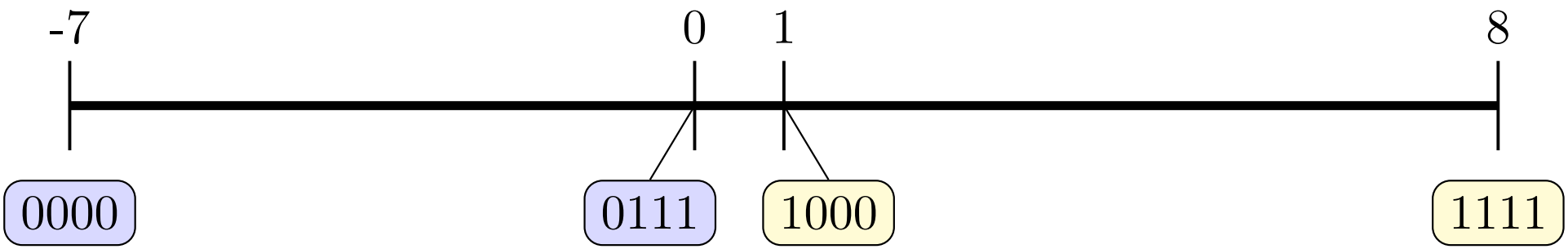
Seguiremos o algoritmo para ver como um valor armazenado como um inteiro de 16 bits é recuperado. O processo é mostrado a seguir

Valor recuperado:	1111111111101011 (sinal -)
Complemento:	0000000000010100
Soma 1:	0000000000010101
Converte para decimal:	21
Acrescenta o sinal:	-21

O valor recuperado é negativo, assim o sinal é guardado para ser acrescentado no final.
O número é então complementado e 1 é adicionado ao resultado.
O resultado é convertido para decimal e o sinal é acrescentado.

Excesso-N

- Existem aplicações que necessitam mais comparações numéricas que operações aritméticas.
- Para esses casos o IEEE criou uma estratégia chamada Excesso-N para armazenar inteiros com sinal.
- A ideia é adicionar um valor fixo N, chamado de valor de polarização, ao número de tal forma que qualquer número negativo no sistema torne-se não-negativo ao ser armazenado na memória.
- A figura a seguir mostra inteiros de 4 bits representados em Excesso-7. Note que o intervalo vai de -7 a 8 e que tantos os números negativos quanto os positivos são armazenados como inteiros sem sinal.

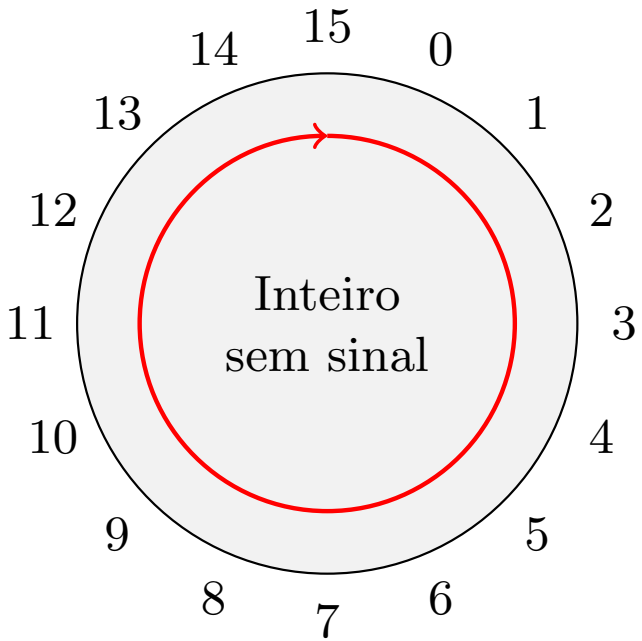


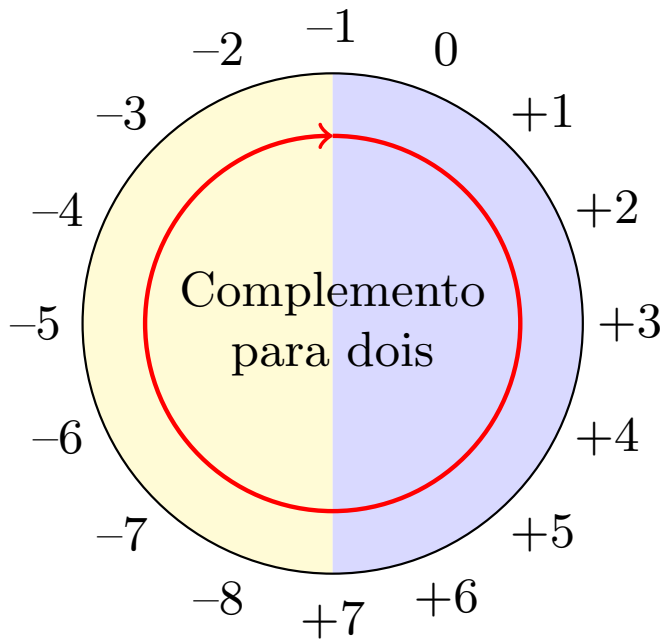
Overflow

- Um problema ocorre quando dois números somados produzem um resultado que necessita de mais bits para ser representado do que está disponível no computador.
- Chamamos esse problema de *overflow*, transbordo, e pode acontecer em somas ou subtrações.
- Por exemplo, se o tamanho de um inteiro em determinada máquina for de 4 bits podemos armazenar um inteiro sem sinal de 0 a 15 ou um inteiro com sinal entre -8 e 7, usando complemento para dois. Qualquer número fora deste intervalo transborda os valores possíveis.
- Em muitos sistemas não há uma mensagem de erro ou algum aviso.
  - Ele apenas descarta o bit extra ou os bits que não couberem no espaço alocado.
  - Isso cria um resultado inválido, positivo ou negativo, que ao ser impresso gera resultados inesperados.

Intervalo de valores inteiros

Podemos compreender melhor o *overflow* se mostrarmos o intervalo dos inteiros que podem ser armazenados na forma de um círculo.





No círculo para o inteiro sem sinal é mostrado que $15 + 1$ é 0. Já no círculo do complemento para dois, $7 + 1$ é -8.
Se incrementarmos um inteiro que mantenha o maior valor possível, obteremos o menor valor possível.

Armazenando números reais

- Em notação científica, escrevemos o número real -314.625 como $-3.14625 \times 10^{+2}$.
- Esta representação é composta por três partes: o sinal $(-)$, a precisão (3.14625) e a potência de 10 $(+2)$.
- Os computadores usam os conceitos da notação científica para armazenar um número real.
- Se escrevermos toda a informação em um sistema binário, -314.625 será representado como mostrado a seguir, com a potência também em binário (4)

$$-10011.10101010 \times 2^{100}$$

- Note que se fixarmos a base para a potência, 10 ou 2, não precisamos usar nem o operador de multiplicação nem a base na representação.

Normalização

- Precisamos resolver um problema antes de armazenar um número real no computador:
  - A posição do ponto binário, ou seja, o ponto que separa a parte inteira da fracionária.
  - Na memória, podemos armazenar apenas dígitos binários, 0 ou 1, não um ponto.
- A solução é chamada de normalização.
  - Normalizaremos o número de tal forma que o ponto sempre esteja em uma posição fixa e conhecida.
- Para normalizar um número, deslocamos o ponto para a esquerda ou para a direita, dependendo de sua posição original, até que haja apenas um dígito diferente de 0 antes do ponto decimal.
- Exemplos de normalização decimal:
  - deslocamentos para a esquerda:
    $314.625 \rightarrow 31.4625 \times 10^1 \rightarrow 3.14625 \times 10^2$
  - deslocamentos para a direita:
    $0.00712 \rightarrow 0.0712 \times 10^{-1} \rightarrow 0.712 \times 10^{-2} \rightarrow 7.12 \times 10^{-3}$
- Se o número for o zero, o dígito antes do ponto será 0.

Normalização de números binários

- O número deve ter apenas um dígito binário à esquerda do ponto.
  - Para valores não nulos, o dígito é 1.
- Deslocar números binários requer a multiplicação ou a divisão por 2 em cada deslocamento.
  - Em outras palavras, se movermos o número para a *esquerda*, precisamos adicionar o número de dígitos deslocados ao expoente. Se, ao contrário, deslocarmos para a *direita*, precisamos subtrair do expoente o número de dígitos deslocados.
- Vamos aplicar esta ideia ao número $-10011.10101010 \times 2^{100}$:
  - Precisamos mover o ponto 4 casas para a esquerda. Assim, devemos acrescentar 4 ao expoente $100_2 = 4_{10}$, cujo resultado será $8_{10} = 1000_2$.
  - O número normalizado será então

$$-1.001110101010 \times 2^{1000}$$

- Após a normalização, teremos somente que armazenar o sinal, a precisão e o expoente.
  - Não temos que armazenar a parte inteira da precisão, que sempre será 1, e nem o ponto binário.

Sinal, expoente e mantissa

- Os números reais contém três partes: o sinal (s), o expoente (e) e a mantissa (m).
- Podemos dizer que o número original (N) pode ser expresso por

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

- O sinal do número é armazenado usando um bit: 0 para mais e 1 para menos.
- O expoente de uma potência de 2 pode ser negativo ou positivo. Vamos usar o método do Excesso-N para armazenar o expoente.
- A mantissa é o número binário à direita do ponto binário. Ela define a precisão do número e é armazenada como um inteiro sem sinal.

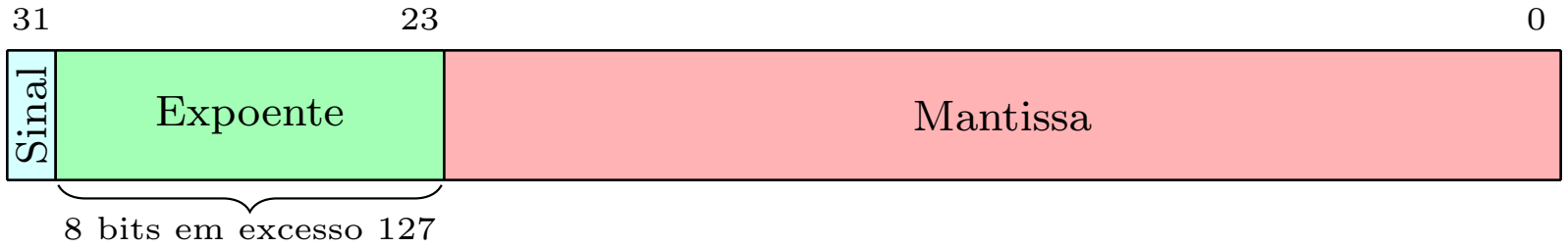
Padrões IEEE

- O Institute of Electrical and Electronics Engineers (IEEE) definiu o padrão IEEE 754-2008 para armazenar números na memória.
- Alguns deles são mostrados na tabela a seguir:

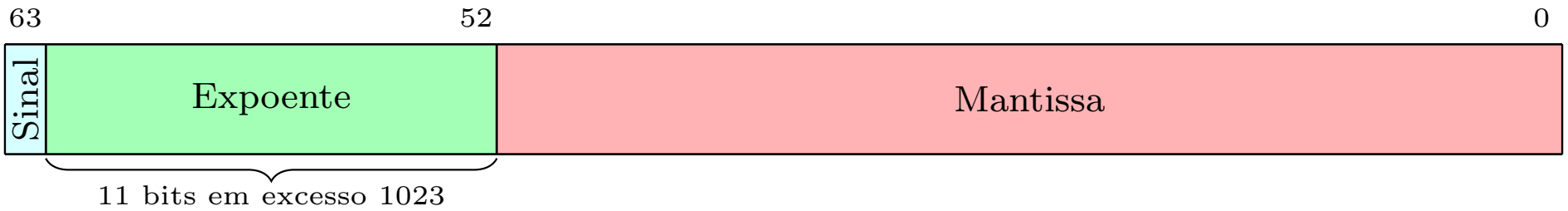
Nome	Nome popular	Bits na mantissa	Bits no expoente	Valor de polarização do Expoente	Expoente mínimo	Expoente máximo
binary16	meia precisão	10	5	$2^4 - 1 = 15$	-14	+15
binary32	precisão simples	23	8	$2^7 - 1 = 127$	-126	+127
binary64	precisão dupla	52	11	$2^{10} - 1 = 1023$	-1022	+1023
binary128	precisão quádrupla	112	15	$2^{14} - 1 = 16383$	-16382	+16383
binary256	precisão óctupla	236	19	$2^{18} - 1 = 262142$	-262142	+262143

Representação em ponto flutuante no padrão IEEE

Representação na memória para precisão simples



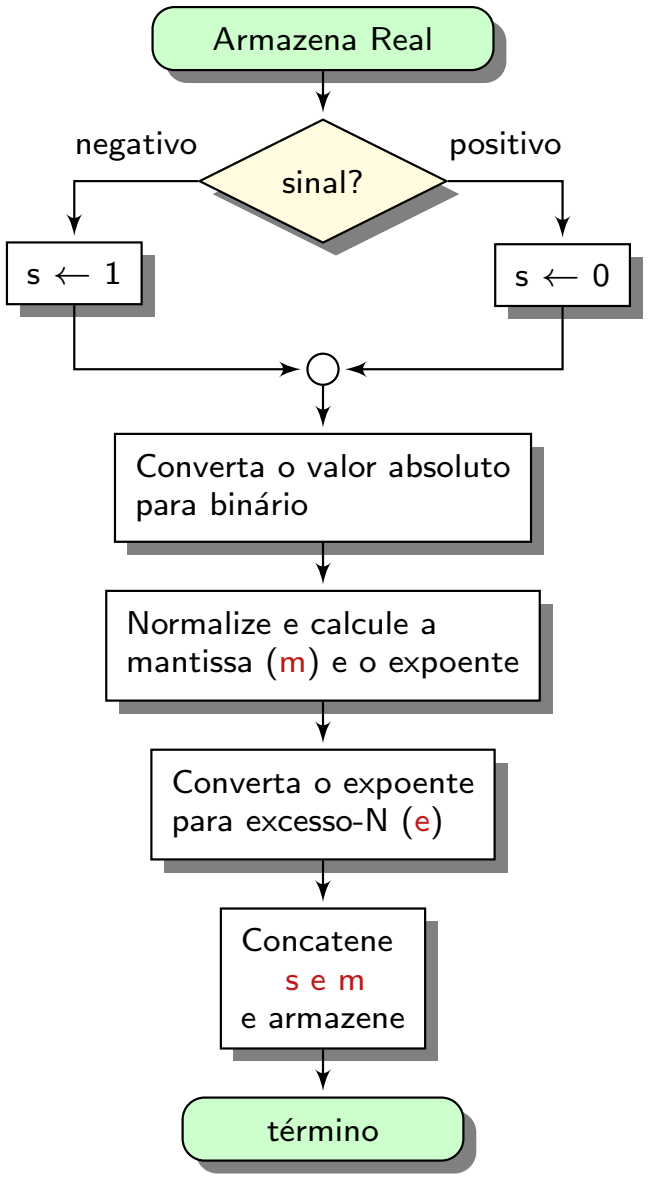
Representação na memória para precisão dupla

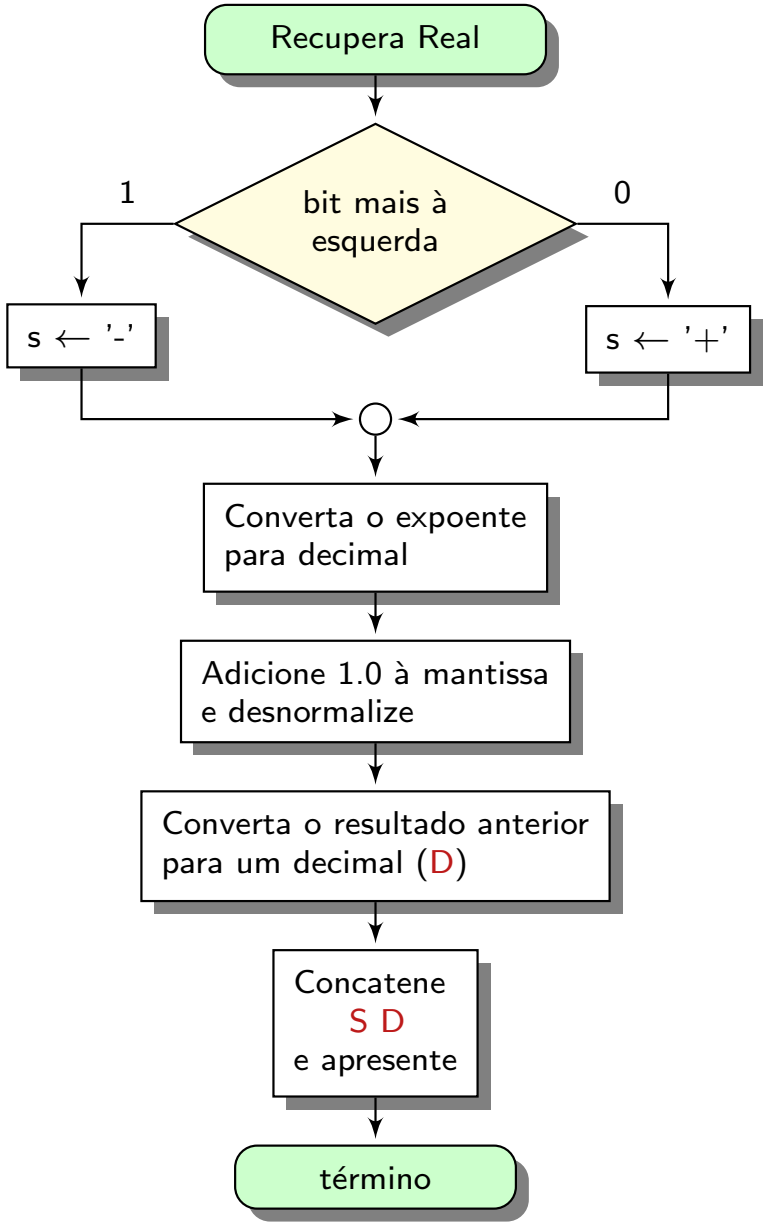


Algoritmos de armazenamento e recuperação

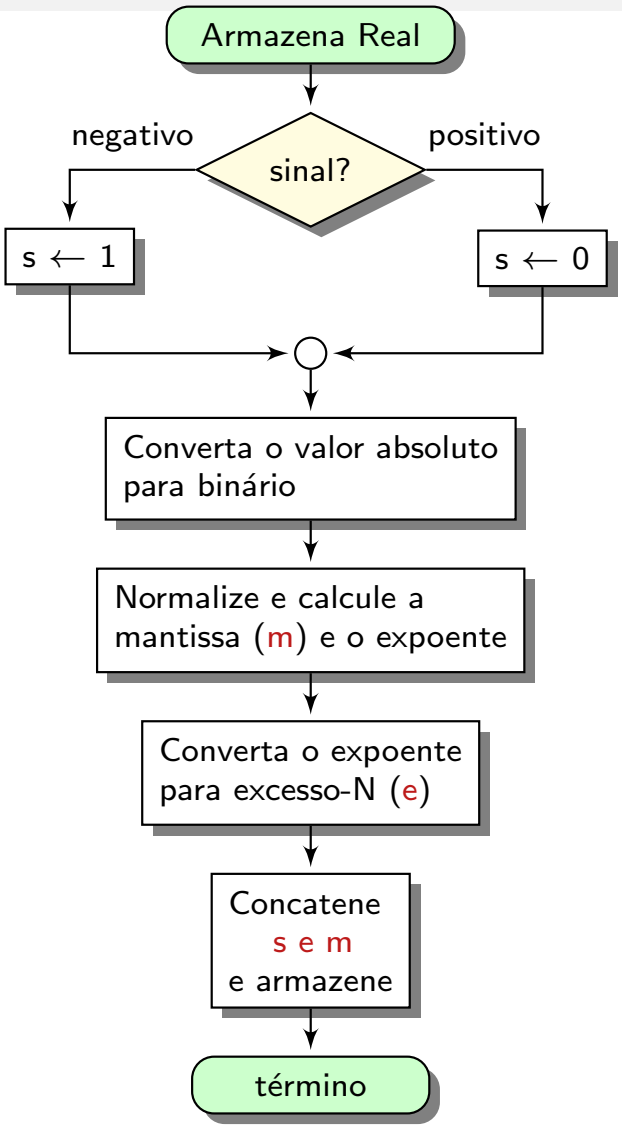
- Usando o padrão IEEE veremos dois algoritmos para armazenar e recuperar números reais.
- Esses algoritmos, mostrados no próximo slide, não dão uma ideia de como as funções de entrada e de saída, tais como `scanf` e `printf`, poderiam armazer e recuperar números reais.

Armazenando e recuperando em ponto flutuante





Exemplo de armazenamento usando o padrão IEEE com precisão simples

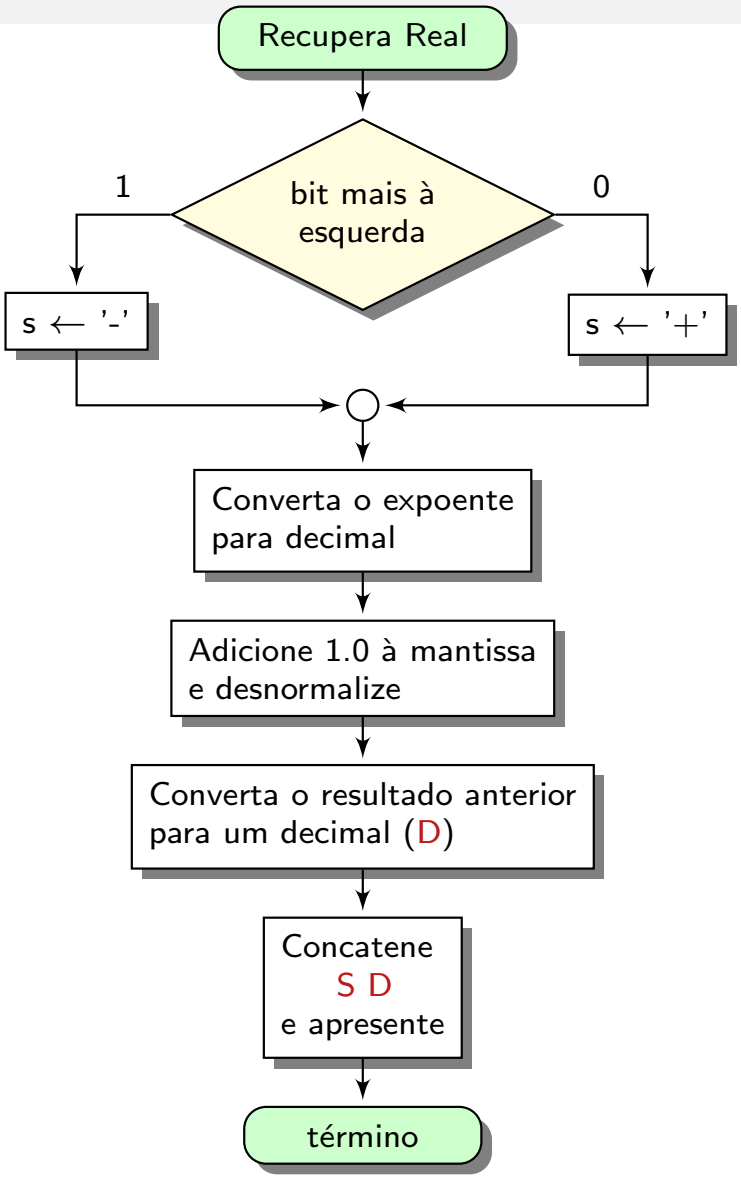


Usaremos o algoritmo ao lado para armazenar o número real 123.8125 em precisão simples.

1. O sinal é positivo: `s = 0`.
2. O valor absoluto em binário é : `1111011.1101`
3. Normalizamos este valor para `1.1110111101`. O valor de `m` em precisão simples com 23 bits é `m = 11101111010000000000000` e o expoente é 6.
4. O valor de `e` em precisão simples é `6 + 127 = 133` ou `10000101`.
5. Ao concatenarmos `s`, `e` e `m`, obteremos o valor a seguir:

0 10000101 11101111010000000000000

Exemplo de recuperação usando o padrão IEEE com precisão simples



Mostraremos como encontrar a representação decimal do seguinte número de 32 bits armazenado usando precisão simples:

1 10000010 00011000000000000000000

1. O bit mais à esquerda é 1: `s = '-'`
2. O valor dos próximos 8 bits é 130. Vamos subtrair 127 dele obtendo 3.
3. Adicionamos 1 e o ponto binário a `m` e deslocamos o ponto binário 3 dígitos para a direita, obtendo `1000.11` e ignorando os zeros finais.
4. Convertemos o número anterior para decimal obtendo `D = 8.75`.
5. Concatenamos `S` e `D`, obtendo `-8.75`

Para saber mais

- Forouzan, B. A and Gilbert, R. F. *Computer Science: a structured programming approach using C*. 3rd edition. Cengage Learning, 2007.
- IEEE 754. In: *Wikipedia, The Free Encyclopedia*. Disponível em `https://en.wikipedia.org/w/index.php?title=IEEE_754&oldid=840612631`.

Fontes

- Forouzan, B. A and Gilbert, R. F. *Computer Science: a structured programming approach using C*. 3rd edition. Cengage Learning, 2007.
- IEEE 754. In: *Wikipedia, The Free Encyclopedia*. Disponível em `https://en.wikipedia.org/w/index.php?title=IEEE_754&oldid=840612631`.