



# ***Sistemas de Numeração***

Texto Didático/Pedagógico realizado no âmbito de apoio à disciplina de  
**Tecnologia da Informática**

*Álvaro Santos  
Anabela Gomes  
Cristina Chuva  
José Luís Nunes  
Luís Macedo  
Viriato Marques*

# Índice

1.	Generalidades .....	3
1.1.	O Conceito de <i>Bit</i> .....	3
1.2.	<i>Bytes</i> , <i>Words</i> e Outras Unidades.....	5
2.	Sistemas de Numeração.....	6
2.1.	Notação Posicional e Base.....	6
2.2.	Sistema Binário Natural.....	7
2.3.	Sistemas Hexadecimal e Octal.....	16
3.	Operações Aritméticas .....	19
3.1.	Adição .....	19
3.2.	Multiplicação .....	22
3.3.	Subtracção .....	25
3.4.	Divisão.....	27
3.5.	Aritmética Binária e Portas Lógicas .....	30
4.	Códigos Bipolares .....	31
4.1.	Sinal e Valor Absoluto .....	31
4.2.	Complementos de 1 .....	32
4.3.	Complementos de 2 .....	35
4.4.	Código Binário Deslocado ou Excesso $2^{n-1}$ .....	38
4.5.	Overflow.....	39
5.	Formatos dos números .....	41
5.1.	Vírgula Fixa.....	41
5.2.	Vírgula Flutuante e IEEE-754.....	43
6.	Outros códigos - Binário Codificado em Decimal .....	47
6.1.	BCD Natural ou 8421 .....	48
6.2.	Código 2421 .....	49
6.3.	Código 5421 .....	50
6.4.	Excesso 3.....	50
6.5.	Biquinário.....	50
6.6.	1 de 10 .....	51
7.	Códigos Binários para Tarefas Especiais .....	52
7.1.	Códigos para Visualização .....	52
7.2.	Códigos Contínuos .....	53
7.3.	Códigos para Detecção de Erros.....	56
8.	Representação de Caracteres .....	57

# Sistemas de Numeração

Na disciplina de Sistemas Digitais contactaram com os circuitos digitais como componentes de sistemas em que toda a informação é representada e tratada numericamente, servindo-se para isso apenas dos dígitos 1 e 0 que, electricamente, não são mais do que tensões compreendidas em intervalos cuja definição depende da tecnologia de construção dos circuitos integrados em consideração. O presente manual descreve como estes 0's e 1's são organizados de modo a representarem números e caracteres. Como se verá, existem vários códigos destinados a resolver os mais diversos problemas, desde a representação do simples número inteiro positivo aos números negativos, fraccionários e caracteres específicos de diferentes nacionalidades. Existem também sistemas de codificação vocacionados para situações particulares, tais como a representação de posições angulares e a minimização da taxa de erros na transmissão de informação. Definidos os códigos numéricos, torna-se possível realizar operações aritméticas, nomeadamente a adição, a subtração, a multiplicação e a divisão. Descrevem-se vários métodos para a realização destas operações.

## 1. Generalidades

### 1.1. O Conceito de *Bit*

A palavra *bit* é uma abreviatura de *binary digit*, isto é, uma designação abreviada para os algarismos 1 e 0, correspondentes aos valores lógicos T (True) e F (False) ou aos níveis H (High) e L (Low) dos circuitos digitais. Um *bit* assumirá sempre um dos valores 0 ou 1. De modo semelhante, para o sistema decimal existe o conceito de *decit*, abreviatura de *decimal digit*, que pode assumir os valores 0 a 9.

Contudo esta definição, algo simplista, carece de um significado mais profundo. O que traduz

exactamente um *bit*?

A chave para uma definição mais exacta encontra-se na palavra informação, conceito abordado matematicamente por Shannon em 1948 na Teoria da Informação. Esta teoria foi inspirada pela área das telecomunicações<sup>1</sup>, considerando sistemas emissores e receptores de mensagens. Como se viu, os circuitos integrados digitais também comunicam entre si e portanto comportam-se como emissores e receptores de mensagens digitais. Que informação, ou, melhor, que quantidade de informação contêm estas mensagens? Será possível medi-la?

A resposta à última questão é sim. Segundo Shannon, se um emissor puder enviar a um receptor  $n$  mensagens distintas com probabilidades de ocorrência  $p_1, p_2 \dots p_n$ , a quantidade de informação  $I_k$  contida na mensagem de ordem  $k$  cuja probabilidade de ocorrência é  $p_k$ , é dada por:

$$I_k = \log_2 \frac{1}{p_k} \quad [1]$$

O significado da Equação 1 pode perceber-se com um exemplo simples: se uma pessoa se dirigir a um espectáculo previamente agendado e lhe telefonarem da bilheteira a dizer que o espectáculo se realiza de facto, é evidente que a quantidade de informação contida nesta mensagem é quase nula, porque a probabilidade  $p_k$  de o espectáculo se realizar é praticamente 1. De facto, fazendo  $p_k = 1$  na Equação 1, obtém-se  $I_k = 0$ . Pelo contrário, se o telefonema informar que o espectáculo não se realiza devido a acidente de um dos actores, então sim, o conteúdo informativo de tal mensagem será alto por se tratar de um acontecimento inesperado, ou seja, de baixa probabilidade. Correspondentemente, tendo  $p_k$  um valor muito pequeno na Equação 1, o valor de  $I_k$  resultará elevado.

Considere-se agora a situação em que existem apenas duas mensagens possíveis,  $m_1$  e  $m_2$ , cujo conteúdo é 0 e 1, respectivamente. Suponha-se, além disso, que ocorrem com igual probabilidade de 1/2. Fazendo  $p_k = 1/2$  na Equação 1 obtém-se  $I_k = 1$ . Embora a Equação 1 seja adimensional, convencionou chamar-se a esta quantidade de informação, de valor 1, *bit*<sup>2</sup>.

---

<sup>1</sup> O artigo original foi publicado, aliás, no *The Bell System Technical Journal*.

<sup>2</sup> Como num sistema digital um determinado ponto não tem forçosamente que se encontrar ao nível 0 ou 1 com igual probabilidade, pode, se de interesse para a análise em curso, chamar-se *binít* ao valor 0 ou 1 que esse ponto apresenta, reservando o termo *bit* para a quantidade de informação contida em cada *binít*. Nestas condições, a informação contida

Um emissor que produz apenas e sempre a mesma mensagem é incapaz de transmitir qualquer informação porque gera sempre o mesmo conteúdo ( $p_k = 1$ ). Tudo se passa como se 2 fosse a quantidade mínima de mensagens distintas necessárias para que possa ocorrer alguma transmissão de informação. Como 1 *bit* pode assumir precisamente e apenas 2 valores distintos, 1 *bit* é também o suporte mínimo exigível para que possa ocorrer alguma transmissão de informação.

## 1.2. *Bytes, Words e Outras Unidades*

Um *bit*, por si só, é extremamente limitado, no sentido em que apenas pode assumir dois valores distintos, por exemplo, Ligado/Desligado, Sim/Não, “1”/”0”. No sistema decimal, por exemplo, um *decit* pode assumir 10 valores distintos e portanto representar 10 mensagens diferentes.

Não é possível lidar com um só *bit* quando se trata de representar números e caracteres, uma vez que a quantidade de diferentes números possíveis é infinita e os caracteres ortográficos atingem, considerando sinais de pontuação, acentos e particularidades de cariz internacional, um número bastante elevado. Além disso os computadores são máquinas especialmente adequadas ao tratamento de grandes volumes de informação: milhões de números, palavras, fórmulas, desenhos, etc. Para representar informação mais complexa há que agrupar diversos bits. As unidades mais vulgares são por isso as seguintes:

- *Byte* (ou octeto)      8 *bits*
- *Word* (ou palavra)    16 *bits*
- *Kbit* (Kilo-bit)        1024 *bits*
- *KByte* (Kilo-Byte)    1024 *bytes*
- *MByte* (Mega-Byte)   1024 × 1024 = 1.048.576 bytes ( $\cong$  1 milhão)
- *GByte* (Giga-Byte)    1048576 × 1024  $\cong$  1 bilião de *bytes*
- *TByte* (Tera-Byte)     $\cong$  1 trilião de *bytes*

Como se pode verificar, estas unidades são sempre potências de 2 – por exemplo,  $8 = 2^3$ ,  $16 = 2^4$  – ou obtêm-se a partir de outras multiplicando por 1024, o que não é surpresa dado que  $1024 = 2^{10}$  e portanto também é potência de 2. O porquê destes factos tornar-se-á claro no decorrer deste

---

num *binit* de valor 0 poderá ser superior ou inferior à contida no mesmo *binit* quando apresentar o valor 1, ou vice-versa. No âmbito deste manual a diferença entre *bit* e *binit* é irrelevante.

capítulo.

## 2. Bases para representação de valores

### 2.1. Notação Posicional e Base

Um grupo de dígitos binários permite representar qualquer tipo de informação discreta. A lei que define uma determinada correspondência entre um código binário único e cada informação que se pretende representar no computador é designada por código. A memória de um computador contém exclusivamente 0's e 1's. O seu significado depende do código que lhes é associado em cada momento. Um determinado código binário só é interpretável se tiver associado um determinado contexto. Qual é então o código usado para representar valores numéricos num computador? Na realidade não existe um único código. Existem diversos esquemas para realizar essa codificação. Na escrita vulgar de números utiliza-se a notação posicional, uma *notação* em que o valor de um dígito depende da sua posição no número em que figura. Nesta notação, dado um número qualquer, cada dígito representa um valor igual ao resultado do seu produto por uma potência de  $B$ , em que  $B$  é a *base do sistema de numeração* considerado. Para o sistema decimal,  $B = 10$ . Por exemplo, no número 209 o algarismo 2 tem um peso de 100 enquanto o algarismo 9 tem um peso de 1 uma vez que:

$$\begin{aligned} 209 &= ? \\ &= 2 \times 10^2 + 0 \times 10^1 + 9 \times 10^0 \\ &= 200 + 9 \end{aligned}$$

O número 10 que intervém nesta expressão através das suas sucessivas potências constitui a base do sistema de numeração que utilizámos – neste caso o sistema de base 10 ou sistema decimal. Note-se que a base também pode ser elevada a um expoente negativo. Por exemplo:

$$\begin{aligned} 104.5 &= ? \\ &= 1 \times 10^2 + 0 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1} \\ &= 100 + 0 + 4 + 0.5 \end{aligned}$$

Na sua forma geral, um número  $N$  expresso em notação posicional na base  $B$  é um somatório do tipo

$$N = \sum_{i=-n}^{p-1} d_i \cdot B^i \quad [2]$$

com  $N$  valor representado

$n$  número de dígitos à direita do ponto

$p$  número de dígitos à esquerda do ponto

$d$  um dígito

$B$  base do sistema de numeração considerado

Pode pensar-se num sistema de base 5, ou 19, ou outra qualquer. Mas, por razões evolutivas, históricas ou técnicas, os que ficaram para a posteridade foram os de base 2, 8, 16, 10, 12, 20 e 60. O de base 10 é sobejamente conhecido por uma razão muito simples: a utilização das mãos auxiliava a contagem. O de base 12 tem uma excelente representação no sistema monetário inglês. Do de base 20 encontra-se apenas uma reminiscência no termo *quatre-vingt* da língua francesa, crendo-se que esta base proveio do facto de certas tribos primitivas utilizarem os dedos das mãos e dos pés para contar. O de base 60 figura nos nossos relógios. Restam os de base 2, 8 e 16 que se apresentam nas secções seguintes.

## 2.2. Sistema Binário Natural

### Notação

No *sistema binário* ou *sistema de base 2*, assim designado por utilizar dois dígitos: 0 e 1, o *bit* que figura mais à direita tem o menor peso, enquanto que o situado mais à esquerda possui o maior peso. O sistema binário utiliza, portanto, uma notação posicional. Como a Equação 1 é aplicável a qualquer notação posicional de base  $B$ , para o caso particular do sistema binário basta fazer  $B = 2$  obtendo-se:

$$N = \sum_{i=-n}^{p-1} d_i \cdot 2^i \quad [3]$$

Os seguintes são exemplos de números binários: 10110101, 101, 001011.101, 0, 1, 10, 100. Tal como no sistema decimal, os 0 à esquerda do primeiro 1 antes da vírgula não têm qualquer significado, isto é,  $001011.101 = 1011.101$ . De facto, a Equação 2. dará para estes 0's parcelas de valor nulo que não influenciam, por isso, o somatório N. Contudo, os sistemas digitais reais utilizam  $m$  dígitos na representação de qualquer número, uma vez que as estruturas físicas subjacentes a estas representações, tais como unidades de memória, têm uma certa "largura" fixa. Por isso as representações reais incluem, de facto, os 0's não significativos, embora eles não influenciem os resultados das operações aritméticas e lógicas, como se verá.

## Conversão Decimal/Binário para Números Inteiros

Devido à existência de várias bases de numeração e à necessidade de se trabalhar, por vezes, com várias em simultâneo, é vulgar colocar-se a base ou um seu símbolo –  $D$  para decimal,  $B$  para binário – como um índice à direita do número em consideração. Por exemplo,  $234_{10}$  ou  $234_D$  significa 234 na base 10. Pelo facto de o sistema de base 10 ser o mais utilizado no dia-a-dia, na prática omite-se a base deste sistema de numeração.

Nestas condições, e atendendo à Equação 2, o número  $10011_B$  não representa *dez mil e onze* mas sim

$$\begin{aligned} 10011_B &= ?_D \\ &= 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 16 + 2 + 1 \\ &= 19_D \end{aligned}$$

e o número  $10011.101_B$  representa

$$\begin{aligned} 10011.101_B &= ?_D \\ &= 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 16 + 2 + 1 + 0.5 + 0.125 \\ &= 19.625_D \end{aligned}$$

Generalizando, para converter um número numa determinada base  $N$  para o número decimal equivalente, basta multiplicar cada dígito do número pela potência dessa base (relativa à posição



por ele ocupada) e somar os resultados.

## Gama de Representação

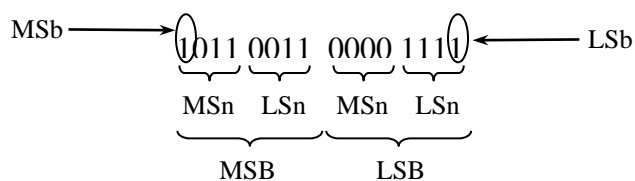
Um *bit* permite apenas representar duas situações diferentes, por exemplo, ligado e desligado. Para representar maior quantidade de informação fazem-se associações de *bits*. Assim, com 2 *bits* é possível representar quatro situações diferentes: 00, 01, 10 e 11. Extrapolando para  $n$  *bits*, podem representar-se  $2^n$  situações diferentes, correspondentes aos números inteiros no intervalo  $[0, 2^n - 1]$ .

## Terminologia

Num número binário designa-se por MSb (*Most Significant bit*) o dígito de *mais alta ordem*, ou seja, o situado mais à esquerda, tendo por isso o maior peso; e por LSb (*Least Significant bit*) o dígito de *mais baixa ordem*, ou seja, o situado mais à direita e por conseguinte o de menor peso.

Se o número ocupar vários *bytes*, utilizam-se as siglas MSB (*Most Significant Byte*) e LSB (*Least Significant Byte*) para indicar respectivamente o *byte* mais à esquerda e o *byte* mais à direita.

Dentro de um mesmo *byte*, cada conjunto de 4 *bits* constitui um *nibble*, podendo usar-se as siglas MSn (*Most Significant nibble*) e LSn (*Least Significant nibble*) para indicar os *nibbles* mais e menos significativos, respectivamente.



**Figura 1** – Siglas associadas a um número binário de 16 *bits*

- Um *nibble* pode representar números inteiros no intervalo  $[0, 15_D]$ ;
- Um *byte* pode representar números inteiros no intervalo  $[0, 255_D]$ ;
- Uma *word* pode representar números inteiros no intervalo  $[0, 65535_D]$ .

## Conversão Decimal/Binário para Números Inteiros

Para converter números decimais noutros sistemas, como por exemplo o binário existem vários métodos. O mais frequente consiste em separar a parte inteira e a parte fraccionária, tratando cada uma delas segundo os métodos de seguida apresentados.

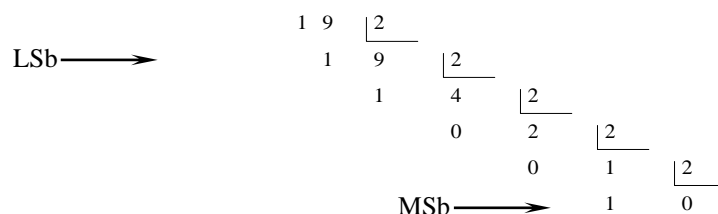
### Método das Divisões Sucessivas

A Equação 2 é a base para conversão de números binários a números decimais. A conversão de um número inteiro decimal a binário, realiza-se à custa do *método das divisões sucessivas* por 2. Aplicando este método a um número  $N_D$ , chega-se ao seguinte algoritmo:

#### Algoritmo

1.  $i = 0$ ;
2. Enquanto  $N_D > 0$  {
  - a. Fazer a divisão inteira de  $N_D$  por 2;
  - b. O resto da divisão é o *bit* de ordem  $i$  do número pretendido na base 2 (na primeira iteração obtém-se o *bit* menos significativo);
  - c. Tomar o quociente obtido como novo dividendo,  $N_D$ ;
  - d. Fazer  $i = i + 1$ ;
  - e. Voltar ao passo 2 }
3. FIM

Por exemplo, para converter o número  $19_D$  a binário procede-se assim:



Donde,  $19_D = 10011_B$ . Note-se que:

- O processo termina quando o quociente for 0;
- O primeiro resto obtido é o LSb do número pretendido;
- O último resto obtido, sempre de valor 1, é o MSb do número pretendido.

O método das divisões sucessivas é aplicável a qualquer base  $B$ , bastando para isso considerar divisores de valor  $B$  em vez de valor 2.

### Método Heurístico

Com alguma prática é possível realizar uma conversão decimal/binário por um processo heurístico. Nesta abordagem basta simplesmente atender aos valores das potências de 2 previamente conhecidos e ao valor do número a converter. Por exemplo, para converter  $193_D$  a binário, pode fazer-se assim:

1. Qual a maior potência de 2 imediatamente inferior ou igual a  $193_D$ ?
2. É  $2^7 = 128_D$ . Escreve-se então um 1 na posição 7, obtendo-se: 1xxx xxxx;
3. Calcula-se  $193_D - 128_D = 65_D$ ;
4. Procede-se como no ponto 1 utilizando agora o valor  $65_D$  e obtendo-se como resposta  $2^6 = 64$ ;
5. Escreve-se um 1 na posição 6, obtendo-se: 11xx xxxx;
- 6.
7. Calcula-se  $65 - 64 = 1$ ;
8. Repete-se o ponto 1 utilizando o valor  $1_D$  e obtendo-se  $2^0 = 1$ ;
9. Escreve-se um 1 na posição 0 e preenchem-se com 0's as posições restantes. Obtém-se o resultado final: 1100 0001 $_B$ .

A Tabela 1 esquematiza este procedimento. Note-se que este processo heurístico também pode ser utilizado para conversão a outras bases,  $B$ , além da binária, bastando para isso substituir as potências  $2^N$  por  $B^N$ .

	Binário								Decimal
Potências	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	
Pesos	128	64	32	16	8	4	2	1	
	1	0	0	0	0	0	0	0	128
	0	1	0	0	0	0	0	0	64
	0	0	0	0	0	0	0	1	1
	1	1	0	0	0	0	0	1	193

**Tabela 1** – Método heurístico de conversão de decimal a binário

## Conversão Decimal/Binário para Partes Fraccionárias

O número  $19_D$  é um número inteiro. Porém, para partes fraccionárias o método das divisões sucessivas não resulta. Basta considerar o número  $0.5_D$ : a aplicação do processo acima indicado daria como resultado sucessivos restos de valor 0.5 não se vislumbrando, sequer, quando deveria terminar.

Para a conversão das partes fraccionárias de um número decimal para outra base, utiliza-se o *método das multiplicações sucessivas*. Seja o número fraccionário  $.N_D$ :

### Algoritmo

1.  $i = -1$ ;
2. Multiplicar  $.N_D$  por 2;
3. A parte inteira do produto é o *bit* de ordem  $i$  do número pretendido na base 2 (na primeira iteração obtém-se o primeiro *bit* à direita do ponto decimal);
4. Tomar a parte fraccionária do produto como um novo multiplicando,  $.N_D$ ;
5. Se a parte fraccionária for diferente de 0 {
  - a. Se a precisão do número convertido não for ainda suficiente {
    - i.  $i = i-1$ ;
    - ii. Voltar ao passo 2; }}
6. FIM

A primeira questão que ressalta deste algoritmo é a sua *regra de paragem* (passos 5 e 5a). O que se entende por precisão suficiente? Vejam-se os dois exemplos seguintes:

### Exemplo 1

Seja o número a converter  $0.625_D$ . Neste caso obtem-se:

$$\begin{array}{l} \xrightarrow{\text{MSb}} \\ 0.625 \times 2 = 1.250 \\ 0.250 \times 2 = 0.500 \\ 0.500 \times 2 = 1.000 \\ 0.000 \times 2 = 0.000 \\ 0.000 \times 2 = 0.000 \\ \xleftarrow{\text{LSb}} \end{array}$$

É evidente que por mais que se prolonguem as multiplicações o resultado será sempre o mesmo porque a parte decimal de  $0.000 \times 2$  é sempre 0. Portanto,  $0.625$  é exactamente representável por  $0.101_B$ . Isto pode verificar-se recorrendo à Equação 2:

$$\begin{aligned} 0.101_B &= ?_D \\ &= 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 0.5 + 0 + 0.125 \\ &= 0.625_D \end{aligned}$$

### Exemplo 2

Seja o número a converter  $0.637_D$ . Neste caso obtem-se:

$$\begin{array}{ll} \xrightarrow{\text{MSb}} & \\ 0.637 \times 2 = 1.274 & 0.384 \times 2 = 0.768 \\ 0.274 \times 2 = 0.548 & 0.768 \times 2 = 1.536 \\ 0.548 \times 2 = 1.096 & 0.536 \times 2 = 1.072 \\ 0.096 \times 2 = 0.192 & 0.072 \times 2 = 0.144 \\ 0.192 \times 2 = 0.384 & 0.144 \times 2 = 0.288 \\ & \dots \xleftarrow{\text{LSb}} \end{array}$$

O número obtido é portanto  $0.1010001100_B$ .

## Precisão

É evidente que ao prolongar-se a sucessão de multiplicações haverá um ponto em que, de novo, o produto de uma parte fraccionária por 2 gerará um novo dígito de valor 1, tal como aconteceu em 1.536 e em 1.072. A regra é: quantos mais dígitos se obtiverem por este processo, maior será a precisão da representação obtida. Por exemplo, considerando apenas os primeiros três dígitos tem-se  $0.101_B = 0.625_D$  que é um valor próximo de  $0.637_D$  mas, diga-se, ainda com um erro considerável. Porém, recorrendo aos dez dígitos acima obtidos, tem-se:

$$\begin{aligned} 0.1010001100_B &= ?_D \\ &= 1 \times 2^{-1} + 1 \times 2^{-3} + 1 \times 2^{-7} + 1 \times 2^{-8} \\ &= 0.5 + 0.125 + 0.0078125 + 0.00390625 \\ &= 0.63671875 = \\ &\cong 0.637_D \end{aligned}$$

Como se pode verificar, o número  $0.63671875_D$ , resultante da representação através de dez *bits* (dos quais apenas oito são significativos), encontra-se muito mais próximo do valor  $0.637_D$ . Na realidade, se se arredondar  $0.63671875_D$  a três casas decimais, o número  $0.637_D$  já é reconstituído dado que o dígito de ordem 4 é superior a 5.

É por esta razão que os sistemas digitais permitem aumentar a precisão das representações à custa, simplesmente, do número de *bits* utilizados nessas representações. Com um *número infinito de bits* obter-se-ia um erro nulo. Afinal, quando deve parar o processo de conversão a binário da parte fraccionária de um número decimal?

- Em termos práticos deve parar quando o número máximo de *bits* disponíveis para a representação for atingido. Este número máximo é ditado pelo sistema em consideração e é normalmente uma potência de 2 tal como 8, 16, etc.
- Em termos matemáticos, deve parar quando a representação binária obtida garantir uma precisão maior ou igual àquela que o número, expresso em decimal, possuir.

Suponha-se o número  $0.637_D$ . O último dígito deste número é de ordem 3 e portanto permite representações com uma precisão de milésima, uma vez que  $10^{-3} = 0.001$ . Isto significa que o número que se pretende obter na nova base deve ter uma quantidade de dígitos suficientes para

garantir que o último deles consegue representar valores inferiores a 0.001 (iguais será impossível, dado que 0.001 não é potência de 2). Como os valores representados são do tipo  $B^{-m}$ , em que  $B$  é a base e  $m$  a posição do dígito contada do ponto decimal para a direita, a condição genérica é:

$$B^{-q} < 10^{-p} \quad [4]$$

com  $B$  nova base

$q$  posição do último dígito a ser considerado

$p$  posição do último dígito do número na base 10

Aplicando logaritmos obtém-se:

$$\log_{10} B^{-q} < -p$$

Atendendo a que

$$\log_b x = \log_b a \cdot \log_a x \quad [5]$$

tem-se, com  $a = B$ ,  $b = 10$  e  $x = B^{-q}$

$$\log_{10} B \cdot \log_B B^{-q} < -p$$

$$-q \cdot \log_{10} B < -p$$

$$q \cdot \log_{10} B \geq p$$

$$q \geq \frac{p}{\log_{10} B} \quad [6]$$

com  $q$  número de dígitos do número convertido, à direita do ponto

$p$  número de dígitos do número original, à direita do ponto

$B$  base do novo sistema de numeração

Aplicando a Equação 6 ao número  $0.637_D$  obtém-se:

$$q \geq \frac{3}{\log_{10} 2}$$

$$q \geq 9.96$$

Portanto terão de ser usados pelo menos 10 dígitos binários. De facto:

$$\frac{1}{2^{-10}} = \frac{1}{1024} = 0.0009765 < \frac{1}{1000}$$

pelo que o décimo dígito é o primeiro que consegue representar valores inferiores a  $0.001_D$ .

### 2.3. Sistemas Hexadecimal e Octal

O *sistema hexadecimal*, usa 16 algarismos ou dígitos, e daí a sua designação. Portanto, para além dos 10 dígitos utilizados na base decimal, surge a necessidade de introduzir 6 símbolos adicionais, que são: A, B, C, D, E e F. Por sua vez, o *sistema octal*, ou de base 8, utiliza 8 dígitos: 0, 1, 2, 3, 4, 5, 6 e 7. Logo, um sistema de base  $B$  recorre a  $B$  símbolos para efectuar as suas representações.

A razão de ser destes sistemas é que facilitam o agrupamento de *bits* proporcionando representações mais condensadas, que tornam mais fácil a manipulação de números binários extensos. Como as suas bases são 16 e 8, ambas potências de 2, estes agrupamentos são, respectivamente, de 4 e de 3 *bits*. A Tabela 2 representa os números decimais de 0 a 16 e os seus correspondentes em binário, hexadecimal e octal.

De acordo com esta tabela, o número  $1110\ 0011_B$  representa-se, em hexadecimal, por  $E3_H$ . Para obter a sua representação em octal convém agrupar os dígitos em conjuntos de 3; obtém-se  $11\ 100\ 011_B$ . Portanto, em octal será  $343_8$ .

Naturalmente que estas conversões também se podem realizar no sentido inverso, isto é, de hexadecimal ou octal para binário. Em suma, conversões entre as bases 2, 8 e 16 são imediatas nos dois sentidos e isto acontece por 8 e 16 serem ambos potências de 2. Genericamente também se pode dizer que as conversões são imediatas entre bases que são potências de 2.

As conversões de hexadecimal e octal a decimal e vice-versa seguem os princípios expostos na secção anterior a respeito do sistema binário. A Equação 1 toma as seguintes formas:

Conversão Hexadecimal/Decimal: 
$$N = \sum_{i=-n}^{p-1} d_i \cdot 16^i \quad [7]$$



Conversão Octal/Decimal:

$$N = \sum_{i=-n}^{p-1} d_i \cdot 8^i \quad [8]$$

Decimal	Binário	Hexadecimal	Octal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17
16	10000	10	20

**Tabela 2** - Os números 0 a 16 em decimal, binário, hexadecimal e octal

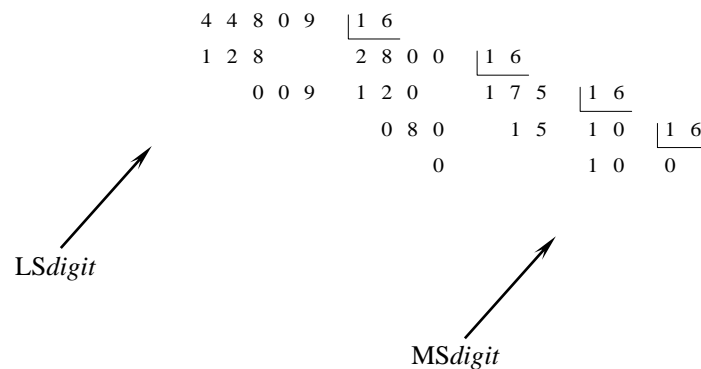
Assim, para converter o número  $AF09.8_H$  a decimal faz-se:

$$\begin{aligned}
 AF09.8_H &= ?_D \\
 &= 10 \times 16^3 + 15 \times 16^2 + 0 \times 16^1 + 9 \times 16^0 + 8 \times 16^{-1} \\
 &= 40960 + 3840 + 9 + 0.5 \\
 &= 44809.5_D
 \end{aligned}$$

O mesmo resultado se obteria se o número fosse previamente convertido a binário e depois a decimal. Contudo, repare-se como seria pouco confortável trabalhar nestas condições, dado que  $AF09.8_H = 1010\ 1111\ 0000\ 1001.1000_B$ , o que originaria o cálculo de 20 produtos em vez dos 5 acima utilizados.

As conversões de decimal para hexadecimal e octal realizam-se também de acordo com os algoritmos e princípios expostos na secção anterior. No caso do sistema hexadecimal apenas há que ter em atenção a substituição dos restos maiores que 9 pelos símbolos próprios deste sistema, isto é: 10 = A, 11 = B...15 = F. Assim, para converter o número  $44944.5_D$  a hexadecimal, faz-se:

## Parte inteira



Os restos de valor superior a 9 são, neste caso, 15 e 10, representados em hexadecimal respectivamente por F e por A. Portanto, a parte inteira resulta em  $AF09_H$ .

## Parte fraccionária

$$0.5 \times 16 = 8.0$$

$$0.0 \times 16 = 0.0$$

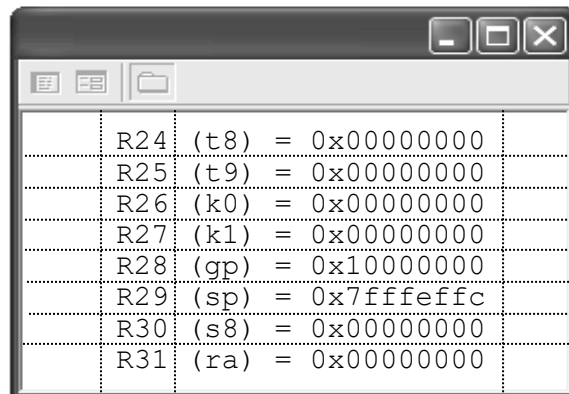
...

Donde a parte fraccionária é composta apenas pelo dígito  $8_H$ . Portanto,  $44809.5_D = AF09.8_H$ .

Para a conversão de partes fraccionárias com garantia de determinada precisão continua a ser válida a Equação 2 bastando fazer  $B = 16$  ou  $B = 8$  consoante se trate de conversão a hexadecimal ou a octal, respectivamente.

O sistema hexadecimal é hoje muito mais vulgar que o octal. É, por exemplo, o padrão quando se utilizam programas designados por *assembladores* (*assemblers*) que transformam linguagem simbólica de baixo nível, contudo minimamente inteligível para os humanos (*código fonte*), em sequências de 0's e 1's que apenas os processadores conseguem perceber (*código executável*). Os assembladores são geralmente acompanhados por uma ferramenta designada por *debugger*, que permite testar a execução dos programas examinando, por exemplo, o conteúdo da memória do computador e de *registos do processador*. Normalmente os valores em hexadecimal são assinalados por um  $0x$  que precede o número. Na Figura 2 apresenta-se o aspecto de um *debugger* em execução, mostrando o conteúdo dos registos R24 a R31 de um processador. Como se pode observar, tratam-

se de registos de  $8 \times 4 = 32$  bits, dado que o conteúdo de cada um é composto por 8 dígitos hexadecimais. Por exemplo o registo R29 contém o valor  $7FFFEFFC_H$ .



R24	(t8)	=	0x00000000
R25	(t9)	=	0x00000000
R26	(k0)	=	0x00000000
R27	(k1)	=	0x00000000
R28	(gp)	=	0x10000000
R29	(sp)	=	0x7fffeffc
R30	(s8)	=	0x00000000
R31	(ra)	=	0x00000000

**Figura 2** - Conteúdo de alguns registos de um processador em hexadecimal

O sistema octal era muito popular há algumas décadas atrás porque alguns computadores possuíam luzes e interruptores arranjados em grupos de 3. Pode dizer-se que caiu em desuso dado que hoje em dia a organização interna dos processadores é em múltiplos de 8 bits (*bytes*) pelo que o hexadecimal se torna muito mais prático. No sistema operativo Unix persistem algumas referências ao octal, como por exemplo no comando *od*, iniciais de *Octal Dump*.

Uma das utilizações actuais para o sistema octal continua a ser a especificação de caracteres especiais através do seu código, utilizada nas strings de formatação da linguagem C (por exemplo, na função *printf*).

### 3. Operações Aritméticas

As operações aritméticas podem ser realizadas em qualquer base utilizando os princípios elementares da adição, subtracção, multiplicação e divisão usados na base 10. Contudo o sistema binário, pela sua simplicidade, dá origem a algumas particularidades que permitem executar estas operações por algoritmos mais expeditos.

#### 3.1. Adição

Sejam por exemplo os números  $65_D$  e  $124_D$ . A adição em binário, hexadecimal e octal realiza-se da seguinte forma:

Decimal	Binário	Hexadecimal	Octal
---------	---------	-------------	-------

	1		
6 5 <sub>D</sub>	1 0 0 0 0 0 1 <sub>B</sub>	4 1 <sub>H</sub>	1 0 1 <sub>8</sub>
+ 1 2 4 <sub>D</sub>	+ 1 1 1 1 1 0 0 <sub>B</sub>	+ 7 C <sub>H</sub>	+ 1 7 4 <sub>8</sub>
1 8 9 <sub>D</sub>	1 0 1 1 1 1 0 1 <sub>B</sub>	B D <sub>H</sub>	2 7 5 <sub>8</sub>

Explica-se em seguida, pormenorizadamente, como se efectuam cada uma destas operações. Quanto à adição binária, basta ter em conta a Tabela 3:

Adição Binária		
Parcelas	Transporte	Soma
0 + 0	0	0
1 + 0	0	1
0 + 1	0	1
1 + 1	1	0

**Tabela 3** – Adição binária

Nesta tabela apenas a última linha poderá suscitar dúvidas. Porém, uma análise mais atenta revela que tudo se passa como na base 10: quando a adição de dois algarismos iguala ou supera a base em uso (por exemplo  $3_D + 7_D = 10_D$ ) escreve-se o dígito de menor peso do resultado (0) e gera-se um transporte (*carry*), o vulgar “e vai um”; este 1 escreve-se à esquerda do 0 e o resultado final é  $10_D$ . Na adição binária é exactamente isto que se passa: como o resultado de  $1+1 = 2_D$  iguala o valor da base,  $2_D$ , escreve-se um 0 e gera-se o transporte 1 que se coloca à sua esquerda; o resultado final é  $10_B$ .

Veja-se agora a adição  $11001111_B + 11001111_B$  em que este princípio é aplicado por várias vezes, conforme se ilustra em seguida:

$$\begin{array}{r}
 \text{Transportes} \longrightarrow \begin{array}{ccccccc} 1 & & 1 & 1 & 1 & 1 & \end{array} \\
 \begin{array}{r} 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1_B \\ + \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1_B \\ \hline 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0_B \end{array}
 \end{array}
 \qquad
 \begin{array}{r} a \\ + \ b \\ \hline S \end{array}$$

Como se pode verificar, a existência ou não de um *carry* num determinado dígito de ordem  $n$  (proveniente da adição dos dígitos de ordem  $n-1$ ), depende do conhecimento prévio do resultado da adição de ordem  $n-1$ . Como consequência disto, em última análise o resultado de uma adição de  $n$  dígitos só pode ser conhecido depois de o *carry* eventualmente gerado na adição dos dígitos de ordem 0 se ter propagado, ou não, através de todos os outros até atingir o de ordem  $n$ . Para acelerar este processo existem métodos que conseguem prever a existência ou não de um *carry* e que são implementados por circuitos integrados desenhados para tal função. Estes circuitos chamam-se geradores de transporte antecipado ou *look-ahead-carry-generators*.

Para adicionar números hexadecimais pode proceder-se exactamente da mesma forma. Por exemplo, a adição anterior pode escrever-se:

$$\begin{array}{r}
 \text{Transportes} \longrightarrow \begin{array}{ccc} 1 & 1 & 0 \end{array} \\
 \begin{array}{r} C \ F_H \\ + \ C \ F_H \\ \hline 1 \ 9 \ E_H \end{array}
 \end{array}$$

Em português corrente a adição anterior pode ser descrita por: “F mais F igual a E e vai 1; 1 mais C igual a D; D mais C igual a 9 e vai 1”. Assim, tal como as adições noutra base qualquer, sempre que o resultado da adição de duas parcelas for igual ou superior à base em causa, subtrai-se a base ao resultado obtido e faz-se o transporte (“e vai 1”) para a coluna seguinte. Neste exemplo tem-se:  $F+F = 15+15 = 30$ ; como 30 é maior que a base, 16, faz-se  $30-16 = 14$  ( $E_H$ ) “e vai 1”; de seguida calcula-se  $C+C+1 = 12+12+1 = 25$ ;  $25-16 = 9$  “e vai 1”; no final obtém-se  $CF_H+CF_H = 19E_H$ .

Contudo, é realmente difícil pensar em termos de adição de letras (contar pelos dedos pode ajudar!). Por isso e por a conversão hexadecimal/binário ser imediata, é vulgar converter os operandos a binário, fazer a soma e reconverter a hexadecimal. A calculadora do Windows, quando configurada

em modo científico, também realiza operações nas bases 2, 8 e  $16^3$ .

A adição em octal efectua-se de forma semelhante, tendo em atenção que existe transporte sempre que a soma de dois dígitos igualar ou ultrapassar o valor 8.

### 3.2. Multiplicação

Apresenta-se apenas a multiplicação em binário, aquela que tem verdadeiro interesse do ponto de vista dos sistemas digitais. Para realizar multiplicações nas bases 8 e 16 pode sempre converter-se a binário – o que é imediato – efectuar a operação e reconverter à base original. Descrevem-se em seguida três métodos para a realização de multiplicações em binário.

#### Multiplicação por Adições Sucessivas

O primeiro consiste simplesmente em adicionar tantas vezes o multiplicando quantas o valor do multiplicador. Por exemplo para a multiplicação  $9_D \times 5_D = 1001_B \times 101_B$  ter-se-ia uma adição de 5 parcelas de valor  $1001_B$  o que, como se pode comprovar, dá o resultado  $101101_B = 45_D$ . Este método designa-se por *adições sucessivas*.

#### Multiplicação Formal

Tal como para a adição, a multiplicação binária também pode ser realizada segundo os princípios elementares usados para a base 10. A tabuada da multiplicação binária apresenta-se na Tabela 4.

Multiplicação Binária	
Factores	Produto
$0 \times 0$	0
$0 \times 1$	0
$1 \times 0$	0
$1 \times 1$	1

**Tabela 4** – Multiplicação binária

---

<sup>3</sup> Isto não impede que, por motivos de avaliação de conhecimentos, os exercícios não devam ter que ser resolvidos manualmente, utilizando a base proposta, sem qualquer conversão prévia.

Seja a multiplicação  $9_D \times 5_D = 45_D$ . Convertendo os factores a binário pode ser realizada assim:

$$\begin{array}{r}
 1\ 0\ 0\ 1_B \leftarrow \text{multiplicando} \\
 \times 1\ 0\ 1_B \leftarrow \text{multiplicador} \\
 \hline
 1\ 0\ 0\ 1 \leftarrow a \\
 0\ 0\ 0\ 0 \downarrow \leftarrow b \\
 \hline
 0\ 1\ 0\ 0\ 1 \\
 1\ 0\ 0\ 1 \downarrow \downarrow \leftarrow c \\
 \hline
 1\ 0\ 1\ 1\ 0\ 1_B
 \end{array}$$

### Algoritmo de Multiplicação

A adição  $11001111_B + 11001111_B = 110011110_B$  exemplificada na secção anterior consiste em duas parcelas iguais,  $a$  e  $b$ . Portanto, a soma  $S$  terá de ser o dobro de  $a$  (e de  $b$ ). Ora como se pode verificar,  $S$  apenas difere de  $a$  e de  $b$  pelo facto de conter mais um zero à direita. Ou seja:

Para multiplicar um número binário por 2 basta acrescentar-lhe um 0 à direita (tal como na base 10 para multiplicar um número por  $10_D$ ).

Em terminologia de sistemas digitais esta operação é designada por *deslocamento à esquerda* e pode ser executada por unidades de memória de configuração especial chamadas *registos de deslocamento* ou *shift registers*. Estes deslocamentos são utilizados no algoritmo geral da multiplicação binária, descrito em seguida<sup>4</sup>. Antes de o apresentar, verifique-se porém a existência das seguintes propriedades, decorrentes da utilização da base 2:

- Se o dígito em consideração no multiplicador for 0, a parcela correspondente será 0 (isto também acontece na base 10). Por exemplo, a parcela  $b$ ;
- Se o dígito em consideração no multiplicador for 1, a linha gerada será igual ao multiplicando;
- O valor resultante será deslocado à esquerda de uma posição por cada parcela calculada. Este deslocamento é nulo para a primeira parcela. Por exemplo a linha  $a$  – a primeira – tem um deslocamento nulo e a linha  $c$  – a terceira – tem dois deslocamentos à esquerda.

### Algoritmo

<sup>4</sup> Elas equivalem aos deslocamentos que também se realizam numa multiplicação decimal.

1.  $n = 0$ ;
2.  $\text{produto} = 0$ ;
3. Tomar o dígito  $d_n$  (de ordem  $n$ ) do multiplicador;
4. Se for 1 {
  - a. Calcular  $\text{produto} = \text{produto} + \text{multiplicando}$ ; }
5.  $n = n + 1$ ;
6. Se  $n < n^\circ$  de dígitos do multiplicador {
  - a. Deslocar o multiplicando à esquerda 1 posição
  - b. Voltar ao passo 3 }
7. FIM

Segundo este algoritmo, a multiplicação  $9_D \times 5_D = 1001_B \times 101_B$  resulta assim:

1.  $n = 0$
2.  $\text{produto} = 0$ ;

Condições Iniciais

3.  $d_0 = 1$
4.  $\text{produto} = 0 + 1001 = 1001$
5.  $n = 0 + 1 = 1 (< 3)$
6.  $\text{multiplicando} = 10010$

1ª Iteração

3.  $d_1 = 0$
4. ...
5.  $n = 1 + 1 = 2 (< 3)$
6.  $\text{produto} = 100100$

2ª Iteração

3.  $d_2 = 1$
4.  $\text{produto} = 100100 + 1001 = 101101$
5.  $n = 2 + 1 = 3 (= 3)$
6. ...
7. FIM

3ª Iteração e Fim



E o resultado, conforme esperado, é  $9_D \times 5_D = 1001_B \times 101_B = 101101_B = 45_D$ . Esta é a ideia base do algoritmo usado pelos microprocessadores para multiplicar dois números binários<sup>5</sup>.

### 3.3. Subtracção

A subtracção em binário, hexadecimal ou octal, faz-se exactamente como no sistema decimal. Porém, tal como para a adição, não é prático realizá-la em hexadecimal ou octal, pelo que geralmente se converte a decimal ou binário, faz-se o cálculo e reconverte-se à base original.

Tal como acontece no sistema decimal, na subtracção em binário torna-se necessário, em certas condições, pedir valor emprestado (*borrow*) ao dígito mais significativo seguinte. Por exemplo quando se calcula  $145_D - 57_D$  faz-se, mentalmente: "7 para 15, 8 e vai 1; 1 e 5, 6; 6 para 14, 8 e vai 1; 1 para 1, nada". O pedir emprestado ocorre quando, neste exemplo, se toma o 5 por 15 e o 4 por 14. A tabuada da subtracção binária apresenta-se na Tabela 5.

Subtracção Binária		
Parcelas	Empréstimo	Resultado
0 - 0	0	0
0 - 1	1	1
1 - 0	0	1
1 - 1	0	0

**Tabela 5** – Subtracção binária

Aplicando-a, veja-se como a subtracção acima referida,  $145_D - 57_D$ , se faz em binário:

$$\begin{array}{rcccccccc}
 & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & & & & \longleftarrow \text{borrow} \\
 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1_B \longleftarrow \text{subtraendo (d)} \\
 - & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1_B \longleftarrow \text{subtractor (s)} \\
 \hline
 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0_B \longleftarrow \text{resultado (r)} \\
 & d_7 & d_6 & d_5 & d_4 & d_3 & d_2 & d_1 & d_0 \longleftarrow \text{ordem dos dígitos}
 \end{array}$$

Neste exemplo, a subtracção dos três dígitos menos significativos  $d_0 \dots d_2$ , não levanta qualquer

problema. Porém, em  $d_3$  acontece que se tem no subtraendo 0 e no subtrator, 1: como  $0 < 1$ , pede-se 1 emprestado a  $d_4$ , o que significa que se está a fazer  $10_B - 1_B$  em vez de  $0_B - 1_B$ . Este procedimento tem como consequência a propagação do *borrow* aos dígitos seguintes.

No exemplo anterior o subtraendo,  $d$ , tinha um valor superior ao do subtrator,  $s$  e ambos se assumiram como positivos. Veja-se o que acontece quando  $d = s$  e  $d < s$ , tomando os exemplo  $15_D - 15_D$  e  $15_D - 17_D$ , respectivamente:

$$\begin{array}{r} 1 \ 1 \ 1 \ 1_B \\ - \ 1 \ 1 \ 1 \ 1_B \\ \hline 0 \ 0 \ 0 \ 0_B \end{array} \qquad \begin{array}{r} 0 \ 0 \ 1 \ 1 \ 1 \ 1_B \\ 0 \ 1 \ 0 \ 0 \ 0 \ 1_B \\ \hline \dots \ 1 \ 1 \ 1 \ 1 \ 1 \ 0_B \end{array}$$

Quando  $d = s$  o resultado é 0, como era de esperar. Porém, quando  $d < s$  acontece que o resultado cresce indefinidamente "para a esquerda", propagando dígitos 1 uns após outros. Sobre este facto é preciso compreender que:

- Na realidade, e como se tem vindo a chamar a atenção, num sistema digital binário a capacidade de representação é limitada a  $n$  bits. Portanto, o que acontece na realidade é que, no resultado  $r$ , um bit de ordem  $n+1$ , isto é, à esquerda do MSb dos operandos, assumirá o valor 1 significando que o resultado é negativo;
- A subtracção de números binários é realizada, na prática, recorrendo a uma adição em que um dos números é negativo. As representações normalmente utilizadas para números negativos e as suas aplicações à subtracção são analisadas mais adiante, neste manual.

A subtracção de números binários é frequentemente utilizada pelos microprocessadores na execução dos seus programas dado que, para além da realização de operações aritméticas, lhes permite determinar qual de dois números é maior ou se são iguais (comparação) optando em seguida por uma sequência de instruções num caso e por outra no outro (os *IFs* e *loops* das linguagens de alto nível). Em sistemas digitais básicos as comparações recorrem a circuitos integrados chamados *comparadores* (*comparators*) que comparam dois números bit a bit sem realizarem, contudo, qualquer subtracção.

---

<sup>5</sup> Como se verá, as representações não são em binário puro pelo que este algoritmo não pode ser aplicado directamente. Contudo, a ideia base persiste.

A subtração em hexadecimal e octal realiza-se de forma semelhante à binária mas, mais uma vez, é vulgar realizar a conversão a binário, subtrair e reconverter à base original.

### 3.4. Divisão

Tal como para a multiplicação, descreve-se apenas a divisão na base 2. Apresentam-se também três métodos, correspondentes aos descritos para a multiplicação.

#### Subtrações Sucessivas

Evidentemente que é possível dividir executando subtrações sucessivas do divisor ao dividendo até que se obtenha um resultado negativo (em linguagem corrente isto significa que "já não há", isto é, que o dividendo já é inferior ao divisor). Portanto, neste método o quociente é o número de subtrações realizadas menos 1, e o resto é o subtraendo da penúltima subtração.

#### Divisão Formal

Sabendo multiplicar e subtrair pode realizar-se a divisão de números binários exactamente como no sistema decimal. Seja a divisão  $45_D \div 5_D = 101101_B \div 101_B$ :

$$\begin{array}{r}
 101101 \\
 - 101 \phantom{000} \\
 \hline
 0001 \phantom{0} \\
 - 0000 \phantom{0} \\
 \hline
 00010 \\
 - 0000 \phantom{0} \\
 \hline
 00101 \\
 - 101 \phantom{0} \\
 \hline
 0000
 \end{array}
 \quad
 \begin{array}{r}
 101 \\
 \hline
 1001
 \end{array}$$

$$45_D \div 5_D = 9_D$$

#### Algoritmo de Divisão

Simetricamente ao que acontece com a multiplicação, a divisão de um número binário por 2 pode realizar-se deslocando uma posição à direita os *bits* desse número. Note-se que se o LSb do número for 1, esta operação implica uma perda de precisão porque corresponde à perda de casas decimais, a não ser que haja algum meio de localizar o ponto, caso em que o referido dígito poderá ser mantido. Ver-se-á mais adiante como lidar com números que têm parte fraccionária.

O deslocamento à direita é utilizado no algoritmo da divisão binária, base da execução desta operação em microprocessadores. A operação chave consiste em comparar o divisor com o dividendo, para se determinar se ele é maior ou menor. Neste ponto reside a grande diferença e a simplificação que decorre do uso da base 2 relativamente à base 10: enquanto para esta há a necessidade de determinar "quantos há", isto é, se o novo dígito para o divisor será 0, 1, 2 ... ou 9, em binário apenas pode "haver" ou "não haver", o que se consegue muito facilmente atendendo a que:

- Se o dividendo ou o resto actual  $\geq$  divisor, então "há" e o novo dígito do quociente será 1 (subtrai-se então o divisor ao dividendo ou ao resto actual, "alinhando-o" pelo seu MSb);
- Se dividendo  $<$  divisor, então "não há" e o novo dígito do quociente será 0 (neste caso o resto será igual ao dividendo ou ao resto anterior porque o produto de 0 pelo divisor será 0 e nada haverá a subtrair).

Estes testes são realizados recorrendo a uma subtracção e analisando se o resultado é positivo ou negativo. Tal como na divisão manual, essa subtracção é realizada alinhando o dividendo ou o resto actual e o divisor, pela esquerda: por exemplo, em  $845_D \div 4_D$  testa-se o 4 do divisor relativamente ao 8 do dividendo, o que corresponde à subtracção  $845_D - 400_D$  (equivalentemente, o dígito 4 multiplicou-se por  $10^2$ ). Por isso o algoritmo da divisão inicia-se multiplicando por 2 o divisor tantas vezes quantas as necessárias para garantir este alinhamento. Como todos os números binários começam por 1 (dado que além do 1 apenas existe o dígito 0 e este não tem valor à esquerda), para garantir isto basta que inicialmente se considere um divisor com o mesmo número de *bits* do dividendo, o que se consegue acrescentando-lhe tantos 0's à direita quantos os necessários. Este "factor de escala" inicialmente aplicado é depois consumido ao longo do algoritmo através de deslocamentos à direita que, em cada teste, garantem o correcto alinhamento de resto e divisor (passo 7 do algoritmo). O algoritmo completo apresenta-se em seguida:

### Algoritmo

1.  $n = 1$ ;
2. resto = dividendo;
3. Deslocar divisor à esquerda (acrescentando-lhe 0's à direita) tantas vezes quantas as

- necessárias para atingir o número de *bits* do dividendo;
4. Calcular resto = resto-divisor;
  5. Se  $\text{resto} \geq 0$  {
    - a. Acrescentar à direita do actual quociente, um 1 ( $d_0 = 1$ ) }
  6. Senão {
    - a. Acrescentar à direita do actual quociente, um 0 ( $d_0 = 0$ )
    - b. Restaurar o resto (isto é, anular a operação 4:  $\text{resto}_n = \text{resto}_{n-1}$ ) }
  7.  $n = n+1$ ;
  8. Se  $n \leq (n^\circ \text{ inicial de dígitos do divisor} + 1)$  {
    - a. Deslocar o divisor à direita 1 bit
    - b. Voltar ao passo 4 }
  9. FIM

Segundo este algoritmo, a divisão  $45_D \div 5_D = 101101_B \div 101_B = 1001_B$  e resto 0, resulta assim:

- |  |  |
|--|--|
| <ol style="list-style-type: none"> <li>1. <math>n = 1</math>;</li> <li>2. resto = 101101<sub>B</sub>;</li> <li>3. divisor = 101000<sub>B</sub>;</li> </ol>   | <div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;">Condições Iniciais</div> |
| <ol style="list-style-type: none"> <li>4. resto = 101101<sub>B</sub> - 101000<sub>B</sub> = 000101<sub>B</sub>;</li> <li>5. resto &gt; 0: quociente = 1;</li> <li>6. ...</li> <li>7. <math>n = 1 + 1 = 2</math>;</li> <li>8. <math>n &lt; 4</math>: divisor = 10100<sub>B</sub>;</li> </ol>  | <div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;">1ª Iteração</div>        |
| <ol style="list-style-type: none"> <li>4. resto = 000101<sub>B</sub> - 10100<sub>B</sub> = ...1110001<sub>B</sub>;</li> <li>5. resto &lt; 0:</li> <li>6. quociente = 10<sub>B</sub>, Resto = 000101<sub>B</sub>;</li> <li>7. <math>n = 2 + 1 = 3</math>;</li> <li>8. <math>n &lt; 4</math>: divisor = 1010<sub>B</sub>;</li> </ol> | <div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;">2ª Iteração</div>        |
| <ol style="list-style-type: none"> <li>4. resto = 000101<sub>B</sub> - 1010<sub>B</sub> = ...111011<sub>B</sub>;</li> <li>5. resto &lt; 0:</li> <li>6. quociente = 100<sub>B</sub>, resto = 000101<sub>B</sub>;</li> </ol>   | <div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;">3ª Iteração</div>        |

7.  $n = 3+1 = 4$ ;
8.  $n = 4$ : divisor =  $101_B$ ;
4. resto =  $000101_B - 101_B = 000_B$ ;
5. resto = 0: quociente =  $1001_B$ ;
6. ...
7.  $n = 4+1 = 5$ ;
8.  $n > 4$ :
9. FIM.

4ª Iteração e Fim
-------------------

Portanto, o quociente é  $1001_B$  e o resto é  $0_B$ , ou seja,  $9_D$  e  $0_D$  conforme pretendido.

### 3.5. Aritmética Binária e Portas Lógicas

Expostos os princípios que norteiam a execução de operações aritméticas com números binários, resta saber como podem os circuitos digitais realizar tais operações, uma vez que, conforme estudado em Sistemas Digitais, eles apenas efectuam operações lógicas.

A Álgebra de Boole define as operações lógicas básicas AND, OR e NOT que nos circuitos integrados digitais são implementadas essencialmente à custa de transístores e assumindo que, em lógica positiva, o valor lógico 1 (ou T) é representado pelo nível H e o valor lógico 0 (ou F) pelo nível L. Ora, por exemplo a operação lógica AND (e por conseguinte uma porta lógica AND) é definida como produzindo uma saída de valor 1 (ou seja, de nível H) somente quando ambas as entradas forem de valor 1. Esta definição representa-se na Tabela 6, à direita da tabela da adição binária.

Adição Binária			Porta Lógica AND	
Parcelas	Transporte	Soma	Entradas	Saída
0 + 0	0	0	L, L	L
0 + 1	0	1	L, H	L
1 + 0	0	1	H, L	L
1 + 1	1	0	H, H	H

**Tabela 6** – Adição binária de 2 bits e operação de uma porta lógica AND

Como se pode observar:

- A saída Transporte corresponde linha por linha à saída produzida por uma porta lógica AND. Por outras palavras, uma porta AND é suficiente para detectar se da adição de 2 *bits* resulta ou não um transporte;
- Quanto ao *bit* Soma esta correspondência directa já não existe. Porém, em Sistemas Digitais estudámos as ferramentas e procedimentos necessários para deduzir, a partir de tabelas deste tipo, as portas lógicas necessárias para produzir um *bit* Soma ou outra *função lógica* qualquer.

Em suma, quando se trabalha numa base binária torna-se possível implementar operações aritméticas básicas a partir de funções lógicas. Este é o segredo que confere à base 2 e aos sistemas digitais todo o seu poder.

## 4. Códigos Bipolares

A necessidade de representar quantidades positivas e negativas, aliada ao facto de um mesmo circuito (*somador*) permitir realizar operações de adição e subacção (por exemplo,  $9_D - 6_D = 9_D + (-6_D) = 3_D$ ), levou à criação de códigos genericamente designados por *bipolares*, que permitem a representação de números positivos e negativos. Nesta secção apresentam-se 4 destes códigos.

Por razões que se tornarão claras mais adiante, é conveniente optar-se, nesta ocasião, por representações com um número fixo de *bits*, o que corresponde exactamente ao que se passa na realidade: se um sistema digital pode tratar 8 *bits* em paralelo, então qualquer número será representado em 8 *bits* acrescentando-se-lhe, para isso 0's à esquerda. Por exemplo, a representação do número  $1_D$  com 8 *bits* será 00000001<sub>B</sub>.

### 4.1. Sinal e Valor Absoluto

Naturalmente que a primeira hipótese que ocorre para a representação de números binários negativos consiste em reservar um *bit* para a indicação do seu sinal. Na *representação em sinal e valor absoluto* é exactamente isto que acontece: reserva-se o MSb para o sinal, atribuindo-se-lhe o valor 0 para números positivos e o valor 1 para números negativos; os restantes *bits* utilizam-se para representar o valor absoluto do número. Assim, assumindo uma representação de 8 *bits*, tem-se, por exemplo:

$$-127_D = 1111\ 1111_B$$

$$-126_D = 1111\ 1110_B$$

$$-125_D = 1111\ 1101_B$$

...

$$-1_D = 1000\ 0001_B$$

$$-0_D = 1000\ 0000_B$$

$$+0_D = 0000\ 0000_B$$

$$+1_D = 0000\ 0001_B$$

...

$$+125_D = 0111\ 1101_B$$

$$+126_D = 0111\ 1110_B$$

$$+127_D = 0111\ 1111_B$$

Este código tem as seguintes características:

- Os números positivos começam todos por 0 e os negativos por 1;
- O zero tem duas representações:  $+0_D = 0000\ 0000_B$  e  $-0_D = 1000\ 0000_B$ ;
- Dados  $n$  bits, permite representar igual quantidade de números inteiros positivos e negativos, compreendidos na gama  $[-(2^{n-1}-1), +(2^{n-1}-1)]$ .

Contudo este sistema é inadequado para a realização de operações aritméticas na base 2 de forma algorítmica, visto implicar uma série de testes e de decisões: de facto, para adicionar dois números o sistema digital teria de examinar os sinais de cada um deles e em seguida optar por somar os bits à sua direita, se fossem ambos do mesmo sinal, ou subtraí-los, se fossem de sinais contrários. Esta sequência de operações é composta pela avaliação de uma condição para teste do sinal; pela selecção de  $n-1$  bits para extracção do valor absoluto; e por duas operações distintas – soma e subtracção – o que implica uma elevada complexidade para o circuito destinado a realizá-la.

## 4.2. Complementos de 1

Neste código, os números positivos são representados como no código de sinal e valor absoluto. Um número negativo obtém-se a partir do positivo correspondente trocando os 0's por 1's e vice-



versa. Assim:

$$-127_D = 1000\ 0000_B$$

$$-126_D = 1000\ 0001_B$$

$$-125_D = 1000\ 0010_B$$

...

$$-1_D = 1111\ 1110_B$$

$$-0_D = 1111\ 1111_B$$

$$+0_D = 0000\ 0000_B$$

$$+1_D = 0000\ 0001_B$$

...

$$+125_D = 0111\ 1101_B$$

$$+126_D = 0111\ 1110_B$$

$$+127_D = 0111\ 1111_B$$

Portanto, este código tem as seguintes características:

- Os números positivos começam todos por 0 e os negativos por 1;
- O zero tem duas representações:  $+0_D = 0000\ 0000_B$  e  $-0_D = 1111\ 1111_B$ .
- Dados  $n$  bits, permite representar igual quantidade de números inteiros positivos e negativos, compreendidos na gama  $[-(2^{n-1}-1), +(2^{n-1}-1)]$ ;
- O complementos de 1 de um número negativo regenera o positivo correspondente.

Para realizar uma subtracção recorrendo ao complemento de 1, procede-se assim:

1. Transforma-se a subtracção em adição algébrica, afectando com sinal menos o subtrator;
2. O subtraendo original e o subtrator obtido em 1 representam-se em complementos de 1;
3. Adicionam-se os operandos;
4. Adiciona-se o transporte obtido em 3), se o houver.

Note-se que o resultado obtido em 4 fica em complementos de 1. Isto significa que, se se pretender saber o valor decimal correspondente com base na Equação 3, é necessário analisar se esse resultado expressa um número positivo ou negativo e, no segundo caso, convertê-lo à notação

binária pura antes de efectuar a conversão à base 10. Os exemplos seguintes esclarecem este procedimento.

### Exemplo 1:

$$\begin{aligned}
 16_D - 15_D &= ? \\
 &= 00010000_B - 00001111_B \\
 &= 00010000_B + (-00001111_B) \\
 &= 00010000_{C1} + 11110000_{C1} \\
 &= 100000000_{C1}
 \end{aligned}$$

De acordo com o passo 4, há agora que adicionar o transporte ao resultado da adição:

$$\begin{array}{r}
 \begin{array}{cccccccc}
 1 & 1 & 1 & 1 & & & & \\
 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
 + & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
 \hline
 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 \hline
 \begin{array}{c} \text{└───────────▶} \end{array} & + & 1 \\
 \hline
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \longrightarrow \text{Este é o valor } +1_D
 \end{array}$$

Conforme referido, o resultado encontra-se em complementos de 1. Qual o número decimal correspondente? Olhando para a tabela dada no início deste manual, vê-se de imediato que se trata do número  $+1_D$ . Contudo, o método geral de procedimento consiste em analisar o *bit* de mais alta ordem e, atendendo a que em complementos de 1 todos os números negativos começam por 1 e os positivos por 0, verificar qual o seu sinal. Neste exemplo o *bit* de sinal, 0, indica que o número é positivo, o que significa que a sua representação em complementos de 1 é igual à de binário natural. Assim, o número obtido é  $0000\ 0001_{C1} = +(000\ 0001_B) = +1_D$ .

### Exemplo 2:

$$\begin{aligned}
 15_D - 16_D &= ? \\
 &= 00001111_B - 00010000_B \\
 &= 00001111_B + (-00010000_B) \\
 &= 00001111_{C1} + 11101111_{C1} = \\
 &= 11111110_{C1}
 \end{aligned}$$

Como não houve transporte, nada há a adicionar ao resultado da adição, de acordo com a tabela dada no início deste manual, vê-se de imediato que o resultado é  $-1_D$ . De facto, como o *bit* de mais alta ordem é 1, está-se em presença de um número negativo representado em complementos de 1. Para passar ao positivo correspondente, há portanto que o complementar. Donde:  $1111\ 1110_{CI} = -(000\ 0001_B) = -1_D$ .

### Exemplo 3:

$$\begin{aligned} 15_D - 15_D &= ? \\ &= 00001111_B - 00001111_B \\ &= 00001111_B + (-00001111_B) \\ &= 00001111_{CI} + 11110000_{CI} \\ &= 11111111_{CI} \end{aligned}$$

Como se pode verificar, o resultado,  $11111111_{CI}$  é exactamente uma das representações do 0 em complementos de 1. Mas, como em binário puro existe apenas uma representação para o 0, a conversão a binário puro segue ainda a regra geral: portanto, analisa-se o *bit* de sinal. Como ele é 1, significa que o número é negativo pelo que é preciso complementá-lo obtendo-se  $1111\ 1111_{CI} = -(000\ 0000_B) = 0_D$ .

Em termos de circuitos digitais, a complementação dos *bits* que compõem um dado número é simples de realizar: basta negá-los todos, o que se consegue com portas lógicas NOT. Assim, a subtracção em complementos de 1 é mais linear do que em sinal e valor absoluto. Contudo este sistema possui ainda alguns problemas no que respeita à necessidade de adicionar o transporte decorrente da primeira adição (note-se que não é necessária nenhuma condição porque ele pode ser sempre adicionado: se for 0 não afectará o resultado) e de reconhecer duas formas diferentes do número 0<sub>D</sub>.

## 4.3. Complementos de 2

Neste código, os números positivos são representados como no código de sinal e valor absoluto e de complementos de 1. A representação de um número negativo obtém-se a partir da do positivo trocando os 0's por 1's (tal como nos complementos de 1) e adicionando 1 ao resultado. Contudo, a

transformação para complementos de 2 pode fazer-se na prática e de uma forma mais rápida, através do seguinte algoritmo:

Começando pelo dígito menos significativo, copiam-se todos os dígitos até se encontrar o primeiro 1, que também se copia. A partir daí, trocam-se todos os 0's por 1's e vice-versa.

$$-128_D = 1000\ 0000_B$$

$$-127_D = 1000\ 0001_B$$

$$-126_D = 1000\ 0010_B$$

$$-125_D = 1000\ 0011_B$$

...

$$-1_D = 1111\ 1111_B$$

$$-0_D = 0000\ 0000_B$$

$$+1_D = 0000\ 0001_B$$

...

$$+125_D = 0111\ 1101_B$$

$$+126_D = 0111\ 1110_B$$

$$+127_D = 0111\ 1111_B$$

Portanto, este código tem as seguintes características:

- Os números positivos começam todos por 0 e os negativos por 1;
- O zero tem uma representação única,  $0_D = 0000\ 0000_B$ ;
- Dados  $n\ bits$ , permite representar mais um número inteiro negativo que positivo; estes números estão compreendidos na gama  $[-2^{n-1}, +(2^{n-1}-1)]$ ;
- O complementos de 2 de um número negativo regenera o positivo correspondente.

Para realizar uma subtracção em complementos de 2, procede-se assim:

1. Transforma-se a subtracção em adição algébrica, afectando com sinal menos o subtrator;
2. O subtraendo original e o subtrator obtido em 1 representam-se em complementos de 2;
3. Adicionam-se os operandos;
4. Despreza-se o transporte, se existir. (*Ver secção de Overflow*)

### Exemplo 1:

$$\begin{array}{cccccccc}
 & 1 & 1 & 1 & 1 & & & \\
 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
 + & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
 \hline
 \textcircled{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \longrightarrow \text{Este é o valor } +1_D \\
 \downarrow & & & & & & & & \\
 & & & & & & & & \text{O transporte despreza-se}
 \end{array}$$

### Exemplo 2:

$$\begin{aligned} 15_D - 15_D &= ? \\ &= 00001111_B - 00001111_B \\ &= 00001111_B + (-00001111_B) \\ &= 00001111_{C2} + 11110001_{C2} \\ &= 00000000_{C2} \end{aligned}$$

$$\begin{array}{r}
 \begin{array}{cccccccc}
 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
 + & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
 \hline
 \textcircled{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
 \end{array}
 \end{array}$$

Esta é a representação

O transporte despreza-se

Como o *bit* de mais alta ordem é 0, o número é positivo pelo que,  $0000\ 0000_{C2} = +(000\ 0000_B) = 0_D$ .

### Exemplo 3:

$$\begin{aligned}
 15_D - 16_D &= ? \\
 &= 00001111_B - 00010000_B \\
 &= 00001111_B + (-00010000_B) \\
 &= 00001111_{C2} + 11110000_{C2} \\
 &= 11111111_{C2}
 \end{aligned}$$

Atendendo à tabela dada no início deste manual, vê-se de imediato que o resultado é  $-1_D$ . De facto, seguindo o procedimento geral, como o *bit* de mais alta ordem é 1, o número é negativo, pelo que é necessário aplicar a regra de conversão *complementos de 2*  $\rightarrow$  *binário* para obter o binário puro correspondente. Assim tem-se:  $1111\ 1111_{C2} = -(000\ 0001_B) = -1_D$ .

O código de complementos de 2 é o mais utilizado por ter uma representação única para o zero e permitir a obtenção do resultado directamente, sem necessidade de operações adicionais. Note-se que a geração de complementos de 2 de um número também é simples por não incluir quaisquer condicionais, isto é, basta trocar os 0's por 1's e adicionar 1 ao resultado.

## 4.4. Código Binário Deslocado ou Excesso $2^{n-1}$

Estamos perante uma família de códigos dado que o valor  $2^{n-1}$  depende de  $n$ , que representa o número de *bits* usados na codificação. Por exemplo, para  $n = 8$  tem-se  $2^{n-1} = 128_D$ . Nestas condições está-se em presença de um *código de deslocamento (bias)* 128 em que a representação de qualquer número se faz adicionando  $128_D$  a esse número e exprimindo o resultado em binário natural:

$$-128_D = 0000\ 0000_B$$

$$-127_D = 0000\ 0001_B$$

$$-126_D = 0000\ 0010_B$$

$$-125_D = 0000\ 0011_B$$

...

$$-1_D = 0111\ 1111_B$$

$$0_D = 1000\ 0000_B$$

$$+1_D = 1000\ 0001_B$$

...

$$+125_D = 0111\ 1101_B$$

$$+126_D = 0111\ 1110_B$$

$$+127_D = 0111\ 1111_B$$

A representação, à custa de  $n$  bits, de um número positivo ou negativo, é obtida somando  $2^{n-1}$  a esse número e exprimindo o resultado em código binário natural.

Portanto, este código tem as seguintes características:

- Os números positivos começam todos por 1 e os negativos por 0;
- O zero tem uma representação única, que depende do excesso;
- Dados  $n$  bits, permite representar mais um número inteiro negativo que positivo; estes números estão compreendidos na gama  $[-2^{n-1}, +(2^{n-1}-1)]$ .

Os códigos deste tipo são utilizados essencialmente para a representação de números em vírgula flutuante, uma abordagem completamente diferente de todas as anteriores, e descrita mais adiante.

## 4.5. Overflow

A realização de operações aritméticas em qualquer dos códigos atrás apresentados necessita sempre de certos cuidados relativamente à validade do resultado. Com efeito, dado que nos sistemas digitais o número de *bits* disponíveis para uma representação é limitado, pode acontecer que o resultado de uma operação aritmética não seja representável nesse número de *bits* por ser demasiado grande. Neste caso diz-se que ocorre uma *ultrapassagem de capacidade* ou *overflow*. É o que acontece, por exemplo, quando as calculadoras produzem erros na divisão por 0 ou no cálculo

Dado que o complemento de 2 é um dos códigos mais usuais para a representação de números positivos e negativos, analisa-se apenas este código. Além disso, como as operações de multiplicação e divisão são, devido à sua complexidade, executadas por microprocessadores ou circuitos especialmente desenvolvidos para esse efeito, trata-se apenas da adição algébrica.

$C_{in} \longrightarrow$ 

0
1 0 0 0 0 0 0 0
+ 1 1 1 1 1 1 1 1
1 0 1 1 1 1 1 1

  
 $C_{out} \longleftarrow$ 

0
1 0 0 0 0 0 0 0
+ 1 1 1 1 1 1 1 1
1 0 1 1 1 1 1 1

  
 $\longrightarrow$  *Bit de sinal*

$C_{in}$   $\longrightarrow$ 

1	1	1	1	1	1	1
0	1	1	1	1	1	1

  
 $+ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1$ 


---

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

  
 $C_{out}$   $\longleftarrow$   $\longrightarrow$  *Bit de sinal*

Em complementos de 2 uma adição origina overflow quando os sinais dos operandos são iguais e o resultado apresenta sinal contrário.



Outra observação que se pode fazer em relação aos dois exemplos acima, é que em ambos o *carry* gerado à saída do *bit* de sinal ( $C_{out}$ ) é de valor contrário ao *carry* que nele deu entrada ( $C_{in}$ ). De facto, no primeiro caso  $C_{in} = 0$  e  $C_{out} = 1$ ; no segundo,  $C_{in} = 1$  e  $C_{out} = 0$ . Estas constatações são de facto gerais, pelo que uma outra regra, talvez mais conhecida, enuncia-se assim:

Em complementos de 2 uma adição origina overflow quando o *carry* que dá entrada no bit de sinal for diferente do que *carry* que dele sai.

## 5. Formatos dos números

Designa-se por formato de um número a forma como esse número é apresentado no computador à custa de dígitos binários. A grande maioria dos computadores apenas pode executar operações sobre números que lhe sejam fornecidos num determinado formato. Este tem em consideração o comprimento, expresso através de um determinado número de bits, e a convenção utilizada para codificar ou representar os números inteiros, fraccionários ou mistos. Para a representação dos diversos tipos de números foram utilizados, ao longo dos tempos, essencialmente dois tipos de formato: formato fixo ou em vírgula fixa e formato flutuante ou em vírgula flutuante.

### 5.1. Vírgula Fixa

O formato em vírgula fixa considera que a vírgula ocupa uma posição (imaginária) fixa, deixando ao programador o cuidado de atribuir factores de escala aos números representados e de fazer as necessárias alterações a esses factores de escala à medida que as operações forem evoluindo. Relativamente a este formato existiram ainda duas convenções: vírgula fixa inteira e vírgula fixa fraccionária. No primeiro caso, admitia-se que a vírgula se encontrava à direita dos números representados, caso em que todos os números eram representados como inteiros. No segundo caso considerava-se que a vírgula se encontrava à esquerda dos números representados, significando que todos os números eram expressos como números fraccionários. Assim, por exemplo, o número 11010,001 poderia ser expresso por

Vírgula Fixa Inteira

1	1	0	1	0	0	0	1	,
---	---	---	---	---	---	---	---	---

Vírgula Fixa Fraccionária

,

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

A vírgula indicada seria apenas imaginária, deixando ao programador o cuidado de recordar que estes números estariam afectados dos factores de escala  $2^{-3}$ , e  $2^5$ , no 1º e 2º caso, respectivamente.

Como facilmente se depreende as operações com vírgula fixa levantavam vários problemas ao programador. Uma delas prendia-se com o facto de a adição de dois números poder dar lugar a transbordo ou ultrapassagem da capacidade. No exemplo seguinte, assumindo a adição de  $1011100,1_2$  com  $1001101,1_2$  e utilizando a representação de números no formato de vírgula fixa inteira, gerar-se-ia no resultado um número demasiado longo para poder ser representado à custa de 8 bits.

		1	0	1	1	1	0	0	1
		1	0	0	1	1	0	1	1
1	0	1	0	1	0	1	0	1	0

Nestas situações, era necessário testar o resultado para verificar se houve transbordo para além de se ter de ajustar o factor de escala do resultado caso o transbordo se verificasse. Ou seja, as duas parcelas e o resultado estão afectadas do factor de escala  $2^{-1}$ , e dado que se verifica um transbordo, o resultado teria que ser ajustado para  $10101010$  que corresponde a um factor de escala de  $2^0$ .

Outra dificuldade prende-se com o facto de, antes de se programar uma adição ou subtracção, ser necessário prever o deslocamento dos operandos para a direita ou esquerda, de tal forma que apresentem factores de escala iguais. Por exemplo, a soma de  $10011,110$  com  $1001,1010$  só poderá ser efectuada depois de os operandos se encontrarem alinhados sob a seguinte forma:

1	0	0	1	1	1	1	0
0	1	0	0	1	1	0	1

que corresponde à afectação de um factor de escal de  $2^{-3}$ .

Outra problema diz respeito à execução de multiplicações. Para fazer os cálculos com a máxima precisão, é necessário prever a ordem de grandeza dos resultados intermédios para os poder enquadrar (à custa de factores de escala), de forma a que tenham o máximo de algarismos significativos.

Os problemas apresentados são ultrapassados recorrendo a representações de vírgula flutuante.

## 5.2. Vírgula Flutuante e IEEE-754

Como se sabe, o processamento de dados em computador exige, em muitas aplicações, numerosos e apurados cálculos em que intervêm valores numéricos positivos e negativos das mais variadas ordens de grandeza. Além disso, devido à diversidade de processadores, sistemas operativos e linguagens de programação que com o tempo foram surgindo, impunha-se o estabelecimento de algum tipo de norma que estabelecesse métodos de representação numérica homogéneos.

A norma IEEE-754 (*Institute of Electrical and Electronic Engineering*), de 1985, é exactamente uma norma deste tipo: ela define a representação de números reais baseando-se num sistema designado por *vírgula flutuante* (*floating-point*). É hoje o modo de representação mais comum, usado nos PC's baseados em processadores Intel, nos Macintosh e em grande parte dos sistemas Unix.

Basicamente, o sistema de vírgula flutuante representa os números em notação científica, ou seja, recorrendo a uma *mantissa* e a um *expoente*. Por exemplo, na base 10 o número -123.456 seria representado por  $-1.23456 \times 10^2$  e na base 16 o número ABCD.EF14 seria  $A.BCDEF14 \times 16^3$ . Nestas representações, 1.23456 e A.BCDEF14 são designados por *mantissa* e o 2 e o 3 que potenciam as bases 10 e 16, respectivamente, são designados por *expoente*. Em suma, o sistema de vírgula flutuante representa números reais num formato do tipo

$$s \ m_n.m_{n-1}...m_0 \times B^e$$

com  $s = \text{signal}$

$m = \text{dígitos que compõem a mantissa}$

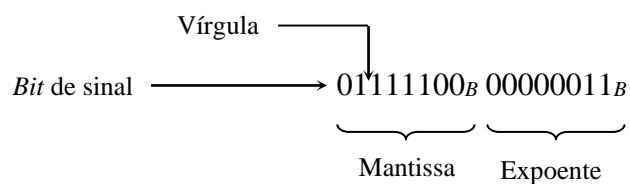
$B = \text{base}$

$e$  = expoente (que pode ser negativo)

Uma característica fundamental do sistema de vírgula flutuante é que a vírgula se situa, sempre, à direita do primeiro dígito significativo (ou seja, diferente de *zero*). Nestas condições o número diz-se *normalizado*. Por exemplo,  $12.34 \times 10^2$  não está normalizado;  $1.234 \times 10^3$  está normalizado. Na base 2 tudo se passa da mesma forma. Por exemplo, o número  $15.5_D = 1111.1_B$  será  $1.1111_B \times 2^{11_B}$ . Acerca destes exemplos, note-se o seguinte:

- A base não necessita de ser representada: de facto, desde que num dado sistema se saiba qual a base em que se está a trabalhar, é, evidentemente, escusado representá-la;
- A vírgula também não, uma vez que por definição ela se situa sempre à direita do primeiro dígito.

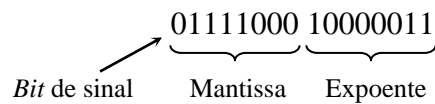
Portanto, supondo que se dispõe de dois *bytes*, o número  $1.1111_B \times 2^{11}$  poderia ser representado por algo do género:



Contudo, números menores que 1 exigem um expoente negativo. Que código usar para representar expoentes negativos? A norma IEEE-754 determina que seja um código do tipo *binário deslocado*, já anteriormente descrito. Seguindo o exemplo anterior, dado que se têm 8 *bits* para o expoente, este código poderá utilizar um deslocamento de  $2^{n-1} = 2^7 = 128$ . Assumindo este tipo de código a representação anterior fica:

01111100 10000011

Finalmente, quando se trabalha na base 2 os dígitos podem apenas assumir os valores 0 e 1. Portanto, se a vírgula se supõe sempre à direita do primeiro dígito significativo, então à esquerda da vírgula existe sempre um 1. Portanto, poderá ser suprimida a sua representação. O número reduz-se a:



A norma IEEE-754 especifica representações do tipo da anterior mas com algumas diferenças fundamentais.

- Define dois tipos de números: precisão simples (*single precision*) e precisão dupla (*double precision*):
  - A precisão simples ocupa 32 bits [0 a 31];
  - A precisão dupla ocupa 64 bits [0 a 63];
- O *bit* de sinal é o primeiro (1 *bit*, 0 para números positivos e 1 para números negativos), segue-se o expoente (8 ou 11 *bits*) e depois a mantissa (23 ou 52 *bits*). Portanto,  $1+8+23 = 32$  e  $1+11+52 = 64$  *bits*, consoante o tipo de precisão;
- O expoente é representado em excesso/deslocamento  $2^{n-1}-1$ , sendo  $n$  o número de dígitos reservados para o expoente.

A Tabela 7 resume estas características (os números entre [...] indicam a posição ocupada pelos diversos *bits*).

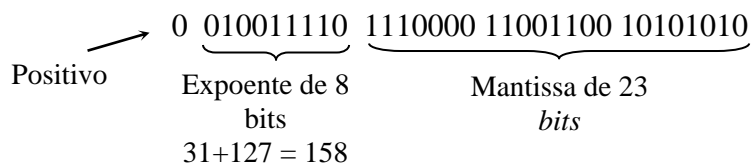
	Sinal	Expoente	Mantissa
<b>Pr. Simples</b>	1 [31]	8 [30-23]	23 [22-00]
<b>Pr. Dupla</b>	1 [63]	11 [62-52]	52 [51-00]

**Tabela 7** – Precisão Simples e Dupla de acordo com a norma IEEE-754

Este sistema permite a representação de números com ordens de grandeza  $\pm 10^{38}$  para precisão simples e  $\pm 10^{708}$  para precisão dupla. Note-se porém que alguma coisa se perde, e trata-se da precisão. Suponha-se o inteiro de 32 bits:

11110000 11001100 10101010 00001111

Como se trata de um inteiro, este número representa exactamente o valor pretendido, sem qualquer erro ou aproximação. Em precisão simples ele será:



Como se percebe, os últimos 4 *bits* do número original perderam-se e no pior caso – último byte =  $11111111_B$  – perder-se-iam 8 *bits*. A "reconstituição" do número original dará:

11110000 11001100 10101010 00000000

Note-se que o erro é pequeno: no exemplo considerado o valor original era equivalente a  $4.039.944.719_D$  e o reconstituído  $4.039.944.704_D$  que diferem entre si de apenas 15 unidades, ou seja, um erro relativo de 0.00000371%.

Na representação de números segundo a norma IEEE-754 não são utilizados os expoentes mínimos, 0 (correspondente a -127 ou -1023) e máximos 255 ou 1023 (correspondente a 128 e 1024, respectivamente). Estes são usados para representação de valores especiais.

De facto, como se partiu do princípio que o primeiro dígito significativo é sempre 1, o *zero* não poderia deixar de ter um tratamento particular. A indicação de um expoente -127 ou -1023 (bits de expoente todos preenchidos com 0) indica, segundo a norma, que a mantissa representa números não normalizados, os quais se assumem estar no formato 0,xxxx.xxx. Nesta situação caso os xxx sejam todos preenchidos a 0 estamos na presença da representação do valor  $0,00...00 \times (2^{-127} \text{ ou } 2^{-1023})$ , ou seja o valor 0. Caso algum desses x's seja diferente de 0, estamos na presença de números não inicialmente representáveis, extremamente pequenos, por exemplo,  $0,0000...00001_B \times (2^{-127} \text{ ou } 2^{-1023}) = (2^{-23} \text{ ou } 2^{-52}) \times (2^{-127} \text{ ou } 2^{-1023}) = 1,0_B \times (2^{-150} \text{ ou } 2^{-1075})$ , o que vai permitir a representação de valores pertencentes ao intervalo  $[2^{-150}, 0,1111..111_B \times 2^{-127}]$  ou  $[2^{-1075}, 0,1111..111_B \times 2^{-1023}]$ .

Os valores de expoente máximos (expoente todo preenchido a 1) são usados para representar as situações de infinito (com mantissa toda a 0) e *Not-a-Number* (mantissa com qualquer padrão de bits diferente de 0).

<b>Normalizado</b>	$\pm$	$0 < \text{Exp} < \text{Max}$	Qualquer padrão de bits
<b>Não normalizado</b>	$\pm$	0	Qualquer padrão de bits diferente de zero
<b>Zero</b>	$\pm$	0	0
<b>Infinito</b>	$\pm$	111...1	0
<b>Not-a-Number</b>	$\pm$	111...1	Qualquer padrão de bits diferente de zero

Como ilustração deste tema, refira-se que por exemplo as linguagens FORTRAN, C e MASM (um assembler) da *Microsoft* usam precisão simples e dupla nas seguintes declarações de variáveis:

- FORTRAN: REAL (simples); DOUBLE PRECISION (dupla);
- C: *float* (simples); *double* (dupla);
- MASM: Directiva DD (simples); directiva DQ (dupla).

Seguem-se alguns exemplos de números decimais convertidos em vírgula flutuante, base 2, precisão simples:

	<u>SEEE</u>	<u>EEEE</u>	<u>EMMM</u>	<u>....</u>	<u>MMMM</u>	
+2 = $1 \times 2^1$	= 0100	0000	0000	....	0000	= 40 00 00 00 <sub>H</sub>
-2 = $-1 \times 2^1$	= 1100	0000	0000	....	0000	= C0 00 00 00 <sub>H</sub>
+1 = $1 \times 2^0$	= 0011	1111	1000	....	0000	= 3F 80 00 00 <sub>H</sub>
2.5 = $1.25 \times 2^1$	= 0100	0000	0010	....	0000	= 40 20 00 00 <sub>H</sub>
0.1 = $1.6 \times 2^{-4}$	= 0011	1101	1100	....	1101	= 3D CC CC CD <sub>H</sub>
(neste caso o último byte tem o valor D por arredondamento do dígito C que se seguiria)						

## 6. Outros códigos - Binário Codificado em Decimal

O recurso aos sistemas de numeração octal e hexadecimal, como processos de exprimir de forma condensada a informação binária, pode considerar-se universalmente adoptado pelos fabricantes de computadores digitais. Porém, em muitos sistemas digitais binários de pequena dimensão, prevalece a preferência por uma conversão directa da informação decimal para a informação binária e vice-versa. Para que essa conversão se possa efectuar sem dificuldade recorre-se à codificação dos dígitos decimais à custa de dígitos binários. Deste tipo de códigos destacam-se os seguintes: Código 8-4-2-1 ou código BCD, Código 2-4-2-1 e Código excesso 3. Estes códigos são frequentemente

utilizados em aplicações em que o *output* é um *display* digital. Por exemplo, contadores de frequências, voltímetros digitais, calculadoras, entre outros.

A designação binário codificado em decimal ou BCD (*binary coded decimal*) aplica-se a um grupo de códigos que têm todos a seguinte característica:

A codificação faz-se para cada dígito decimal e não para o número no seu todo.

Por exemplo: enquanto para converter a binário puro o número  $254_D$  se atende ao seu valor global, para o converter a um código do tipo BCD atende-se à codificação binária de cada um dos dígitos que o compõem: 2, 5 e 4; a representação em BCD é obtida justapondo o código de cada dígito.

Como existem 10 dígitos decimais, também são necessárias 10 representações em BCD. E, como para obter 10 códigos diferentes são necessários pelo menos 4 *bits* (uma vez que  $2^3 = 8$  e  $2^4 = 16$  é a primeira potência de 2 superior a  $10_D$ ), em qualquer código BCD existem sempre pelo menos  $16 - 10 = 6$  combinações que não são usadas.

Alguns códigos BCD são ponderados (mas não todos), isto é, o valor decimal de cada código obtém-se multiplicando o dígito binário correspondente por um coeficiente de ponderação que pode ou não ser uma potência de 2.

### 6.1. BCD Natural ou 8421

No BCD natural cada dígito é codificado exactamente como em binário, começando por  $0_D = 0000_B$  e terminando em  $9_D = 1001_B$ . As combinações de  $1010_B$  a  $1111_B$  não são utilizadas. Portanto em BCD natural o número  $254_D$  é representado por  $0010\ 1001\ 0100_B$ . Note-se que em BCD natural um *byte* apenas pode representar valores até  $99_D = 1001\ 1001_B$ , contrariamente ao binário puro cujo limite é de  $255_D = 1111\ 1111_B$  por *byte*. Este exemplo demonstra o desperdício de *bits* para a representação da informação.

A diferença entre o sistema binário e o código BCD só se faz sentir a partir do número decimal 9, uma vez que no código BCD os números decimais não são tratados no seu conjunto, mas antes dígito a dígito.



Em muitos sistemas digitais que recorrem ao código BCD, utiliza-se um bit adicional à esquerda dos conjuntos de 4 bits que exprimem nesse código os valores decimais-**extensão da gama**. Assim é possível representar o dobro dos valores que seriam representáveis na sua ausência. Por exemplo, com 12 bits poderemos representar valores BCD entre 0 e 999, com mais um bit poderemos representar valores entre 0 e 1999.

Este código é ponderado pois atribui aos diferentes dígitos binários os pesos 8, 4, 2 e 1, consoante as posições dos dígitos da esquerda para a direita. Por este facto, também é conhecido por código 8421.

Tal como na adição em base 10, a adição em BCD é efectuada dígito a dígito. O problema que se põe é saber como proceder se a soma de dois dígitos exceder o valor máximo representável em BCD ( $9_D = 1001_B$ ). Ou, doutra forma, quando e como gerar um transporte para o dígito BCD seguinte.

Quando a soma de dois dígitos BCD exceder  $9_D$ , adiciona-se  $0110_B = 6_D$  a esse resultado: trata-se de uma correcção destinada a gerar o transporte, acima referido, para o dígito BCD seguinte (note-se que  $6_D = 15_D - 9_D = 1111_B - 1001_B$ ).

Apesar da realização de operações aritméticas em BCD ser mais complexa, este código é bastante utilizado em instrumentos de medida cujos resultados são visualizados através de algarismos decimais em mostradores numéricos.

## 6.2. Código 2421

No *código 2421* os pesos atribuídos aos dígitos binários são 2, 4, 2 e 1, da esquerda para a direita. Daí o seu nome. Este código pode no entanto gerar algumas ambiguidades uma vez que admite para a maioria dos números decimais duas representações, pois existem dois *bits* com o mesmo peso. O dígito  $5_D$ , por exemplo, pode ser codificado por  $0101_B$  ou  $1011_B$ .

Este código é *autocomplementado* para 9, isto é, a diferença entre o dígito 9 e qualquer outro dígito pode obter-se substituindo os 1's por 0's e vice-versa, neste último. Por exemplo,  $9_D - 3_D = 6_D$ ; ora,  $3_B = 0011_B$  e  $6_D = 1100_B$ .

### 6.3. Código 5421

A diferença relativamente aos anteriores é que a ponderação é feita segundo 5, 4, 2 e 1. Por exemplo, o dígito  $5_D$  é codificado por  $1000_B$  e o  $9_D$  por  $1100_B$ .

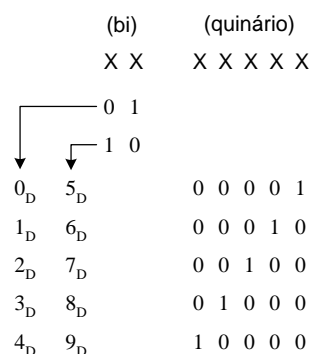
Este código tem a particularidade de conseguir representações para os dígitos 0...4 que diferem dos dígitos 5...9 apenas no *bit* mais significativo: 0 para o primeiro grupo e 1 para o segundo, coincidindo os restantes 3 *bits*.

### 6.4. Excesso 3

Obtém-se do *biário puro* adicionando  $3_D$  às representações de cada dígito. Assim, em vez do habitual  $0_D = 0000_B$ , o primeiro elemento será  $3_D = 0011_B$ . Este código não é ponderado devido à existência deste deslocamento. Tal como o 2421, este código é *autocomplementado* para 9.

### 6.5. Biquinário

A particularidade deste código reside na utilização de 7 dígitos. Daí o seu nome: *bi* (2) + *quinário* (5) = 7. Os dois dígitos mais à esquerda, que podem tomar os valores 01 ou 10, indicam se o valor representado pelos restantes 5 *bits* se encontra na gama 0...4 ou 5...9, respectivamente. Desses 5 *bits*, apenas um apresenta o valor 1, que se vai “deslocando” da direita para a esquerda para indicar qual o dígito decimal da gama considerada. A Figura 3 mostra o processo de geração deste código. Assim, por exemplo, o dígito 5 é codificado por 10 00001.



**Figura 3** – Geração do Código Biquinário

## 6.6. 1 de 10

Utiliza 10 *bits* dos quais apenas um apresenta o valor 1. A posição deste *bit* indica qual o dígito decimal representado.

As tabelas Tabela 8 e Tabela 9 mostram os dez dígitos decimais em cada um dos códigos BCD analisados.

Decimal	Ponderados		
	BCD Natural 8421	2421	5421
0	0000	0000	0000
1	0001	0001	0001
2	0010	0010/1000	0010
3	0011	0011/1001	0011
4	0100	0100/1010	0100
5	0101	0101/1011	0101/1000
6	0110	0110/1100	0110/1001
7	0111	0111/1101	0111/1010
8	1000	1110	1011
9	1001	1111	1100

**Tabela 8** – Exemplos de códigos BCD ponderados

Decimal	Não Ponderados		
	Excesso 3	Biquinário	1 de 10
0	0011	0100001	1000000000
1	0100	0100010	0100000000
2	0101	0100100	0010000000
3	0110	0101000	0001000000
4	0111	0110000	0000100000
5	1000	1000001	0000010000
6	1001	1000010	0000001000
7	1010	1000100	0000000100
8	1011	1001000	0000000010
9	1100	1010000	0000000001

**Tabela 9** – Exemplos de códigos BCD não-ponderados

Qual a razão de ser de todos estes códigos BCD? A que se deve esta diversidade?

A diversidade de códigos BCD constitui uma mais valia no projecto de sistemas digitais, uma vez que permite seleccionar o mais adequado às necessidades concretas de um determinado problema, nomeadamente os que implicam a apresentação de dados aos utilizadores, mais familiarizados com

a notação decimal. Basicamente todos os códigos BCD estabelecem uma ponte de ligação entre o mundo binário e o mundo decimal, facilitando a conversão entre representações binárias e a leitura ou visualização de dígitos decimais. Alguns circuitos digitais podem até operar directamente sobre representações BCD (caso de alguns *contadores* chamados *decade counters*).

Por exemplo, o código BCD natural pode constituir um estágio intermédio da descodificação binário para 7 segmentos. Dado um *byte*, a visualização do seu conteúdo, suposto em binário natural (valor máximo representável =  $255_D$ ), necessita de 3 dígitos decimais: o primeiro na gama 0...2 e os restantes na gama 0...9. Ora, uma notação tal como  $1111\ 1111_B$  encontra-se a uma distância notável de fazer acender/apagar  $3 \times 7 = 21$  segmentos! Um circuito para realizar esta descodificação teria de ter 8 entradas e 21 saídas e seria com certeza muito complexo dado que teria de controlar os 21 segmentos para cada um dos 256 valores binários possíveis. Por isso é preferível implementar o processo em duas fases: passagem a BCD, o que gera os três dígitos  $0010\ 0101\ 0101_B$ ; posterior conversão de cada dígito BCD a 7 segmentos. Além disso, esta abordagem é mais modular, porque a descodificação BCD-7 segmentos é a mesma para todos os dígitos.

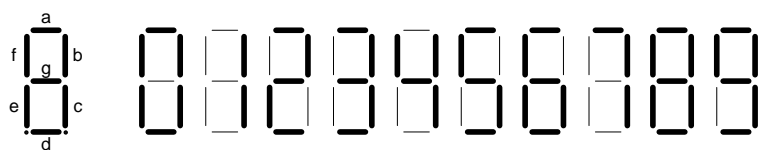
O *biquinário* apareceu pela primeira vez em 1943 e destinava-se a assegurar maior fiabilidade na operação de leitura de uma fita perfurada. O 1-de-10 pode ser usado, por exemplo, para fazer acender uma de 10 lâmpadas ou filamentos com a forma de dígitos decimais.

## 7. Códigos Binários para Tarefas Especiais

Existem ainda outros códigos com características que os tornam apropriados à realização de tarefas específicas.

### 7.1. Códigos para Visualização

Um dispositivo ainda hoje bastante comum destinado à visualização de informação numérica, é o *display* de 7 segmentos. Este dispositivo encontra-se, por exemplo, nas calculadoras de bolso e instrumentos de medida digitais. É composto por sete segmentos implementados à custa de díodos emissores de luz (LED - *Light Emitting Diode*), dispostos conforme indicado na Figura 4, permite reproduzir todos os dígitos do sistema decimal. Vulgarmente estes *displays* incluem também um ou dois pontos decimais destinados a permitir a visualização de números com parte fraccionária.



**Figura 4** – *Display de 7 segmentos e representações possíveis*

De um modo geral, os números são tratados nestes dispositivos em BCD e, antes de serem apresentados em indicadores numéricos luminosos, são convertidos no *código de 7 segmentos*. Este, pode recorrer a lógica positiva – “1” corresponde a um segmento activo – ou negativa – “0” corresponde a um segmento activo. A título de exemplo apresenta-se na Tabela 10 o código de 7 segmentos em lógica positiva.

Decimal	7 Segmentos						
	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

**Tabela 10** – *Código de 7 segmentos em lógica positiva*

## 7.2. Códigos Contínuos

Incluem-se nesta categoria os *códigos de Gray* e de *Johnson*. Ambos são códigos não ponderados e de distância unitária, caracterizados por a representação de valores adjacentes diferir em apenas num único *bit*. A Tabela 11 representa estes códigos.

Decimal	Gray	Johnson
0	0000	00000
1	0001	00001
2	0011	00011
3	0010	00111
4	0110	01111
5	0111	11111
6	0101	11110
7	0100	11100
8	1100	11000
9	1101	10000
10	1111	
11	1110	
12	1010	
13	1011	
14	1001	
15	1000	

**Tabela 11** – Códigos de Gray e de Johnson

Como se pode verificar, o código de Gray é mais condensado no sentido em que representa 16 números com 4 *bits*, ao passo que o de Johnson representa apenas 10 com 5 *bits*. No código de Johnson o valor seguinte pode obter-se a partir do anterior rodando-o à esquerda e fazendo entrar pela direita o MSb depois de complementado<sup>6</sup>:

Valor Inicial:            **1 1 1 1 1**

Rotação:                1 1 1 1 1 ← 0      Complementação

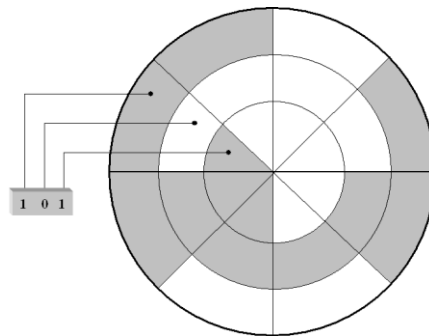
Valor Seguinte:        **1 1 1 1 0**

O código Gray tem ainda a particularidade de ser um código reflectido, como se pode verificar na tabela, se se fosse excluindo o bit mais significativo, ia-se obtendo uma tabela cuja segunda metade é igual à imagem que se obteria se a primeira metade fosse reflectida num espelho.

O código de Gray é utilizado na codificação de posições angulares (dispositivos que convertem ângulos em valores digitais) dado que na passagem de uma posição para outra apenas um *bit* varia. Na Figura 5 encontra-se representado um codificador angular muito simples, constituído por um disco codificado em código Gray. As áreas sombreadas correspondem a condutores metálicos. À medida que o disco roda, os condutores ao longo das quatro pistas passam sob quatro contactos colocados radialmente, um em cada pista, e vão estabelecer ou interromper circuitos. Que permitem

<sup>6</sup> Pode ser facilmente implementado à custa de um *registo de deslocamento* e de uma porta lógica NOT.

expressar em código Gray os valores de 16 ângulos possíveis. Um código binário traria problemas dada a muito provável não simultaneidade na comutação de todos os *bits*, o que daria origem à leitura de uma posição espúria. Por exemplo, se o disco estiver numa posição entre 3 e 4, e se ainda só tiverem mudado os dois bits mais significativos, obter-se-á a leitura 0110, que corresponde a 6, e portanto não haverá erro. Se em vez do código Gray se tivesse utilizado o código binário, a mesma situação conduziria à leitura 0111, que corresponde a 7, tendo-se um erro grosseiro.



**Figura 5** – Um disco em código Gray para leitura da posição angular

Para efectuar a conversão de um número em binário a Gray, começa por se copiar o *bit* mais significativo; a partir daqui e progredindo do bit mais significativo para o menos significativo, dá-se a cada um dos restantes *bits* o valor “0”, se no código binário original o *bit* em consideração for igual ao que se encontra à sua esquerda (mais significativo), e o valor “1” caso contrário. Ou seja, cada variação 0->1 ou 1->0 produz um “1” e cada manutenção de valor 0->0 ou 1->1 produz um “0”. Por exemplo:

$$\begin{array}{ccccccc}
 1 & 1 & 1 & 1 & 0 & 0 & 1_B \\
 \begin{array}{c} \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \end{array} & \begin{array}{c} \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \end{array} & \begin{array}{c} \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \end{array} & \begin{array}{c} \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \end{array} & \begin{array}{c} \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \end{array} & \begin{array}{c} \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \end{array} & \begin{array}{c} \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \\ \text{=} \end{array} \\
 \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
 1 & 0 & 0 & 0 & 1 & 0 & 1 \text{ (gray)}
 \end{array}$$

O processo inverso, ou seja, a conversão de Gray para binário, é feita da seguinte forma: copia-se o *bit* mais significativo; em seguida, continuando a ler o valor do código Gray do bit mais significativo para o menos significativo, no código binário que se está a construir mantém-se o valor do *bit* anterior, se o *bit* em consideração no código Gray for 0, e o inverso do *bit* anterior, se o *bit* em consideração no código Gray for 1. Por exemplo:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \text{ (gray)}$$

$$\downarrow \downarrow \downarrow \downarrow \downarrow \downarrow$$

$$\begin{matrix} = & = & = & \neq & = & \neq \\ 1 & 1 & 1 & 1 & 0 & 0 & 1_B \end{matrix}$$

### 7.3. Códigos para Detecção de Erros

Apesar da grande imunidade ao ruído, a transmissão de dados no formato digital pode ainda estar sujeita a erros devido a vários factores tais como distância, tipo de interface, magnitude e origem das perturbações externas e, em última análise, falha física. Um erro consiste na troca de um (ou vários) 1 por 0, ou vice-versa.

Devido a isto têm sido desenvolvidos vários sistemas de codificação cujo objectivo é o de permitir reconhecer, na recepção, que um erro ocorreu. Na sua forma mais simples mas também mais vulgar, supõe-se que o erro ocorreu em apenas 1 *bit* de entre os  $n$  *bits* que se pretendiam transmitir.

De entre estes códigos são bastante conhecidos os de *Hamming* e os *CRC* (*Cyclic Redundancy Codes*). Estes últimos têm uma importante aplicação nos sistemas *RAID* (*Redundant Array of Inexpensive Disks*) um método que permite distribuir por  $n+1$  discos a informação destinada a  $n$  discos, proporcionando assim uma certa redundância que permite reconstruir os dados em caso de falha.

Mais vulgares e mais simples do que estes são o *código de paridade par* (*even parity*) e o de *paridade ímpar* (*odd parity*). Em qualquer deles se usa um *bit* adicional chamado *bit de paridade* que se justapõe à informação a transmitir, normalmente *byte a byte*. No código de paridade par este *bit* é escolhido de forma a que os 9 *bits* assim obtidos passem a conter um número par de 1's incluindo o *bit* de paridade. No de paridade ímpar o *bit* adicional é escolhido de forma a que passem a conter um número ímpar de 1's incluindo o *bit* de paridade. A Tabela 12 mostra alguns exemplos.

Byte original	Paridade Par	Paridade Ímpar
0000 0000	0000 0000 0	0000 0000 1
1111 1111	1111 1111 0	1111 1111 1
0000 0111	0000 0111 1	0000 0111 0
0111 1111	0111 1111 1	0111 1111 0

**Tabela 12** – Códigos de paridade par e ímpar aplicados a 1 *byte*



Estes códigos são utilizados, por exemplo, na transmissão de dados em série através das portas RS232 de computadores e periféricos. Existem circuitos digitais especiais para a geração e interpretação destes códigos, chamados *geradores de paridade (parity checkers/generators)*. Outro tipo de circuito, mais complexo, recebe um *byte* em paralelo e transmite-o em série ou recebe-o em série e disponibiliza-o em paralelo. Trata-se da UART (*Universal Asynchronous Receiver/Transmitter*) que inclui, ela própria, gerador de paridade.

## 8. Representação de Caracteres

Além de números, os sistemas digitais e os computadores, em particular, lidam com caracteres. Mais uma vez, trata-se de uma mera questão de codificação.

### Código ASCII ou ANSI X3.4

O código ASCII (*American Standard Code for Information Interchange*), também designado por ANSI X3.4 (*ANSI - American National Standards Institute*), é provavelmente uma das formas de codificação mais conhecidas.

O conjunto de caracteres ASCII standard encontra-se dividido em quatro grupos de 32 caracteres cada. Cada grupo é diferenciado entre si pelos valores dos bits 5 e 6 (em que o bit 0 representa a posição menos significativa) do código ASCII, tendo este código o tamanho de um byte (8 bits). A tabela standard não usa o bit 7 pelo que este se encontra sempre a zero. O que significa que o conjunto de caracteres ASCII consome apenas metade dos códigos possíveis. Os restantes 128 códigos são utilizados para vários caracteres especiais incluindo caracteres internacionais (acentuação, ...), símbolos matemáticos, caracteres para desenho de linhas, entre outros.

Bit 6	Bit 5	Grupo
0	0	Grupo 1
0	1	Grupo 2
1	0	Grupo 3
1	1	Grupo 4

Cada um dos grupos existentes representa um dado tipo de caracteres:

Grupo 1 – Este grupo contém um conjunto especial de caracteres designados caracteres de controlo. Contém os códigos ASCII compreendidos entre 0 e 1F<sub>H</sub>.

Grupo 2 – Este grupo contém os caracteres representativos dos dígitos ( $30_H$  a  $39_H$ ) e ainda símbolos de pontuação e outros caracteres especiais como o carácter espaço ( $20_H$ ). Contém os códigos ASCII compreendidos entre  $20_H$  e  $3F_H$ .

Grupo 3 – Este grupo contém os caracteres maiúsculos ( $41_H$  a  $5A_H$ ) e ainda alguns caracteres especiais. Contém os códigos ASCII compreendidos entre  $40_H$  e  $5F_H$ .

Grupo 4 – Este grupo contém os caracteres minúsculos ( $61_H$  a  $7A_H$ ) e ainda alguns caracteres especiais e o backspace. Contém os códigos ASCII compreendidos entre  $60_H$  e  $7F_H$ .

Algumas observações :

- O espaço (SP de space) tem o código  $20_H$ ;
- Os algarismos iniciam-se com o 0 ( $30_H$ ) e prosseguem sequencialmente até ao 9 ( $39_H$ ). Ou seja, o código ASCII de cada algarismo é igual ao seu código binário adicionado de  $30_H$ . Por exemplo, a  $5_D = 101_B$  corresponde, em ASCII, o valor  $101_B + 011\ 0000_B = 011\ 0101_B$ . Assim, subtraindo  $30_H$  ao valor ASCII de um dígito, obtém-se o binário desse dígito ex: '1' -  $30_H = 1$ . Isto acontece porque o *nibble* (4 bits) menos significativo do código ASCII de um dígito é o equivalente binário do número por ele representado. Como os dígitos estão representados na gama  $30_H$  a  $39_H$  ao se retirar  $30_H$  fica-se apenas com o valor binário do dígito em questão. Desta forma, utilizando apenas lógica booleana podem-se converter dígitos nos seus equivalente numéricos e vice-versa;
- As letras maiúsculas iniciam-se com o 'A' em ( $41_H$ ) e prosseguem sequencialmente até ao 'Z' ( $5A_H$ ). Ou, de forma equivalente: o código ASCII de cada letra é igual à sua ordem no alfabeto ('A' é a primeira) adicionado de  $40_H$ . Por exemplo, 'B', 2ª letra, tem o código  $10_B + 100\ 0000_B = 100\ 0010_B$ ;
- As letras minúsculas iniciam-se em  $61_H$ , sendo o seu ordenamento similar ao das letras maiúsculas;
- O valor em binário dos caracteres maiúsculos e minúsculos difere apenas no valor do bit 5. (bit 5 = 0 – carácter maiúsculo; bit 5 = 1 carácter minúsculo). Resultado desta relação: as conversões de maiúsculas para minúsculas (e vice versa) podem ser realizadas rapidamente com recurso a operações de lógica booleana.
- Os caracteres especiais encontram-se todos abaixo do código do espaço.

Estes caracteres especiais não têm representação gráfica num écran ou impressora. Trata-se

essencialmente de *caracteres de controlo* com funções específicas, tais como:

- $07_H$  BEL Ligar o *beep* do computador;
- $0A_H$  LF *Line Feed*: avançar para linha seguinte;
- $0C_H$  FF *Form Feed*: avançar para a página seguinte;
- $0D_H$  CR *Carriage Return*: voltar ao início da linha actual.

Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII
0	00	NUL	32	20	SP	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	\$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	^	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	/	124	7C	
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

**Tabela 13** – Código ASCII

## UNICODE

O código ASCII é contudo muito limitado quando se pretendem representar caracteres nacionais, uma vez que foi inicialmente concebido para o alfabeto latino, apenas. De facto, as variantes em termos de acentuações e caracteres mesmo considerando apenas línguas ocidentais, são imensas. Este problema tem sido resolvido recorrendo a variantes do código ASCII original, que são accionadas consoante a língua em que se pretende trabalhar.

Para resolver este problema, proporcionando assim um fácil intercâmbio de documentos à escala mundial, foi desenvolvido o *Unicode*, que se encontra ainda em evolução. Inicialmente pensou-se numa codificação em 16 *bits*, o que daria suporte a cerca de 65.000 caracteres diferentes. Contudo, actualmente o Unicode suporta 3 formas de codificação: em 8 (designada por *UTF-8*), 16 e 32 *bits*. Com isto consegue codificar-se mais de 1 milhão de caracteres, o que é suficiente para todos os símbolos conhecidos incluindo os provenientes de documentos históricos.

A	0000 0000 0100 0001
S	0000 0000 0101 0011
C	0000 0000 0100 0011
I	0000 0000 0100 1001
I	0000 0000 0100 1001
	0000 0000 0010 0000
天	0101 1001 0010 1001
地	0101 0111 0011 0000
	0000 0000 0010 0000
س	0000 0110 0011 0011
ل	0000 0110 0100 0100
ط	0000 0110 0011 0111
م	0000 0110 0100 0101
	0000 0000 0010 0000
α	0000 0011 1011 0001
≤	0010 0010 0111 0000
γ	0000 0011 1011 0011

**Tabela 14** – Um excerto de codificação Unicode em 16 *bits*

## Código EBCDIC

A sigla *EBCDIC* provém de *Extended Binary Coded Decimal Interchange Code*. Note-se a presença de *Binary Coded Decimal (BCD)* nesta designação. De facto, o EBCDIC é uma extensão do BCD natural, de 4 para 8 bits, usado pela IBM nos seus *mainframes* (computadores de grande porte) e série AS400.