ISCTE IUL
Instituto Universitário de Lisboa

Departamento de Ciências e Tecnologias da Informação

Arquitectura de Computadores (I)

Textos de apoio

– Representação da Informação –

Draft – Versão 0.03b – Set./2011 – João Baptista e Tomás Brandão.

Índice

1. REPRESENTAÇÃO NUMÉRICA EM DIFERENTES BASES 5

1.1. BASE 10 (DECIMAL) 5
1.2. SIGNIFICADO DA REPRESENTAÇÃO NUMÉRICA 5
1.3. BASE 8 (OCTAL) 6
1.4. BASE 16 (HEXADECIMAL) 7
1.5. BASE 2 (BINÁRIO) 7
1.6. GENERALIZAÇÃO DA REPRESENTAÇÃO NUMÉRICA 8
1.7. A PARTE FRACCIONÁRIA 9

2. OPERAÇÕES 10

2.1. ADIÇÃO 10
2.2. SUBTRACÇÃO 11
2.3. SOMAR E SUBTRAIR EM BINÁRIO 12

3. CONVERSÕES ENTRE BASES 13

3.1. CONVERSÃO DE UMA BASE *B* PARA BASE DECIMAL 13
3.2. CONVERSÃO DE BASE DECIMAL PARA UMA BASE *B* 13
3.3. CONVERSÕES ENVOLVENDO AS BASES 2, 8 E 16 16
3.4. OUTRAS CONSIDERAÇÕES 19

4. REPRESENTAÇÃO DE NÚMEROS INTEIROS EM BINÁRIO 20

4.1. MAGNITUDE E SINAL 20
4.2. REPRESENTAÇÃO EM COMPLEMENTO 20
4.3. OPERAÇÕES ARITMÉTICAS EM COMPLEMENTO PARA 2 21
4.4. OVERFLOW 23

5. REPRESENTAÇÃO DE CARACTERES 24

5.1. CÓDIGO ASCII 24
5.2. CÓDIGO UNICODE 27

Representação da informação

1. Representação numérica em diferentes bases

1.1. Base 10 (decimal)

A representação numérica comum utiliza a base 10. Isso significa que existem 10 algarismos (ou dígitos) diferentes, representados pelos símbolos de 0 a 9. Utilizando apenas estes algarismos podemos representar qualquer número. Começando pelos números mais simples, representáveis apenas por um dígito, temos:

0, 1, 2, …, 9, 10

Ao passar de '9' para '10' ocorre o primeiro "incidente" na contagem: esgotou-se a capacidade de representação de números apenas com um dígito – passamos a usar dois dígitos, em que o da esquerda tem maior peso – neste caso tem o seu valor multiplicado por 10, ou seja, pelo valor da base de numeração utilizada;

Continuando a contagem, temos:

…, 10, 11, 12, …, 18, 19, 20

Ao chegar a '19' esgotaram-se os dígitos na posição com menor peso. Por isso aumenta-se '1' na posição seguinte; já contamos duas vezes a *base*, ou seja, vamos em 2×10.

Retomando a contagem:

…, 20, 21, 22, …, 97, 98, 99, 100

Ao passar de '99' para '100' esgotaram-se os dígitos na segunda posição; isso significa que contámos 10 vezes a *base*, ou seja $10\times10 = 10^2$. Um dígito na 3ª posição vale portanto a *base ao quadrado*.

…, 100, 101, …, 998, 998, 1000, …

A partir de '1000', o dígito na quarta posição vale por 10^3.

1.2. Significado da representação numérica

Com base no que foi visto na secção anterior, é fácil de perceber que um número representado em base 10 pode ser decomposto da forma indicada nos seguintes exemplos:

$25 = 2\times10 + 5$
$367 = 3\times10^2 + 6\times10 + 7$
$2745 = 2\times10^3 + 7\times10^2 + 4\times10 + 5$

Em suma, da direita para a esquerda: um algarismo na primeira posição vale por si próprio, na segunda posição vale pela *base*, na terceira pela *base ao quadrado*, na quarta pela *base ao cubo*, e assim sucessivamente. Por exemplo:

$$8317 = 8\times10^3 + 3\times10^2 + 1\times10 + 7$$

Uma vez que $10^0 = 1$, pode-se uniformizar a representação da seguinte forma:

$$8317 = 8\times10^3 + 3\times10^2 + 1\times10^1 + 7\times10^0$$

Para manter a coerência, designam-se as posições, da direita para esquerda, por posição 0, 1, 2 e assim sucessivamente. Ter a posição é portanto o mesmo que ter o valor do expoente. Designa-se a potência respectiva por peso. Sendo assim, pode-se dizer que um algarismo na posição 0 tem peso $10^0 = 1$; na posição 1 tem peso $10^1 = 10$; na posição 2 tem peso $10^2 = 100$, etc.

Generalizando, um dígito d_i na posição i tem peso 10^i, ou seja, vale

$$d_i \times 10^i$$

Em geral, se um número está representado em base decimal e tem n dígitos – d_0 a d_n – pode ser escrito do seguinte modo:

$$d_n\ d_{n-1}\ \ldots\ d_2\ d_1\ d_0$$

e o seu valor numérico pode ser escrito na forma:

$$d_n\times10^n + d_{n-1}\times10^{n-1} + \ldots + d_2\times10^2 + d_1\times10^1 + d_0\times10^0$$

1.3. Base 8 (octal)

O que foi visto para a base 10 generaliza-se facilmente para outras bases.

Considerando por exemplo a base 8, em vez de 10 dígitos, passamos a ter agora 8 dígitos – e para não inventar o que já está inventado, vamos representá-los pelos símbolos de '0' a '7'. O processo de contagem é em tudo semelhante ao usado em base 10. Começamos com os números representáveis apenas com um algarismo, que agora vão apenas até '7':

0, 1, 2, 3, 4, 5, 6, 7

Na base 8, o último número representável com apenas um algarismo será portanto o '7'. O número que se segue na contagem será o '10' (que tem o valor $1\times8 + 0 = 8$)

Sendo assim a nossa sequência de contagem é:

…, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20

Repare que ao chegar a '17' mais uma vez se esgotaram os dígitos na posição com menos peso: aumenta-se 1 na posição seguinte; já contámos 2 vezes a *base*, ou seja, vamos em $2 \times base$ (e o valor é 2×8);

Seguindo a contagem:

…, 20, 21, 22, …, 75, 76, 77

Repare que o ‘77’ será o maior número com 2 dígitos representável em base 8 (e o seu valor é 7×8 + 7 = 63). A seguir ao ‘77’ virá então o número representado por ‘100’ – contámos *base*×*base* vezes; o dígito ‘1’ na terceira posição vale a *base ao quadrado* (neste caso 2×8^2).

E seguindo este raciocínio poderíamos continuar a contagem indefinidamente:

…, 77, 100, 101, 102, …, 107, 110, 111, …, 117, 120, 121, …, 775, 776, 777, 1000, …

1.4. Base 16 (hexadecimal)

Vamos agora considerar a base 16. Na base 16 teremos 16 dígitos; os primeiros 10 são representados da maneira habitual, isto é, usando os algarismos de ‘0’ a ‘9’; a partir daí são usadas as primeiras letras do alfabeto: ‘A’, ‘B’, ‘C’, ‘D’, ‘E’ e ‘F’.

O processo de contagem é em tudo semelhante ao que foi visto até este ponto. A única particularidade é que agora os números representáveis com um dígito vão até ao ‘F’ (cujo valor é 15):

0, 1, 2, …, 9, A, B, C, D, E, F

Repare agora que depois do ‘9’ não se passou nada de especial, pois ainda há o ‘A’ (cujo valor é 10), depois o ‘B’ (cujo valor é 11), e assim sucessivamente até ao ‘F’, que será o maior número representável com apenas um algarismo na base 16.

Continuando a contagem:

…, E, F, 10

Ao chegar a ‘F’ esgotaram-se os dígitos, logo esgotou-se a capacidade de representação de números só com um dígito – passam-se a ter que usar dois dígitos. Na representação ‘10’, o ‘1’ da esquerda vale a *base* (o valor de ‘10’ na base 16 será portanto 1×16 + 0 = 16);

…, 10, 11, 12, …, 19, 1A, 1B, 1C, 1D, 1E, 1F, 20

Em ‘20’ contámos duas vezes o número de dígitos; vamos em 2×*base* (e o valor respectivo é 2×16 = 32).

…, 20, 21, …, 99, 9A, 9B, 9C, 9D, 9E, 9F, A0, A1, … FC, FD, FE, FF, 100

O número representado por ‘FF’ é o maior valor representado com 2 dígitos na base 16. O seu valor é 15×16+15 = 255. Na representação hexadecimal ‘100’, o ‘1’ da esquerda vale por 16^2.

1.5. Base 2 (binário)

Especialmente importante, dado que é com essa representação com que os computadores trabalham, é a base 2, que tem apenas dois dígitos: ‘0’ e ‘1’. O processo de contagem é idêntico ao praticado em qualquer outra base. No entanto, e como apenas existem dois dígitos nesta base, ao fazer uma contagem rapidamente aparecem sequências com muitos dígitos, como será visto mais à frente. Começando a contagem:

0, 1, 10

Esgotada a capacidade de representação com um dígito, passam-se a usar dois dígitos; o dígito da direita vale uma vez a *base*, ou seja, vale 2;

…, 10, 11, 100

'11' é o maior número representável com apenas dois dígitos. Em '100' passamos a ter três dígitos em que o da esquerda vale a *base ao quadrado* (2^2);

…, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, 10000, …

E poder-se-ia continuar indefinidamente. Repare que na representação '1000', o '1' da esquerda vale pela *base ao cubo* (2^3); e em '10000', o dígito da esquerda vale pela *base à quarta* (2^4).

1.6. Generalização da representação numérica

O conceito de representação numérica apresentado no ponto 1.1 para a base decimal generaliza-se facilmente para outras bases. Em qualquer base um número representa-se como uma sequência de dígitos. Sendo B a base, um dígito d_i na posição i tem peso B^i ou seja vale

$$d_i \times B^i$$

Em geral, para um dado número com os dígitos:

$$d_n\, d_{n-1} \ldots d_2\, d_1\, d_0\ ,$$

o seu valor numérico, dada a base B, é:

$$d_n \times B^n + d_{n-1} \times B^{n-1} + \ \ldots + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0$$

Para clarificar a representação e evitar ambiguidades, pode-se explicitar a base de numeração usando a seguinte notação:

$(312)_{10}$ – a sequência de dígitos 312 na base 10;

$(312)_8$ – a sequência de dígitos 312 na base 8;

$(312)_{16}$ – a sequência de dígitos 312 na base 16.

O valor respectivo de cada um (em decimal) é o que resulta da expressão anterior. Assim:

$$(312)_{10} = 3\times10^2 + 1\times10^1 + 2\times10^0$$
$$(312)_8 = 3\times8^2 + 1\times8^1 + 2\times8^0$$
$$(312)_{16} = 3\times16^2 + 1\times16^1 + 2\times16^0$$

Obviamente os três números representados têm valores diferentes – a *representação* e o *valor* correspondente são coisas distintas. Em geral, sequências de dígitos iguais em bases diferentes representam valores diferentes (só serão os mesmos caso tenham apenas um dígito). Pode confirmar, por exemplo, que:

$$(312)_{10} = (470)_8 = (138)_{16} = (100111000)_2$$
$$(202)_{10} = (312)_8 = (CA)_{16} = (11001010)_2$$
$$(786)_{10} = (1422)_8 = (312)_{16} = (1100010010)_2$$

1.7. A parte fraccionária

A representação estende-se facilmente para a parte fraccionária. Dada uma sequência de dígitos na base *B*:

$$d_n\, d_{n-1} \ldots d_2\, d_1\, d_0\, .\, d_{-1}\, d_{-2} \ldots\, d_{-m}\,,$$

em que o ponto separa a parte inteira e a parte fraccionária, o valor associado é

$$d_nB^n + d_{n-1}B^{n-1} + \ldots + d_2B^2 + d_1B^1 + d_0B^0 + d_{-1}B^{-1} + d_{-2}B^{-2} + \ldots d_{-m}\,B^{-m}$$

Por exemplo, a sequência 42.57 em decimal vale:

$$(42.57)_{10} = 4\times10^1 + 2\times10^0 + 5\times10^{-1} + 7\times10^{-2}$$

Em base 10, o dígito à direita do ponto tem peso 10^{-1}, ou seja uma décima; o dígito seguinte tem peso 10^{-2}, ou seja uma centésima; e assim sucessivamente. Genericamente, o dígito de posição *i* à direita do ponto (dita posição *-i*) tem peso B^{-i}.

Noutras bases, o raciocínio é o mesmo. A diferença está nos pesos, que vão ser diferentes dado que as bases também o são.

Exemplos:

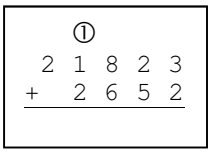
$$(42.57)_8 = 4\times8^1 + 2\times8^0 + 5\times8^{-1} + 7\times8^{-2}$$
$$(4C.A7)_{16} = 4\times16^1 + 12\times16^0 + 10\times16^{-1} + 7\times16^{-2}$$
$$(10.01)_2 = 2 + 1/4$$

2. Operações

2.1. Adição

Contar é basicamente tudo o que é necessário para fazer as operações elementares de somar e subtrair. Vamos agora rever os procedimentos que usamos para a realização destas operações.

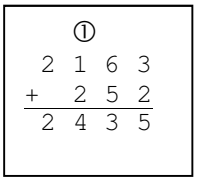
Exemplo (em base 10), da direita para a esquerda:



- ao primeiro algarismo, 3, juntamos 2 (a partir de 3 contamos mais 2);
- a partir de 2 contamos mais 5;
- a partir de 8 contamos 6, para dar 14. Aqui há um incidente: a contagem ultrapassa 10, ou seja, a base. Deixa-se como resultado o algarismo das unidades 4, e juntam-se os restantes 10 somando 1 na posição seguinte. Diz-se então que "*há 1 de transporte*" (ou, de uma forma mais informal, "*e vai um*"...);

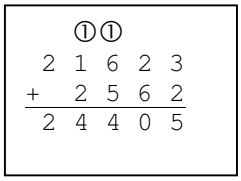
O mecanismo para somar é o mesmo em qualquer outra base. Vejamos alguns exemplos em bases diferentes:

Exemplo, em base 8, da direita para a esquerda:



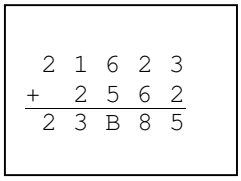
- 3 + 2 dá 5;
- 6 + 5 dá 13 (a partir do '6' contamos 5 vezes na base 8: 7, 10, 11, 12, 13). Como habitualmente, deixamos o algarismo '3' no resultado e juntamos o '10', fazendo o transporte de '1' para a posição seguinte;
- …

Exemplo, ainda em base 8:



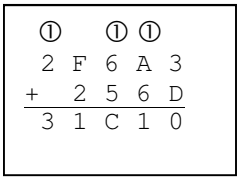
- 3 + 2 dá 5;
- 2 + 6 dá 10; ou seja, dá '0' *e vai um*;
- 6 + 5 + 1 (que vem de trás) dá '14'; ou seja, dá '4' *e vai um*;
- …

Exemplo, agora em base 16:



- 2 + 3 dá 5;
- 2 + 6 dá 8;
- 6 + 5 dá B (ou seja, a partir de '6' conta-se 5 na base 16: 7, 8, 9, A, B);
- …

Exemplo, outra vez em base 16



- 3 + D dá 10 (D + 3 => E F 10)
- 1 + A + 6 dá 11 (B + 6 => C D E F 10 11)
- 1 + 6 + 5 dá C
- F + 2 dá 11
- …

2.2. Subtracção

Os mecanismos que usamos vulgarmente para fazer a subtracção aplicam-se também, com os mesmos princípios, quer à base 10 quer às outras bases.

Considere uma subtracção elementar, por exemplo 7 – 2. A maneira mais básica de fazer a operação é mais uma vez contar: neste caso contar de baixo (*i.e.* do subtraendo) para cima (*i.e.* para o subtractor). O número de vezes que contarmos será a diferença.

Para fazer a subtracção de números com mais algarismos generalizamos o processo seguindo da direita para a esquerda. Em cada posição fazemos a diferença entre o algarismo de cima e o de baixo.

Exemplo: (em base 10) neste caso faríamos:

```
  2 9 3 7
– 7 6 2
  2 1 7 5
    ①
```

- 2 para 7: conta-se 3, 4, 5, 6, 7, ou seja 5 vezes;
- 6 para 3: conta-se 7 vezes e *vai um* de transporte;
- o transporte soma ao 7, assim fazemos 8 para 9 e dá 1
- …

Recorde que numa subtracção *o transporte se junta ao algarismo de baixo*. Ao fazer *6 para 3* estamos na realidade a fazer a diferença para '13' – por consequência temos que subtrair '10' na posição seguinte. Podemos fazer isso retirando '1' ao algarismo de cima ou, como fazemos habitualmente, adicionando '1' ao de baixo.

Salienta-se ainda a situação de transporte evidenciada no seguinte exemplo:

Neste caso, ao fazer 7 para 2 há um transporte. O transporte junta-se ao 9 para fazer 10 para 13, havendo por isso um novo transporte.

```
  2 9 3 2
– 7 9 7
  2 1 3 5
    ① ①
```

Note bem que apesar de nesta situação pensarmos *0 para 3* e realidade é que estamos a fazer *10 para 13*; não nos podemos esquecer deste transporte.

Exemplo: vejamos agora o que acontece em base 8

```
  2 4 3 7
– 4 6 2
  1 7 5 5
  ① ①
```

- 2 para 7 dá 5;
- 6 para 3 dá 5 (conta 7, 10, 11, 12, 13, ou seja 5 vezes) *e vai um*;
- (4+1) para 4 dá 7 (conta 6, 7, 10, 11, 12, 13, 14, *i.e.* 7 vezes) *e vai um*;
- 1 para 2 dá 1.

Exemplo: novamente em base 8

```
  2 4 3 0
– 5 7 2
  1 6 3 6
  ① ① ①
```

- 2 para 0 dá 6 (conta 3, 4, 5, 6, 7, 10, *i.e.* 6 vezes) *e vai um*;
- 7 + 1 dá 10; 0 para 3 dá 3, e *vai um*;
- 5 + 1 dá 6; 6 para 4 dá 6 e *vai um*.

Exemplo, agora em base 16:

```
  C B C 2
- 5 7 A
-------
  C 6 4 8
      ①
```

- A para 2 dá 8 (conta B C D E F 10 11 12) *e vai um*;
- 7 + 1 dá 8; 8 para C dá 4;
- 5 para B dá 6;
- 0 para C dá C.

Exemplo: novamente em base 16

```
  1 2 3 4
- 5 7 A
-------
    C B A
  ① ① ①
```

- A para 4 dá A (conta B, C, D, E, F, 10, 11, 12, 13, 14) *e vai um*;
- 7 + 1 dá 8; 8 para 3 dá B e *vai um*;
- 5 + 1 dá 6; 6 para 2 dá C e *vai um*;
- 0 + 1 dá 1; 1 para 1 dá 0.

2.3. Somar e subtrair em binário

As operações em binário são em tudo semelhantes. Dado que apenas existem dois dígitos, as operações possíveis são muito poucas.

Na prática, tudo se resume a saber somar dois dígitos, ou três quando há transporte. As possibilidades de soma de dois dígitos são apenas:

0 + 0, que dá 0;

1 + 0, que dá 1;

1 + 1, que dá 0 *e vai um*.

Na soma de três dígitos há apenas uma possibilidade adicional:

1 + 1 + 1, que dá 1 *e vai um*.

Exemplo: uma soma de números representados em binário

```
    ① ①
  1 0 1 1 0
+ 1 0 1 1 1
---------
1 0 1 1 0 1
```

- 0+1 dá 1;
- 1+1 dá 0 *e vai um*;
- 1+1+1 dá 1 *e vai um*;
- ...

Na subtracção as possibilidades também são muito limitadas.

Exemplo:

```
  1 0 1 1 0 1
- 1 0 1 1 1
---------
    1 0 1 1 0
  ①   ① ①
```

- 1 para 1 dá 0;
- 1 para 0 dá 1, *e vai um*;
- 1 e 1 dá 0 (ou seja 10) para 1 (ou seja 11) dá 1 *e vai um*;
- 1 e 0 dá 1 para 1 dá 0;
- ...

3. Conversões entre bases

3.1. Conversão de uma base *B* para base decimal

Considere agora o problema de *dado um número representado numa base de numeração B obter a sua representação em base 10.*

Para resolver este problema, vamos em primeiro lugar relembrar a forma geral de representação de um número, indicada na secção 1.6. Dado uma sequência de algarismos

$$d_n\ d_{n-1}\ ...\ d_2\ d_1\ d_0$$

numa base de numeração *B*, o respectivo valor numérico é:

$$d_n \times B^n + d_{n-1} \times B^{n-1} + \ ... + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0$$

Para converter para a base 10 um número qualquer representado numa base *B*, basta calcular o valor da expressão. Assim, por exemplo:

$(312)_{10} = 3\times10^2 + 1\times10^1 + 2\times10^0 = 3\times100 + 10 + 2$
$(312)_8 = 3\times8^2 + 1\times8^1 + 2\times8^0 = 3\times64 + 8 + 2$
$(312)_{16} = 3\times16^2 + 1\times16^1 + 2\times16^0 = 3\times256 + 16 + 1$
$(1001)_2 = 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0 = 8 + 1$

Caso o número contenha também uma parte fraccionária, pode-se aplicar o que já foi visto na secção 1.7.

$(4.02)_8 = 4\times8^0 + 0\times8^{-1} + 2\times8^{-2} = \ldots$
$(A.BC)_{16} = 10 + 11\times16^{-1} + 12\times16^{-2} = \ldots$
$(101.101)_2 = 4 + 1 + 1/2 + 1/16 = \ldots$

3.2. Conversão de base decimal para uma base *B*

Vejamos agora o problema da conversão de decimal para uma outra base. Ou seja o problema de *dado um número representado em base 10, encontrar a representação do mesmo número noutra base qualquer.*

3.2.1 Parte inteira

Vamos considerar, em primeiro lugar a parte inteira do número. Para converter um número inteiro para base *B* aplica-se o seguinte método:

- dividir (divisão inteira) sucessivamente pela base *B* até obter o quociente 0;
- o resultado da conversão obtém-se juntando, por ordem inversa, os restos dessas divisões inteiras sucessivas;

Note que, ao fazer uma divisão por *B*, o resto é necessariamente um número menor que a base, ou seja, este resto *é um algarismo* da base. Deste modo, juntando os restos sucessivos estamos a formar um número que é válido na base *B*.

Exemplo: pretende-se representar $(1003)_{10}$ em octal

1003 : 8 = 125 com resto **3**
125 : 8 = 15 com resto **5**
15 : 8 = 1 com resto **7**
1 : 8 = 0 com resto **1**

o resultado é $(1753)_8$.

Exemplo: converter o mesmo número, mas agora para hexadecimal

1003 : 16 = 62 com resto **11 (B)**
62 : 16 = 3 com resto **14 (E)**
3 : 16 = 0 com resto **3**

o resultado é $(3EB)_{16}$.

Exemplo: converter novamente o mesmo número, agora para binário

1003 : 2 = 501 com resto **1**
501 : 2 = 250 com resto **1**
250 : 2 = 125 com resto **0**
125 : 2 = 62 com resto **1**
62 : 2 = 31 com resto **0**
31 : 2 = 15 com resto **1**
15 : 2 = 7 com resto **1**
7 : 2 = 3 com resto **1**
3 : 2 = 1 com resto **1**
1 : 2 = 0 com resto **1**

o resultado é $(11\ 1110\ 1011)_2$.

3.2.2 Parte fraccionária

Vejamos agora a parte fraccionária. Aqui o método consiste em *multiplicar o número pela base e recolher parte inteira do número resultante da multiplicação*. Mais uma vez, repare que multiplicando um número só com parte fraccionária (*i.e.* com parte inteira 0) pela base *B* se obtém um número menor que *B* – ou seja *um algarismo da base*.

Exemplo: converter (0.6328125) para octal

$0.6328125 \times 8 = \mathbf{5}.0625$
$0.0625 \times 8 = \mathbf{0}.5$
$0,5 \times 8 = \mathbf{4}.0$

o resultado é $(0.504)_8$.

Assim: em cada passo multiplica-se o número pela base. A parte inteira do resultado dá um algarismo do número. Com a parte fraccionária do resultado repete-se o procedimento.

Eventualmente, como acontece neste exemplo, obtemos um resultado com parte fraccionária 0 e o processo termina. Nestas condições obtemos uma *conversão exacta* – um número na base de conversão exactamente igual ao original.

Mas nem sempre o processo corre tão bem. Vejamos estas duas conversões de um número decimal para binário:

Exemplo: converter: converter $(0.125)_{10}$ para binário

```
0.125 × 2 = 0.250
0.250 × 2 = 0.5
0.5   × 2 = 1.0
```

resultado $(0.125)_{10} = (0.001)_2$.

Exemplo: converter $(0.2)_{10}$ para binário

```
0.2 × 2 = 0.4
0.4 × 2 = 0.8
0.8 × 2 = 1.6
0.6 × 2 = 1.2
0.2 × 2 = 0.4
...
```

Neste caso o processo não termina. Aliás, podemos até concluir que entra num padrão de repetição, originando uma "dízima periódica infinita".

Nestas condições, o número original não pode ser representado com exactidão na base 2. O número de bits é geralmente limitado, ficando-se com uma representação aproximada. Por exemplo, usando um máximo de 16 dígitos na parte fraccionária, o resultado aproximado seria:

$$(0.0011\ 0011\ 0011\ 0011)_2 = (0.1999969482421875)_{10}$$

3.2.3 Síntese

Está agora reunido o conjunto de instrumentos que vamos usar para conversão entre bases.

- Para converter de uma base qualquer para decimal, usa-se a expressão geral indicada na secção 3.1.
- Para converter de decimal para uma outra base *B* usa-se o método de divisões sucessivas apresentado na secção 3.2.1.
- Se o número tiver parte fraccionária, fazemos a conversão separada da parte inteira e da parte fraccionária. A esta última aplicamos o método de multiplicações sucessivas apresentado na secção 3.2.2, tendo em conta que neste caso poderá não ser possível obter uma representação exacta.
- Finalmente, se o problema for uma conversão entre duas bases, B_1 e B_2, ambas diferentes da base 10, podemos:
    - em primeiro lugar converter de B_1 para base 10, …
    - … e depois converter o resultado da base 10 para B_2.

3.3. Conversões envolvendo as bases 2, 8 e 16

3.3.1 Binário

Tudo o que se disse anteriormente é válido tanto para binário como para qualquer outra base. Mas como em binário existem apenas dois dígitos, as expressões geralmente tomam formas mais simplificadas, permitindo de algum modo agilizar os processos de raciocínio.

Seja, por exemplo, a conversão de um número binário para decimal.

$$(101\ 1001)_2 = 1\times2^6 + 0\times2^5 + 1\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0$$

As potências de 2 multiplicam por 0 ou por 1 – no primeiro caso para dar 0: no segundo caso para dar a própria potência de 2. Ou seja, mais simplificadamente, podemos escrever:

$$(101\ 1001)_2 = 2^6 + 2^4 + 2^3 + 1$$

esta expressão acaba por ter uma leitura muito directa: o que de facto interessa são os '1's do número binário; *o valor decimal é o que resulta das somas das potências de 2 correspondentes às posições dos '1's do número binário*.

Para este e para muitos outros efeitos convém por vezes ter presentes os valores das potências de 2. Vale a pena *decorar* estes valores pelo menos até 2^{10}.

2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024

Podemos assim imaginar um número binário numa grelha onde os '1's presentes têm o valor da potência de 2 correspondente à posição onde se encontram

Peso	512	256	128	64	32	16	8	4	2	1
	9	8	7	6	5	4	3	2	1	0

Posição (expoente)

Considerando uma representação em binário nesta grelha imaginária, o valor do número representado é a soma dos pesos dos '1's contidos nesse número.

Exemplo: Qual será o valor de $(101001100)_2$?

9	8	7	6	5	4	3	2	1	0
	1	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**

$= 2^8 + 2^6 + 2^3 + 2^2 = 256 + 64 + 8 + 4$

3.3.2 Conversão de decimal para binário – método das subtracções

O mesmo tipo de modelo poderia ser usado para converter de decimal para binário. Trata-se, afinal de contas, de colocar '1's na grelha fazendo com que a soma dos seus pesos forme o número pretendido.

Seja, por exemplo, o número 300. É claro que seria excessivo colocar um '1' na posição 9, que corresponde ao peso $2^9 = 512$, ou mais para esquerda – isso faria com que o número ficasse

maior que 512. Correcto é colocar um '1' na posição 8, cujo peso é $2^8 = 256$, tendo em vista formar um número maior que 256 (e menor que 512).

Tendo colocado o primeiro '1' na posição de valor 256, falta obter os '1's respeitantes ao resto do número, ou seja, 44 = 300 – 256. Aplicando o mesmo raciocínio, colocamos um '1' na posição 6, que vale $2^6 = 32$, ficando a sobrar 12 = 44 – 32. E assim sucessivamente.

Este procedimento de conversão de decimal para binário chama-se *método das subtracções*. Em suma, este processo consiste em:

- decompor o número da soma de uma potência de 2 com outro factor;
- colocar 1 na posição correspondente à potência de 2 obtida;
- repetir o procedimento para a diferença;

Exemplo: converter o número 300 para base 2

$300 = \mathbf{256} + 44 \Rightarrow 2^8$
$44 = \mathbf{32} + 12 \Rightarrow 2^5$
$12 = \mathbf{8} + 4 \Rightarrow 2^3$
$4 = \mathbf{4} + 0 \Rightarrow 2^2$

O resultado é $(100101100)_2$.

9	8	7	6	5	4	3	2	1	0
	1	0	0	**1**	0	**1**	**1**	0	0

3.3.3 Conversão de octal para binário

A base fundamental para a computação digital é obviamente a base 2. As outras bases de trabalho importantes são a base 8 (octal) e a base 16 (hexadecimal) – elas próprias potências naturais de 2.

Acontece que há uma relação muito simples entre a representação de um número em binário e a representação do mesmo número em octal ou hexadecimal (ou, em geral, numa outra base que também seja potência de 2). De certa forma a representação nestas bases pode ser vista como uma forma compacta de representação em binário e este é realmente o seu principal interesse.

A relação de octal para binário é de um para três dígitos. Para converter um número de octal para binário substitui-se cada dígito pela respectiva representação em binário, usando um grupo de três bits. Se se pretende o contrário, isto é, converter um número de binário em octal, substitui-se cada grupo de 3 bits por um dígito octal.

Para estas conversões convém *decorar* a seguinte tabela que lista os números binários de 0 a 7 (representados com três bits). Para tal basta ter presente a grelha de três bits, com os pesos associados às posições.

4	2	1

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Exemplos:

1	0	1	101 é 4 + 1 = 5

1	1	0	110 é 4 + 2 = 6

Usando a tabela podemos facilmente converter um número de octal para binário:

$(163)_8 = (001\ 110\ 011)_2 = (1110011)_2$
$(32.04)_8 = (011\ 010\ .\ 000\ 100)_2 = (\ 11010.0001)_2$

Igualmente podemos fazer as conversões contrárias, se necessário juntando '0's à esquerda da parte inteira ou à direita da parte fraccionária:

$(11001)_2 = (011\ 001)_2 = (31)_8$
$(11011.01101)_2 = (011\ 011.011\ 010)_2 = (33.32)_8$

3.3.4 Conversão de hexadecimal para binário

O interesse da base 16 (hexadecimal) é semelhante ao do octal. É uma representação compacta de binário em que a cada dígito hexadecimal correspondem 4 dígitos binários.

Para fazer a correspondência é conveniente *decorar* a seguinte tabela. Para tal basta ter presente a grelha de colocação, agora com 4 bits.

8	4	2	1

0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Exemplos:

1	0	0	1	1001 é 8 + 1 = 9
1	1	0	0	1100 é 8 + 4 = 12 = C
1	1	1	0	1110 é 8 + 4 + 2 = 14 = E

Com base nestas tabelas podemos facilmente converter números de hexadecimal para binário.
Exemplo:

$(3A.0E)_{16} = (0011\ 1010\ .\ 0000\ 1110)_2 = (111010.\ 0000111)_2$

3.3.5 Conversões entre octal e hexadecimal

Para converter rapidamente entre octal e hexadecimal o mais fácil é usar a conversão intermédia para binário.

Por exemplo, para converter um número de octal para hexadecimal pode:

1. converter para binário, fazendo corresponder 3 dígitos binário a cada dígito octal;
2. reagrupar em grupos de 4 dígitos, se necessário acrescentando 0 à esquerda da parte inteira ou à direita da parte fraccionária;
3. converter em hexadecimal, fazendo corresponder um dígito hexadecimal a cada grupo de 4 dígitos binários

Exemplo:

$(32.034)_8 = (011\ 010\ .000\ 011\ 100)_2 = (0001\ 1010\ .0000\ 1110)_2 = (1A.0E)_{16}$

3.4. Outras considerações

Num computador os bits aparecem organizados segundo grupos ou palavras. Por exemplo, quando se diz que um computador segue uma arquitectura de 32 bits é porque existem elementos fundamentais dessa máquina e operações nela realizadas que ocorrem em blocos de 32 bits.

Considerando por exemplo uma palavra de 4 bits. Com 4 bits é possível fazer 2^4 combinações binárias diferentes, de '0000' a '1111'. A primeira corresponde ao número 0 e a última corresponde ao número 15 – o maior número inteiro que é possível representar usando apenas 4 bits.

Em geral, com *N* bits:

- conseguem-se fazer 2^N combinações binárias diferentes;
- o maior número inteiro sem sinal representável é $2^N - 1$ (o número seguinte seria 2^N mas nesse caso já seriam necessários N+1 bits para o representar).

Um conjunto de 8 bits é vulgarmente designado por ***byte***. Com um byte conseguem-se $2^8 = 256$ combinações binárias diferentes. O maior número é '1111 1111' que vale 255 ('FF' em hexadecimal, ou '377' em octal).

O número 2^{10}, 1024, é vulgarmente designado por 1 Kilo (K). Muitas vezes relaciona-se 1K com 10 bits, na medida em que com 10 bits podem-se fazer as tais 2^{10} ou seja 1024 combinações.

Com 11 bits temos 2K combinações e com 12 bits temos 4K combinações, ou seja:

$$2^{11} = 2 \times 2^{10} = 2K$$

$$2^{12} = 2^2 \times 2^{10} = 4K$$

Da mesma forma relaciona-se 1 Mega (M) com 20 bits, Giga (G) com 30 bits e Tera (T) com 40 bits:

$1M = 2^{20}$;

$1G = 2^{30}$;

$1T = 2^{40}$;

4. Representação de números inteiros em binário

4.1. Magnitude e sinal

Toda a informação é representada no computador por '0's e '1's. No caso dos números inteiros, uma forma natural de representação é escrevê-los em binário. Por exemplo, suponha que temos um computador onde os números são representados em palavras de 8 bits. Usando uma destas palavras podemos representar números inteiros (positivos) de 0 a 255 em binário. Se tal não for suficiente, poderíamos usar duas palavras (8+8, ou seja 16 bits) para representar cada número; sendo assim, os números já poderiam ir de 0 até 2^{16}-1.

Por outro lado pode ser necessário representar números positivos e negativos. Nesse caso uma possível representação seria a seguinte: usar um bit para o sinal e os restantes para a magnitude (valor absoluto) do número. Seja por exemplo uma palavra com 8 bits. Podíamos usar o 8º bit para sinal, representando, por exemplo os positivos com 0 e os negativos com 1. Assim, por exemplo:

0000 1100 seria o +12

1000 1100 seria o –12

Os números representáveis seriam os que ficam entre $\pm(2^7 - 1)$ ou seja, de –127 a 127. Ou seja, o maior número (mais positivo) seria:

0111 1111 => 127

e o menor (mais negativo) seria

1111 1111 => –127

Usando este esquema de representação, o número '0' teria duas representações possíveis:

0000 0000 => digamos o +0

1000 0000 => digamos o –0.

4.2. Representação em complemento

Uma alternativa mais interessante ao modelo anterior é a chamada representação em complemento.

4.2.1 Complemento para 1

Definição: *o complemento para 1* de um número de *N* bits é a diferença do número para 2^N–1. Ou seja, dado um número *X* de *N* bits, o seu complemento para 1 é, por definição:

$2^N - 1 - X$.

Exemplo: calcular o complemento para 1 do número de 4 bits '0110'

Faz-se a diferença para $2^4 - 1$ ou seja, faz-se

1111 – 0110 =1001

```
  1 1 1 1
- 0 1 1 0
---------
  1 0 0 1
```

A natureza da operação, em que fazemos a diferença para um número composto só por '1's, sugere a seguinte regra prática: *para obter o complemento para 1 basta trocar cada um dos bits do número*.

4.2.2 Complemento para 2

Definição: o *complemento para 2* de um número de *N* bits é a diferença do número para 2^N. Ou seja, dado um número *X* de *N* bits o seu complemento para 2 é, por definição:

$2^N - X$.

Exemplo: calcular o complemento para 2 do número de 4 bits '0110'

```
  1 0 0 0 0
- 0 1 1 0
  ---------
    1 0 1 0
```

Faz-se a diferença para 2^4 ou seja, faz-se

10000 – 0110 = 1010

Existem também regras práticas para obter o complemento para 2 de um número representado em binário, ligeiramente mais complicadas:

- Como se pode verificar facilmente, o complemento para 2 é igual ao complemento para 1 mais '1'. Pode-se por isso aplicar a seguinte regra:
  - trocar todos os bits (complemento para 1)…
  - … e depois somar '1'

- Em alternativa, para calcular o complemento para 2, procede-se da direita para a esquerda e
  - mantém-se o número original até aparecer o primeiro '1' (inclusive)…
  - …e a partir daí troca-se cada um dos bits;

Na maioria dos casos, os números com sinal são representados em complemento para 2.

Vamos exemplificar com 4 bits. Os números positivos são representados da maneira habitual. Para os negativos é feito o complemento para 2. Desta forma obteremos a representação indicada na tabela ao lado

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

Exemplo: o –5 é representado pelo complemento para 2 de 5, ou seja

0101 => 1011

4.3. Operações aritméticas em complemento para 2

O interesse desta representação é que ela permite um tratamento muito uniforme dos números negativos.

Imagine, por exemplo, que se pretende fazer a soma de dois números com sinal. Dependendo do sinal, a soma pode ser na realidade uma subtracção. Por exemplo, fazer 7 – 3 é o mesmo que fazer 7 + (–3). A representação em complemento para dois resolve a questão de forma simples: basta somar.

Exemplo: calcular 7 + (–3) utilizando notação em complemento para 2.

```
    0 1 1 1  <-   7
+   1 1 0 1  <-  -3
  1 0 1 0 0  <-   4
```

Somando e *descartando o bit de excesso* obtém-se 4, o resultado correcto desta operação.

Este comportamento acontece noutros casos:

Exemplo: calcular 5 – 7 utilizando notação em complemento para 2.

5 – 7 é o mesmo que 5 + (–7)

```
    0 1 0 1  <-   5
+   1 0 0 1  <-  -7
  0 1 1 1 0  <-  -2
```

O resultado é –2 (que também está representado em complemento para 2)

As operações anteriores justificam-se também algebricamente, sendo importante nalguns casos a questão de descartar o bit de excesso – em particular nos casos em que esse bit é 1.

Considere a diferença entre dois números A e B. *Em vez da diferença $A - B$, pode-se fazer a soma com o complemento para 2 do número B*, ou seja, $A + (2^N - B) = A - B + 2^N$, que é *quase* o mesmo resultado. Nesta situação, ao descartar o bit de excesso estamos justamente a subtrair 2^N. Assim, obtemos $A + (2^N - B) - 2^N = A - B$.

```
    0 1 1 1  <-   7    A
+   1 0 1 1  <-  -5    2^4-B
  1 0 0 1 0  <-   2    A+(2^4-B)-2^4   (2^4 seria o valor do bit de excesso descartado)
```

Fica como exercício validar o mesmo raciocínio noutras circunstâncias. Por exemplo:

3 – (–2) = 5 faz-se a soma com o complemento p/ 2 de '–2' ou seja:
$A + (2^N - (2^N - B)) = A + B$

3 – 5 = –2 faz-se a soma com o complemento p/2 de '5', ou seja:
$A + (2^N - B) = 2^N - (A - B)$
obtendo o resultado em complemento;

–2 – 3= –5 faz-se a soma do complemento de ambos, ou seja fazemos
$(2^N - A) + (2^N - B)$ e tira-se 2^N (por haver excesso)
obtendo o resultado em complemento;

4.4. Overflow

Seja como for, a capacidade de representação dos números em binário é limitada. Isso quer dizer que podemos fazer operações comuns (designadamente somas) de números válidos que dariam como resultado números já não representáveis.

Por exemplo, se tivermos 8 bits e números sem sinal o maior número representável é 255. Somando 140 + 140 obteríamos 280, que já não é representável com apenas 8 bits (seriam necessários 9 bits). Nesta situação o resultado efectivo da operação (em 8 bits) seria 24.

```
    1  0  0  0  1  1  0  0
+   1  0  0  0  1  1  0  0
---------------------------
 1  0  0  0  1  1  0  0  0
```

Nestas circunstâncias em que operando dois números válidos se obtém um número não representável diz-se que estamos numa situação de ***overflow***. Para o caso de números inteiros sem sinal, a situação é fácil de reconhecer: ocorre *overflow* se o bit de excesso der 1.

Tratando-se de números com sinal, o *overflow* pode ocorrer em duas situações: fazendo uma operação que daria um número positivo maior do que o maior número positivo representável; ou fazendo operação que daria um número negativo menor do que o menor número positivo representável.

Consideremos novamente o caso de números inteiros com sinal representados com 4 bits. No seguinte exemplo pode-se observar a situação em que a soma de números positivos daria um resultado maior do que 7 (e que seria interpretado como número negativo!).

```
    0 1 0 1  <-   5
+   0 1 0 1  <-   4
-------------------
  0 1 1 1 0  <-  -2
```

Outro exemplo ilustra a soma de dois números negativos cujo resultado seria menor que –8 (e que nesta situação seria interpretado como número positivo!).

```
    1 0 1 1  <-  -5
+   1 0 1 0  <-  -6
-------------------
  1 0 1 0 1  <-   5
```

5. Representação de caracteres

Nos computadores digitais o texto é também, naturalmente, representado em bits. A estratégia neste caso é codificar cada um dos caracteres que podem compor o texto, ou seja, representar cada carácter por uma sequência de bits única e distinta.

5.1. Código ASCII

A codificação clássica para este efeito é a que consta da tabela designada por código **ASCII** [(1)]. O código ASCII original utiliza 7 bits para representar cada carácter.

Genericamente, um texto é uma sequência de caracteres. Por exemplo, o texto representado na figura é constituído pela seguinte sequência de caracteres:

```
Ola
Mundo,
bla, bla
```

letra *O*
letra *l*
letra *a*
um carácter de mudança de linha
letra *M*
letra *u*
…

Usando o código ASCII cada um destes caracteres seria representado por 7 bits. De acordo com a tabela 1 este texto poderia ser representado pela seguinte sequência de bits:

100 1111
110 1100
110 0001
000 1010
110 1101
...

5.1.1 Particularidades importantes

O código ASCII tem algumas características importantes que interessa saber e que aliás têm um papel essencial no processamento de caracteres.

Note para começar que os primeiros caracteres são especiais – correspondem a caracteres de controlo, sem representação visível. O termo de *controlo* significa que o dispositivo que trata o carácter o interpreta como um comando e não como um símbolo a escrever. Por exemplo o carácter com código 7 (000 0111) é o sinal *bell* (campainha); quando recebido por um terminal deverá originar um som. Os caracteres 10 e 13 são usados para controlar as mudanças de linha. Por exemplo, uma impressora que receba um carácter CR (000 1101) deve voltar ao início da linha e ao receber o carácter LF (000 1010) deve mudar de linha. Muitos destes sinais de controlo estão actualmente em desuso.

(1) ASCII – *American Standard Code for Information Interchange*.

$B_3B_2B_1B_0$	$B_6B_5B_4$							
	000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NULL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	“	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	´	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Tabela 1 – Código ASCII (B_6 é o bit mais significativo).

Estes mesmos caracteres são, em muitas circunstâncias, usados nos ficheiros de texto para assinalar as mudanças de linha. Os detalhes dependem do editor de texto e do sistema operativo. Por exemplo, num comum ficheiro de texto em Unix a mudança de linha é assinalada pelo carácter 10 – também conhecido por ‘\n’ que é a sua representação simbólica em algumas linguagens de programação.

É importante notar que no código ASCII *os grupos de algarismos, letras maiúsculas e letras minúsculas têm códigos sequenciais*. O código ASCII dos algarismos vai do *0* ao *9* por ordem numérica. O código ASCII das letras minúsculas vai de *a* ao *z* por ordem numérica. O código ASCII das letras maiúsculas vai do *A* ao *Z* por ordem numérica.

Esta circunstância está na base dos processos de ordenação dos caracteres (designadamente da ordenação alfabética) e da aritmética de caracteres. Por exemplo, podemos dizer que a letra *B* é maior que a letra *A*, uma vez que o código ASCII de *B* tem um valor maior que o código ASCII de *A*.

Em muitas circunstâncias representa-se o código de um carácter escrevendo o carácter entre pelicas (‘).

Por exemplo

‘a’ representa o código ASCII da letra *a*

‘5’ representa o código ASCII do algarismo *5*

‘+’ representa o código ASCII do sinal +

Usando esta notação, podemos assim dizer que

‘B’ > ‘A’

Os mesmos princípios permitem fazer operações aritméticas com caracteres – na realidade são operações com o código mas que têm significado lógico em termos dos caracteres envolvidos.

Assim, por exemplo, sabendo que o código ASCII da letra *B* é 100 0010 então o da letra *C* é 100 0011 e o da letra *F* é 100 0110. Quer dizer:

'C' = 'B' + 1

'F' = 'C' + 3

Outro exemplo: pode obter o código ASCII da letra *f* somando ao código ASCII da letra *a* a diferença entre os códigos ASCII de *F* e de *A*. Ou seja:

'f' = 'a' + ('F' – 'A')

Muitas vezes é necessário converter os caracteres que representam algarismos no valor numérico correspondente. O valor numérico de um carácter obtém-se fazendo a diferença para o código do algarismo *0* Por exemplo

'3' – '0' = 3

ou seja, a diferença entre o código ASCII do algarismo *3* e o do algarismo *0* é o valor numérico 3. Repare ainda que, no código ASCII, os 4 bits menos significativos representam o número binário correspondente ao algarismo. Por exemplo, os últimos 4 bits do código ASCII do algarismo *7* são

$B_3B_2B_1B_0 = 0111$

e representam o número binário 7.

5.1.2 Paridade e extensão ao código ASCII – utilização do 8º bit

Normalmente os computadores usam palavras de 8 bits ou múltiplos de 8 bits. Como o código ASCII original é um código de 7 bits, "sobra" um bit que pode ser usado para diferentes finalidades.

A primeira dessas finalidades é a utilização como *bit de paridade*. Imagine que está a transmitir texto sobre uma linha com ruído. Pode acontecer que a transmissão adultere alguns dos bits transmitidos, levando o receptor a obter alguns caracteres errados. Uma das formas de minimizar este problema é usar o 8º bit como bit de paridade. Assim, por exemplo, ao mandar um carácter cujo código ASCII tenha um número par de '1's o 8º bit irá a '0' e no caso contrário irá a '1'. Nestas condições, o receptor pode verificar se a sequência recebida está de acordo com o bit de paridade. Podem-se assim detectar eventuais erros de transmissão e pedir ao emissor para voltar enviar a informação, nesses casos.

Exemplo: o mesmo texto com bit de paridade

1 100 1111

0 110 1100

1 110 0001

...

Outra possível utilização do 8º bit é a extensão do próprio código ASCII. Usando o 8º bit podem-se acrescentar mais 128 caracteres ao código original, ficando assim o código com um

total de 256 caracteres. Esta possibilidade foi muito usada para introduzir caracteres específicos de cada linguagem (por exemplo, o *c* cedilhado – *ç* – e as vogais acentuadas – à, á, é, … –, no caso Português), originando diferentes extensões do código ASCII.

5.2. Código Unicode

O **Unicode** é um código de representação de caracteres cuja primeira versão foi normalizada em 1991. A principal motivação para o aparecimento deste código foi a possibilidade de poder representar todos os caracteres utilizados nas várias linguagens que existem no mundo. Inicialmente, o comprimento das palavras do código Unicode era de 16 bits. Conseguia-se portanto a possibilidade de representar até $2^{16} = 65536$ caracteres distintos.

Na altura pensava-se que essas 2^{16} combinações binárias eram mais do que suficientes, dado que se estima que número total de caracteres distintos usados em jornais e revistas de todos os países do mundo é inferior a 20.000. No entanto, em versões mais recentes do código, e por razões históricas, começaram-se também a introduzir caracteres pertencentes a linguagens que já não são utilizadas (ex: fenício, escrita cuneiforme) e símbolos utilizados em documentos antigos. Para além disso foram também introduzidos símbolos utilizados em diferentes áreas, tais como símbolos musicais, alfabeto Braille, símbolos matemáticos, símbolos usados em jogos (mahjong, dominó), etc.

Em suma, depressa se chegou à conclusão que os 16 bits do Unicode original já não seriam suficientes. Actualmente o Unicode é um código de 32 bits em que as palavras podem ter comprimento variável, organizadas segundo sequências de 8, 16, ou 32 bits, dando origem às formas de codificação UTF-8, UTF-16 e UTF-32.

Isto quer dizer que com os 32 bits do código Unicode actual seria possível representar 2^{32}, ou seja, 4G = 4.294.967.296 (!), símbolos diferentes [(2)]. Na realidade, na última versão da norma até à data (versão 5.1.0) estão representados "apenas" 100.713 símbolos diferentes.

A forma UTF-8 assegura a compatibilidade com o código ASCII e é provavelmente a mais popular em aplicações relacionadas com internet. As palavras de código podem ter comprimento variável, *i.e.*, para além das palavras formadas por uma única sequência de 8 bits (1 byte), podem também aparecer palavras compostas por 2 a 4 sequências de 8 bits (2 a 4 bytes).

No caso de UTF-16, as palavras de código são compostas por uma ou duas sequências de 16 bits e no caso de UTF-32 todas as palavras de código são sequências de 32 bits. À medida que se aumenta o comprimento das sequências a descodificar, aumenta a eficiência da descodificação (é mais rápida) mas é necessário ter mais memória (espaço) para guardar as palavras de código. Isto quer dizer que o UTF-8 é uma forma mais compacta, mas com descodificação mais ineficiente. No outro extremo está o UTF-32, com uma descodificação mais eficiente, mas que requer mais memória.

[(2)] Na realidade as 4G combinações não podem ser todas utilizadas, pois a norma define um conjunto de combinações que têm tratamento especial e não podem ser utilizadas para representar caracteres.

Lista de Revisões

Versão	Autor	Data	Comentários
001	JRG	Jan./2005	Versão draft. Publicação antecipada para o período de avaliações 2004/2005
002	TB	Out./2008	Correcção de gralhas; reformatação do texto; reformulação pontual de texto; acrescentada secção sobre código 'unicode';
003	TB, LC	Set./2009	Actualização do logo ISCTE; correcção de gralhas.
003a	TB, JPO	Set./2009	Correcção de algumas gralhas.
003b	TB	Set./2011	Correcção de gralha na secção 3.2