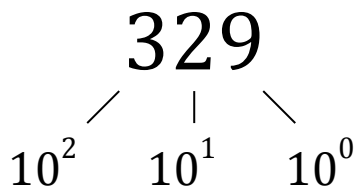
1. Bases de representação numérica: notação posicional pesada

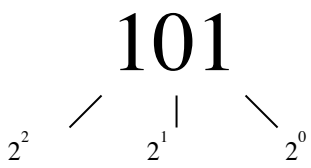
As quantidades numéricas são representadas usando um conjunto de símbolos ("algarismos") cujo valor ("peso") depende da posição que ocupam na representação da quantidade em causa.

1.1. Representação de inteiros

Como os numeros em decimal (base 10) : "329"
"3" vale 300, devido à posição que ocupa, enquanto "9" vale apenas 9



Base 2:



1x4 + 0x2 + 1x1 = 5

Num sistema de numeração posicional à posição de cada dígito é atribuído um peso. Para uma base $r \geq 2$, um dígito na posição i tem o peso r^i.

$r \geq 2$ # base
$d_i \in \{0,...,r\text{-}1\}$ # conjunto de símbolos (alfabeto)

1.2. Representação de quantidades não inteiras

$p + n$ – número de símbolos (p – parte inteira, n – parte fracionária)

Um número D cuja parte inteira inclui p dígitos e a parte fracionária – n dígitos pode ser representado como:

$$D = d_{p-1}d_{p-2}...d_1d_0.d_{-1}d_{-2}...d_{-n} = \sum_{i=-n}^{p-1} d_i * r^i$$

dígito mais significativo (***msb***) dígito menos significativo (***lsb***)

base	alfabeto
2	0,1
8	0, 1,…, 7
10	0, 1,…, 9
16	0, 1,…, 9, A, B, C, D, E, F

binário	decimal	octal	hexadecimal
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	8	10	8
1001	9	11	9
1010	10	12	A
1011	11	13	B
1100	12	14	C
1101	13	15	D
1110	14	16	E
1111	15	17	F

$2007_{10} = 2*10^3 + 0*10^2 + 0*10^1 + 7*10^0$ $\qquad D = \sum_{i=-n}^{p-1} d_i * 10^i$

$19.85_{10} = 1*10^1 + 9*10^0 + 8*10^{-1} + 5*10^{-2}$

$1100110_2 = 1*2^6 + 1*2^5 + 0*2^2 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0 = 64 + 32 + 4 + 2 = 102_{10}$

$101.0011_2 = 1*2^2 + 0*2^1 + 1*2^0 + 1*2^1 + 0*2^{-1} + 0*2^{-2} + 1*2^{-3} + 1*2^{-4}$ $\qquad D = \sum_{i=-n}^{p-1} d_i * 2^i$

$3577_8 = 3*8^3 + 5*8^2 + 7*8^1 + 7*8^0 = 1919_{10}$

$35.77_8 = 3*8^1 + 5*8^0 + 7*8^{-1} + 7*8^{-2}$ $\quad D = \sum_{i=-n}^{p-1} d_i * 8^i$

$2007_{16} = 2*16^3 + 7*16^0 = 8199_{10}$

$7D7_{16} = 7*16^2 + 13*16^1 + 7*16^0 = 2007_{10}$ $\quad D = \sum_{i=-n}^{p-1} d_i * 16^i$

$A.2C_{16} = 10*16^0 + 2*16^{-1} + 12*16^{-2}$

1.3 Mudança de base: parte inteira

A conversão de um número decimal N inteiro para qualquer outra base r pode ser realizada através de divisões sucessivas de N por r até que o resultado da divisão se torne menor que r

Conversão para binário
$13_{10} = ???_2$

	Quociente	Resto
13/2	6	**1 lsb**
6/2	3	**0**
3/2	**1**	**1 msb**

$25_{10} = 1011_2$

Conversão para hexadecimal
$26_{10} = ???_{16}$
$26_{10} =$ $_{16}$

1.4 Mudança de base: parte fracionária

A conversão da parte fracionária F de um número decimal para qualquer outra base r pode ser realizada através de multiplicações sucessivas da F por r até que seja atingida a precisão desejada.

Conversão para binário
$0.6875_{10} = ???_2$

	Parte inteira do produto	Parte fracionária do produto
0,6875 * 2	**1**	0,375
0,375 * 2	**0**	0,75
0,75 * 2	**1**	0,5
0,5 * 2	**1**	0

$0.6875_{10} = 0.1011_2$

Conversão para hexadecimal
$0.25_{10} = ???_{16}$
$0.25_{10} = 0.4_{16}$

E quando a parte fracionária do produto nunca se anula, quando se deve para a conversão?

A precisão da representação é de metade do algarismo menos significativo. No caso da base decimal, sendo **i** o numero de algarismos da parte fracionária:
Precisão = (½) * 10^{-i}
E em base 2, para uma parte fracionária com **j** bits: Precisão = (½) * 2^{-j} = $2^{-(j+1)}$
Portanto $2^{-(j+1)}$ = (½) * 10^{-i} 2^{-j} = 10^{-i}
j = i $\log_2$ 10 = 3,3219... * i
j será o maior inteiro menor que o resultado deste produto

1.3 Mudança de base: casos especiais

Quando é necessário converter um número da base r_1 para a base r_2 e se $\mathbf{r_1=r_2^x}$, então cada dígito da base r_1 pode ser convertido diretamente para x dígitos em base r_2.

Conversão de octal para binário
$r_1 = 8$, $r_2 = 2$, $8 = 2^3$
=> cada dígito octal pode ser representado por 3 dígitos binários
$753.6_8 = 111\ 101\ 011\ .\ 110_2$

Conversão de binário para hexadecimal
$r_1 = 16$, $r_2 = 2$, $16 = 2^4$
=> cada quatro dígitos binários correspondem a um dígito hexadecimal
$110\ 0101\ 1100_2 = 65C_{16}$

II – Representação da informação nos computadores digitais

Que tipos de dados é necessário representar num computador?
Numeros – inteiros com e sem sinal, reais, complexos, …
Texto – carateres, strings, …
Imagens – pixeis, cores, formas, …
Som
Valores Lógicos – verdadeiro, falso
…
Sistemas digitais são constituídos por circuitos que processam dígitos binários. Os códigos usados para representar os diferentes tipos de informação são pois códigos binários, isto é, que utilizam apenas 2 símbolos, 0 e 1.
Um tipo de dados é definido pelo conjunto de valores que variáveis desse tipo podem assumir e pelas operações que podem ser executadas sobre eles.

Tipo de dados "*unsigned integers*" (Inteiros não negativos)
Como são *representados* num computador?
sistema binário de numeração (base 2)
Que *operações* pode o computador executar sobre eles?
As operações aritméticas básicas: soma, subtração, multiplicação, divisão

Adição Base-2 – como para base-10
Somar da direita para a esquerda, propagando o transporte (*carry*):

10010	10010	11111	10111
+ 1001	+ 1011	+ 1	+ 111
11011	11101	100000	

Resultado excede a gama da representação - não é representável em 5 bits

Subtração Base 2
Sutrair da direita para a esquerda, propagando o transporte (*borrow*):

10010	**10010**	**11111**	**10111**
- 1001	**- 1011**	**- 1**	**- 111**
01001	**00111**	**11110**	

Inteiros com Sinal – possíveis 3 representações alternativas:

1. **Sinal e módulo** (*sign-magnitude*)
   ***Msb* – bit de sinal :** 0 para numeros positivos, 1 para negativos
   Restantes bits representam o modulo
   Gama de representação para n = 4: 1111, .., 1000, 0000, .., 0111
   -7 -0 0 7

2. **Complemento para 1** (*1's complement*)
   Para inteiros representados em n-bits: $\mathbf{-X = (2^{n-1}-1) - X}$
   Exemplo para n = 4: X = 0101 -> $-X = (2^4-1) - 0101 = 1111 - 0101 = 1010$
   i.e. complemento para 1 obtem-se invertendo (negando) cada bit
   Gama de representação para n = 4: 1000, .., 1111, 0000, .., 0111
   -7 -0 0 7

3. **Complemento para 2** (*2's complement*)
   Para inteiros representados em n-bits: $\mathbf{-X = 2^{n-1} - X}$
   2's complement = 1's complement + 1
   Exemplo para n = 4: X = 0101 -> $-X = 2^4 - 0101 = 10000 - 0101 = 1011$
   Gama de representação para n = 4: 1000, .., 1111, 0000, .., 0111
   -8 -1 0 7

Em qualquer das representações:
números positivos: msb = 0 números negativos msb = 1

Gama de representação:
sign-magnitude $(-2^{n-1}-1)$ a $(2^{n-1}-1)$

1's complement $(-2^{n-1}-1)$ a $(2^{n-1}-1)$

2's complement -2^{n-1} a $(2^{n-1}-1)$

Representação de zero:
em ***sign-magnitude*** e ***1's complement*** há duas representações para o zero:
***sign-magnitude* +0 = 0000; -0 = 1000**
***1's complement* +0 = 0000; -0 = 1111**
em ***2's complement*** a representação de zero é única: **0 = 0000**

Nas representações em *1's complement* e *2's complement*, ao contrário do que acontece na representação em sinal e módulo, o bit de sinal é um bit pesado (peso negativo):
1's complement: bit de sinal tem peso $(-2^{n-1}-1)$
Exemplo: $1011 = 1 * (-2^3 - 1) + 1 * 2^1 = -7 + 2 = -5$

2's complement: bit de sinal tem peso -2^{n-1}
Exemplo: $1010 = 1 * (-2^3) + 1 * 2^1 + 1 * 2^0 = -8 + 2 + 1 = -5$

Operações aritméticas básicas: adição e subtração

Adição e subtração são efetuadas mod 2^n, isto é, mantendo o número de bits da representação.

Sinal e módulo: bit de sinal tratado à parte

Adição: operandos com o mesmo sinal (ambos positivos ou ambos negativos) – somar os módulos: A = 0101 B = 0010 |A|+|B| = 101 + 010 =111
A + B = 0111
A = 1101 B = 1010 A + B = 1111
operandos com sinais opostos – subtrair o menor módulo ao maior
A = 1011, B = 0100
|B| - |A| = 001 A + B = 0001

Subtração: operandos com o mesmo sinal - subtrair o menor módulo ao maior
operandos com sinais opostos – somar os módulos

Complemento para 1 e **Complemento para 2:**

- bit de sinal tratado como os restantes bits
- **Subtrair é adicionar o complemento –** a unidade de soma efetua adições e subtrações. Desnecessário lógica distinta para a subtração.
- **Adição e subtração são efetuadas mod 2^n**

Complemento para 1

Adição: operandos com o mesmo sinal

ambos positivos A = 0101 B = 0010 A + B = 0111

ambos negativos A = 1101 B = 1010 A + B = 0111+1 = 1000 - "end-around carry"

1101	$(2^{n-1}-1) - \|A\|$
+ 1010	$(2^{n-1}-1) - \|B\|$
10111	$(2^{n-1}-2) - (\|A\| + \|B\|)$
1	1
1000	$(2^{n-1}-1) - (\|A\| + \|B\|)$

Subtração: somar ao subtraendo o complemento do subtrator

A – B = A + [$(2^{n-1}-1)$ – B]

- "end-around carry" duplica o tempo de execução das operações.

Complemento para 2

- Subtrair é somar o complemento
- Não existe "end-around carry"
- Usado em todos os computadores atuais para representar inteiros com sinal

Como detetar quando o resultado da adição (ou da subtração) excede a gama de representação (i.e. não pode ser representado em n bits)?

Sinal e módulo – se carry out da posição mais significativa = 1 o resultado excede a gama de representação

A = 1110 B = 1101

110
+ 101
1011 C_{out} **= 1** **erro**

Complemento para 2 – se os carry gerados nas duas duas posições mais significativas, forem diferentes o resultado excede ("*overflows*") a gama de representação: C_n **XOR** C_{n-1} **= 1** → **erro**

Overflow **=** C_n **XOR** C_{n-1}

A = 1010 **B = 1011**

```
  1010
+1011
  0101
```

Carries **1 0**

A = 0110 **B = 0111**

```
  0110
+0111
  1101
```

Carries **0 1**