2 Representação da informação na memória

A informação básica em sistemas digitais é o *bit*, que pode assumir apenas dois valores: 0 ou 1. Fisicamente um bit é um circuito elétrico – não iremos entrar em mais detalhes neste livro – que pode ser implementado em um espaço diminuto: com a tecnologia atual, cerca de 25×10^{-3} mm^2.

A memória do computador é composta de um número enorme destes circuitos de bits, organizados em *palavras*. Uma palavra é um conjunto de tamanho fixo de bits. Por exemplo, a figura abaixo mostra a representação esquemática de uma memória de 8 palavras, onde cada palavra possui 16 bits:

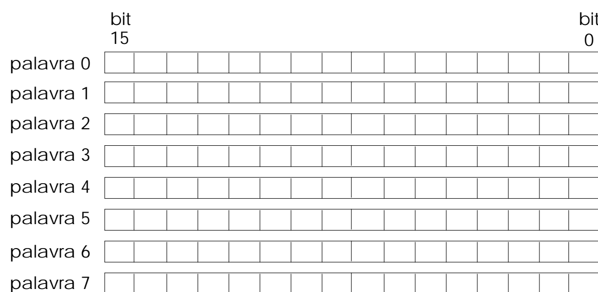


Fig. 2.1 Esquema de uma memória com 8 palavras de 16 bits

Neste livro usaremos a convenção de numerar os bits de uma palavra da direita para a

esquerda, começando com o bit número 0.

A divisão da memória do computador em palavras é apenas uma forma de possibilitar o acesso a um conjunto específico de bits. A memória pode ser vista também como um vetor de palavras. Note que o esquema da Fig. 1.1 já sugere isto: os números à direita de cada palavra são os índices de cada palavra no "vetor" da memória. O índice, ou posição, de uma palavra na memória é usualmente chamado de *endereço* da palavra de memória.

Computadores diferentes podem usar palavras de tamanho diferentes. Os primeiros computadores pessoais utilizavam palavras de 8 bits (uma palavra de 8 bits é chamada de *byte*). Computadores de maior porte e estações de trabalho hoje em dia utilizam palavras de 64 bits.

A quantidade de bits da memória é tradicionalmente medida em bytes. Mais precisamente, são utilizadas as abreviações KB (Kilobytes, ou seja, 2^{10}=1024 bytes), MB (Megabytes, ou 2^{20} bytes) e GB (Gigabytes, ou 2^{30} bytes). Os primeiros computadores pessoais possuíam algumas centenas de milhares de bytes de memória: o IBM XT por exemplo possuía tipicamente 640 Kilobytes. Estações de trabalho modernas possuem da ordem de dezenas ou centenas de megabytes de memória.

Como veremos no Capítulo 3, a memória é um componente passivo do computador. O componente ativo é a unidade de processamento central, ou simplesmente processador. O processador acessa a memória para leitura e para escrita, utilizando-a armazenar as informações que devem ser processadas.

O primeiro grande avanço da tecnologia de programação foi a percepção de que o *programa*, ou seja, a seqüência de instruções que devem ser executadas pelo processador poderia também ser armazenada na memória, da mesma forma que os dados (isto foi descoberto por ?? em ?, muito antes portanto do advento dos computadores eletrônicos). Esta descoberta indicou que um mesmo computador poderia executar computações diferentes, desde que fosse assim programado.

A interpretação das seqüências de 0's e 1's presente na memória é responsabilidade exclusiva do programador: o processador não tem meios de determinar se uma dada seqüência representa por exemplo um caractere, um valor inteiro ou um valor real. O programador é responsável por garantir que o processador interprete o conteúdo da memória corretamente. Neste capítulo vamos estudar como se faz a representação de caracteres e números inteiros em computadores; a representação de números reais será vista em outro capítulo.

2.1 Representação de inteiros

Como toda informação é armazenada na forma de 0's e 1's, é natural usar uma representação *binária* para armazenar números inteiros. Para examinar o conteúdo da memória poderíamos usar o sistema *decimal*, que nos é bastante familiar: leríamos a seqüência de 0's e 1's e a transformaríamos em um valor decimal. Esse procedimento, no entanto, é muito sujeito a erros, pela quantidade de cálculos que teríamos que efetuar. É muito melhor utilizar a representação binária diretamente. Entretanto às vezes isto também se torna inconveniente, principalmente quando manipulamos números grandes, devido à quantidade de dígitos necessária para representá-los em binário. O sistema *hexadecimal* é um bom intermediário entre os sistemas decimal e binário: é uma representação mais compacta que a decimal, e a conversão entre binário e hexadecimal é muito mais fácil do que a entre binário e decimal. Nesta seção vamos estudar estes dois sistemas de representação de números, binário e hexadecimal.

2.1.1 Notação Posicional

Os sistemas binário e hexadecimal, assim como o sistema decimal, são baseados na notação *posicional*. Isto quer dizer que o valor representado por um dígito depende apenas da posição que o dígito ocupa em um número. Por exemplo, o dígito 7 representa setenta tanto em 73 quanto em 6375, por ser o segundo da direita para a esquerda em ambos os números. O sistema romano de representação de números é um exemplo de notação que não é posicional: o dígito 'X' vale dez tanto em XIII (onde é o primeiro a partir da esquerda e o quarto a partir da direita) quanto em CCXV (onde é o terceiro a partir da esquerda e o segundo a partir da direita).

O valor de um dígito em uma representação posicional é o valor do dígito multiplicado por uma potência da base do sistema sendo usado. Se o número tem, em uma dada base b, a representação

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

onde cada d_i representa um dígito, então d_i representa o valor do dígito multiplicado pela base elevada à i-ésima potência. O dígito mais à esquerda, por representar a parcela de maior valor, é usualmente chamado de dígito mais significativo. Correspondentemente, o dígito mais à direita é chamado de dígito menos significativo.

O valor, em decimal, do número acima é

$$(\partial_n \times b^n) + (\partial_{n-1} \times b^{n-1}) + \ldots + (\partial_2 \times b^2) + (\partial_1 \times b^1) + \partial_0 \qquad (2.1)$$

onde ∂_i é o valor decimal do dígito d_i na base 10.

O sistema binário é o sistema de notação posicional com base 2. Como sabemos, um dígito binário é também chamado de um *bit*. O sistema hexadecimal (que abreviaremos por hexa daqui em diante) é o sistema de notação posicional com base 16.

Para uma determinada base b, necessitamos símbolos que identifiquem b valores de dígitos (de 0 a $b - 1$). Para qualquer base $b \leq 10$, podemos usar os dígitos usuais de 0 a $b - 1$. Por exemplo, para base 8, usaríamos 0, 1, 2, 3, 4, 5, 6 e 7 para representar os oito dígitos. Para bases maiores que 10, é necessário acrescentar alguns símbolos aos usuais. Para base 16, em particular, a convenção é utilizar os símbolos 0, 1, 2,...8, 9, A, B, C, D, E e F, onde os caracteres de A a F representam os dígitos de dez a quinze, respectivamente.

2.1.2 Conversão entre binário e hexa

O número de valores distintos que podemos representar usando um número fixo de dígitos t em uma determinada base b é b^t: cada dígito t pode tomar qualquer um dos b valores entre 0 e $b - 1$. Assim, se os valores a serem representados são inteiros positivos, podemos representar os números no intervalo $[0, b^t - 1]$. Por exemplo, com 4 dígitos na base 2 podemos representar os números no intervalo [0, 15]. Note que este exemplo ilustra uma correspondência interessante entre as representações binária e hexa: quatro dígitos binários tem o mesmo poder de representação de um dígito hexa (16 valores distintos). A tabela 2.1 mostra as representações dos números de 0 a 15 em binário e hexa.

Como cada número binário de quatro bits corresponde a um dígito hexa (e vice-versa), podemos usar a representação hexa como uma forma abreviada da representação binária, convertendo cada quatro bits para a representação hexa equivalente.

Exemplo 2.1: converter 111101111000_2 para hexa

```
1111 0111 1000
 \/   \/   \/
 F    7    8
```

Resultado: $F78_{16}$

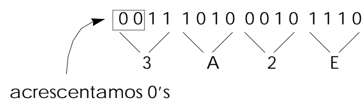
Como o número de bits pode não ser múltiplo de quatro, começamos a agrupar os bits da direita para a esquerda. Se o último grupo não possuir quatro bits, simplesmente completamos o que falta com zeros (pois zeros à esquerda, ao contrário de zeros

Tabela 2.1 Representação binária e hexa de números de 0 a 15.

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

à direita, não influem no valor do número em notação posicional).

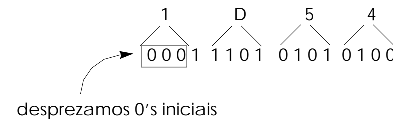
Exemplo 2.2: converter 11101000101110_2 para hexa



Resultado: $3A2E_{16}$

Para converter hexa em binário, usamos o procedimento inverso.

Exemplo 2.3: converter $1D54_{16}$ para binário



Resultado: 1110101010100_2

É fácil compreender porque este método simples de conversão entre as representações binária e hexadecimal funciona. O dígito 9 em $19FA5_{16}$ representa 9×16^3. Os bits correspondentes em binário são 1001, e estão nas casas de 2^{12} a 2^{15}. Portanto, o valor que esses bits representam pode ser calculado por

$$1 \times 2^{15} + 0 \times 2^{14} + 0 \times 2^{13} + 1 \times 2^{12} = (1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) \times 2^{12} =$$

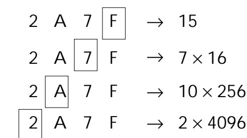
$$= 9 \times 2^{12} = 9 \times (2^4)^3 = 9 \times 16^3$$

Ou seja, o valor representado pelos quatro dígitos binários é exatamente o mesmo que o representado pelo dígito hexa.

2.1.3 Conversão de hexa e binário para decimal

A fórmula (2.1) pode ser utilizada diretamente para converter para a base decimal números representados em outras bases. É conveniente memorizar as primeiras potências de 16 para agilizar o processo de conversão entre hexa e decimal: 16, 256, 4096, 65536...

Exemplo 2.4: converter $2A7F_{16}$ para decimal.



$2A7F_{16} = 15 + 7 \times 16 + 10 \times 256 + 2 \times 4096 = 10909_{10}$

Resultado: 10909_{10}

Note que é mais fácil inverter a posição dos fatores em relação à fórmula original, e co-

meçar o cálculo a partir do dígito menos significativo.

A conversão de números binários em decimais se faz de forma similar, mas como o multiplicador é sempre 0 ou 1, basta fazer a soma das potências de 2 para as posições que têm dígito 1.

Exemplo 2.5: converter 1011001_2 para decimal.

```
1 0 1 1 0 0 [1] → 2^0
1 0 1 [1] 0 0 1 → 2^3
1 0 [1] 1 0 0 1 → 2^4
[1] 0 1 1 0 0 1 → 2^6
```

$1011001_2 = 1 + 2^3 + 2^4 + 2^6 = 1 + 8 + 16 + 64 = 89_{10}$

Resultado: 89_{10}

Uma outra maneira de converter binário em decimal é fazer primeiro a conversão de binário para hexadecimal, usando o método de substituição direta, e então converter de hexa para decimal.

2.1.4 Conversão de decimal para hexa e binário

Já que a conversão de números em outras bases para base decimal envolve multiplicação, é de se esperar que a conversão de números na base decimal para outras bases envolva divisão. Vamos retomar a equação 2.1, que calcula o valor de um número de $n + 1$ dígitos em uma base b qualquer:

$$(\partial_n \times b^n) + (\partial_{n-1} \times b^{n-1}) + \ldots + (\partial_2 \times b^2) + (\partial_1 \times b^1) + \partial_0$$

O único termo que não é divisível pela base b é ∂_0, o dígito mais à direita na representação do número, e que vale entre 0 e $b - 1$. Ou seja, ∂_0 é o resto da divisão do número pela base b. Assim, partindo de um número decimal, podemos determinar o dígito hexa mais à direita na representação desse número na base hexadecimal tomando o resto de sua divisão por 16 (note que a aritmética pode ser feita toda em decimal). O quociente dessa divisão é

$$(\partial_n \times b^{n-1}) + (\partial_{n-1} \times b^{n-2}) + \ldots + (\partial_3 \times b^2) + (\partial_2 \times b^1) + \partial_1$$

Novamente, cada termo é divisível por b exceto ∂_1. Se dividirmos este valor por b, ∂_1 será o resto da divisão (e portanto o valor do segundo dígito da representação na

base b) e o novo quociente será

$$(\partial_n \times b^{n-2}) + (\partial_{n-1} \times b^{n-3}) + \ldots + (\partial_4 \times b^2) + (\partial_3 \times b^1) + \partial_2$$

O processo continua até que o quociente seja zero.

Exemplo 2.6: converter 2223_{10} para hexa

```
2223 | 16
 62    138
 143
 [15]
d0 = F
            138 | 16
           [10]   8
           d1 = A
                       8 | 16
                      [8]  0
                      d2 = 8
```

Resultado: $8AF_{16}$

A conversão de decimal para binário pode ser feita pelo mesmo método acima. Uma outra maneira, que envolve um número menor de divisões, é primeiro converter o número decimal para hexa e então fazer a substituição de cada dígito pela correspondente representação binária.

2.1.5 Aritmética em hexa

Em vários dos exercícios deste livro teremos que somar e subtrair números em hexa. Ao invés de converter os números para decimal, efetuar a operação, e fazer a conversão do resultado para hexa, é conveniente fazer a operação diretamente em hexa. Aritmética em hexa não é muito diferente da decimal, pois o princípio é o mesmo. Basta lembrar que estamos em uma base diferente: na soma, o vai-um só ocorre quando a soma de uma coluna de dígitos excede 15, e não 9. Na subtração, quando tomamos 1 emprestado de uma coluna mais à esquerda, é preciso lembrar que esse empréstimo representa 16 e não 10.

Exemplo 2.7: Somar $4ED3_{16}$ e $1A69_{16}$

① 4 E D 3 + 1 A 6 9 — $3 + 9 = 12 = C_{16}$

② 4 E D 3 + 1 A 6 9 = C — $(D + 6)_{16} = 13 + 6 =$ $= 19 = 16 + 3$, escrevemos 3 e vai-um

③ 1 / 4 E D 3 + 1 A 6 9 = 3 C — $(1 + E + A)_{16} = 1 + 14 + 10$ $= 25 = 16 + 9$, escrevemos 9 e vai-um

④ 1 / 4 E D 3 + 1 A 6 9 = 9 3 C — $1 + 4 + 1 = 6$

⑤ 4 E D 3 + 1 A 6 9 = 6 9 3 C

Resultado: $693C_{16}$

Exemplo 2.8: Subtrair $4A87_{16}$ de $83B5_{16}$

① 8 3 B 5 − 4 A 8 7 — 5 é menor que 7, temos que tomar um emprestado da próxima casa, e a coluna da unidade fica valendo 15_{16}

② 15 / 8 3 B 5 − 4 A 8_1 7 — $15_{16} - 7 = 16 + 5 - 7$ $= 14 = E_{16}$

③ 8 3 B 5 − 4 A 8_1 7 = E — $B - 9 = 2$

④ 8 3 B 5 − 4 A 8 7 = 2 E — 3 é menor que A, tomamos um emprestado

⑤ 13 / 8 3 B 5 − 4_1 A 8 7 = 2 E — $13_{16} - A_{16} = 16 + 3 - 10 = 9$

⑥ 8 3 B 5 − 4_1 A 8 7 = 9 2 E — $8 - 5 = 3$

⑦ 8 3 B 5 − 4 A 8 7 = 3 9 2 E

Resultado: $392E_{16}$

Um caso comum é a necessidade de multiplicação de um número hexa por uma potência de 2 como 2, 4, ou 8. Neste caso, é mais fácil fazer primeiro a conversão para binário e efetuar a operação em binário, pois para multiplicar um número na representação binária por 2 basta acrescentar um zero à direita. Note que para multiplicar um número hexa por 16 também basta acrescentar um zero à direita da representação

desse número em hexa.

2.2 Representação de inteiros com sinal

Como vimos, um número com t dígitos na base b pode representar b^t valores distintos. Se quisermos representar somente números naturais (inteiros positivos), podemos representar os números de 0 a $b^t - 1$. Por exemplo, para um número de 16 bits, podemos representar naturais de 0 a $2^{16} - 1$, ou seja, de 0 a 65535.

No entanto, normalmente não queremos representar somente números positivos, mas também negativos. Vamos considerar três métodos de representação de números inteiros com sinal em binário: sinal e magnitude, complemento-de-um e complemento-de-dois. Os três métodos têm em comum o fato de reservar um bit para indicar que o valor representado é negativo. Nos três métodos, o bit reservado é o mais à esquerda, ou seja, o bit mais significativo. Este bit é usualmente chamado bit de sinal, e, por convenção, o bit de sinal igual a 1 significa que o valor representado é negativo. Nos exemplos a seguir, consideraremos sempre números com 16 bits.

2.2.1 Sinal e magnitude

Um método bastante simples para representar inteiros positivos e negativos é utilizar o bit mais à esquerda para indicar o sinal e os restantes para representar o valor absoluto do inteiro em binário. Por usar bits distintos para a representação do sinal e para a representação do valor absoluto, este método é chamado de sinal e magnitude.

Exemplo 2.9: Representar 5000_{10} e -5000_{10} em sinal e magnitude

Convertendo para binário:

$$5000 = 4096 + 4 = 1004_{16} = 0001000000000100_2$$

Para achar o negativo, complementamos o bit de sinal:

0001000000000100

1001000000000100

$$-5000 = 1001000000000100_2 = 9004_{16}$$

A operação de negação na representação sinal e magnitude é simples: basta inverter o bit de sinal. No entanto, esta simplicidade tem um custo: o valor zero tem duas representações: +0 (bit de sinal zero e magnitude zero) e −0 (bit de sinal um e magnitude zero). A tabela 2.2 mostra a distribuição das seqüência de bits para números que po-

dem ser representados em sinal e magnitude utilizando-se 16 bits.

Tabela 2.2 Números inteiros que podem ser representados com 16 bits em sinal e magnitude

binário	*hexa*	*decimal*
0111111111111111	7FFF	32767
0111111111111110	7FFE	32766
...	...	...
0000000000000010	0002	2
0000000000000001	0001	1
0000000000000000	0000	0
1000000000000000	8000	-0
1000000000000001	8001	-1
1000000000000010	8002	-2
...	...	...
1111111111111110	FFFE	–32766
1111111111111111	FFFF	–32767

Para fazer uma operação de adição entre dois números na representação sinal e magnitude é necessário observar o sinal dos operandos. Se os dois tem o mesmo sinal, somamos os valores absolutos dos operandos e conservamos o sinal. Se os operandos tem sinais diferentes, subtraímos o maior valor absoluto do menor e usamos para o resultado o sinal do operando de maior valor absoluto.

Portanto, apesar de a negação na representação sinal e magnitude ser simples, a operação de adição envolve um procedimento razoavelmente complicado. Devem ser feitos testes e decisões devem ser tomadas que tornam difíceis a execução, pelo processador, de milhões de adições por segundo. A seguir veremos dois métodos que são uma escolha melhor para serem implementados por computador, por permitirem procedimentos de adição mais simples.

2.2.2 Complemento-de-um

Na notação complemento-de-um, o negativo de um número é obtido complementando-se (invertendo-se) todos os bits da representação binária de seu valor absoluto.

Exemplo 2.10: Representar 5000 e –5000 em complemento-de-um

A tabela 2.3 mostra a correspondência entre seqüências de bits e o valor representado em complemento-de-um, para números de 16 bits. Note que o bit de sinal de um nú-

Convertendo para binário:

$$5000 = 4096 + 4 = 1004_{16} = 0001000000000100_2$$

Para achar o negativo, complementamos todos os bits:

0001000000000100 →
1110111111111011

$$-5000 = 11101111111110112 = EFFB_{16}$$

mero negativo é também um, conforme a convenção. Note também que é fácil complementar um número em notação hexa: complementar um bit é o mesmo que subtrair de 1. Em hexa, é o mesmo que subtrair de 15.

Tabela 2.3 Números inteiros que podem ser representados com 16 bits em complemento-de-um.

binário	*hexa*	*decimal*
0111111111111111	7FFF	32767
0111111111111110	7FFE	32766
...	...	...
0000000000000010	0002	2
0000000000000001	0001	1
0000000000000000	0000	0
1111111111111111	FFFF	0
1111111111111110	FFFE	-1
1111111111111101	FFFD	-2
...	...	...
1000000000000001	8001	–32766
1000000000000000	8000	–32767

A adição de dois números é feita como se os operandos fossem números sem sinal, ou seja, tratando o bit de sinal como os outros, mas se houver vai-um do bit mais à esquerda, somamos um ao resultado.

Exemplo 2.11: Adição em complemento-de-um

$(-23422_{10}) + 22171_{10}$

```
 A 4 8 1 +
 5 6 9 B
 -------
 F B 1 C
```

Não houve vai-um, o resultado é FB1C

Conferindo: $FB1C_{16} = -1251_{10}$

$3672_{10} + (-1624_{10})$

```
   0 E 5 8 +
   F 9 A 7
   -------
 1 0 7 F F
```

Houve vai-um, o resultado é 07FF + 1 = 800

Conferindo: $800_{16} = 2048_{10}$

A adição na notação complemento-de-um é mais fácil do que na notação sinal e magnitude, pois não é necessário tomar decisões baseadas no sinal de cada operando. No entanto, a soma do bit de vai-um ao resultado é um passo a mais que, como veremos, pode ser evitado.

Tal como acontece com a notação sinal e magnitude, a notação complemento-de-um também tem o problema da duplicidade da representação do zero, o que pode ser muito incoveniente. Quando se deseja saber se o resultado de uma operação é zero, deve-se comparar com +0 ou –0? Apesar desse problema, alguns computadores utilizaram a notação complemento-de-um, detectando resultados –0 e transformando-os para +0. No entanto, os computadores atuais utilizam a notação complemento-de-dois, que elimina a duplicidade do zero.

2.2.3 Complemento-de-dois

A duplicidade do zero ocorre nas notações sinal e magnitude e complemento-de-um porque em ambas uma representação com o bit de sinal igual a um não é utilizada para representar um valor negativo, mas sim desperdiçada para representar –0. Na notação complemento-de-dois, todas as representações com o bit de sinal igual a um são usadas para representar um valor negativo. A definição da distribuição das seqüências de bits para os valores negativos é feita de forma a facilitar a operação de adição, como

veremos adiante. A tabela 2.4 mostra essa distribuição.

Tabela 2.4 Números inteiros que podem ser representados com 16 bits em complemento-de-dois.

binário	*hexa*	*decimal*
0111111111111111	7FFF	32767
0111111111111110	7FFE	32766
...	...	...
0000000000000010	0002	2
0000000000000001	0001	1
0000000000000000	0000	0
1111111111111111	FFFF	-1
1111111111111110	FFFE	-2
1111111111111101	FFFD	-3
...	...	...
1000000000000001	8001	–32767
1000000000000000	8000	–32768

Compare as distribuições das seqüências de bits nas representações complemento-de-um (tabela 2.3) e complemento-de-dois (tabela 2.4). Note que em complemento-de-dois o valor absoluto do maior número positivo é menor que o valor absoluto do menor número negativo, já que complemento-de-dois é capaz de representar um valor negativo a mais. Note também que para determinar a representação de um número negativo, nós partimos do seu valor absoluto em binário, complementamos todos os bits e somamos 1 ao resultado. Por exemplo, podemos determinar a representação de –2 em complemento-de-dois, partindo da representação de 2 e utilizando a tabela:

binário	*hexa*	*decimal*
...	...	...
0000000000000010	0002	2
0000000000000001	0001	1
0000000000000000	0000	0
1111111111111111	FFFF	-1
1111111111111110	FFFE	-2
1111111111111101	FFFD	-3
...	...	...

A negação de um número deve reverter esse processo, e um dos motivos da popularidade da representação complemento-de-dois deve-se ao fato de que podemos reverter o processo executando o mesmo procedimento: complementar todos os bits e adicionar um. Assim, para determinar a representação de $-(-2)$, fazemos:

binário	hexa	decimal
...	...	...
0000000000000010	0002	2
0000000000000001	0001	1
0000000000000000	0000	0
1111111111111111	FFFF	-1
1111111111111110	FFFE	-2
1111111111111101	FFFD	-3
...	...	...

complementa — soma 1

O procedimento para adição de dois números em complemento-de-dois também é bastante simples. A operação é efetuada como se os operandos fossem números sem sinal, ou seja, tratando o bit de sinal como os outros, e desprezando o bit de vai-um, se houver.

Exemplo 2.12: Adição de números em complemento-de-dois

$(-23422_{10}) + 22171_{10}$

```
 A482 +
 569B
 ----
 FB1D
```

Não houve vai-um, o resultado é FB1D

Conferindo: $FB1D_{16} = -1251_{10}$

$3673_{10} + (-1624_{10})$

```
  0E59 +
  F9A8
1 ----
  0801
```

Desprezamos o vai-um, o resultado é 801

Conferindo: $801_{16} = 2049_{10}$

Como o procedimento para adição não leva em conta o sinal dos operandos, ele funciona tanto se os operandos representarem números com sinal ou sem sinal. Assim, a interpretação dos valores representados fica por conta do programador, e o processador executa sempre o mesmo procedimento para adição.

Tomemos como exemplo uma das adições do exemplo anterior, mostrada ao lado. Se os operandos representam números inteiros, o procedimento de adição efetua o cálculo de $-23422_{10} + 22171_{10}$, e o resultado representa -1251_{10}. No entanto, podemos considerar que os operandos representam números naturais (inteiros positivos). Nesse caso o operando $A482_{16}$ não representa um número negativo, mas sim o número positivo $42144_{10,}$ e o resultado $FB1D_{16}$ deveria portanto representar 42114+22171 = 64285. E de fato, $FB1D_{16}$ representa também o inteiro positivo igual a $15\times4096 + 11\times256 + 1\times16 + 13 = 64285$.

```
A482 +
569B
----
FB1D
```

É importante notar que este procedimento para a adição só é válido se o resultado puder ser representado em uma palavra do computador. Caso contrário o resultado não é correto, ocorrendo uma condição de erro chamada de estouro de campo (*overflow*). Por exemplo, o resultado da adição de $81A0_{16}$ e $A58F_{16}$ não pode ser representado em um computador com palavra de 16 bits (o resultado é menor do que $-2^{16} - 1$, se os números representarem valores negativos, ou maior que $2^{16} - 1$ caso contrário). A ocorrência de estouro de campo deve ser informada ao programador, como veremos na seção x.x.

2.3 Representação de caracteres

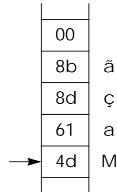
Para permitir que equipamentos como impressoras pudessem ser utilizados por diferentes marcas de computadores, os fabricantes desde muito cedo decidiram criar uma codificação padrão para caracteres. Inicialmente foi estabelecido pelos fabricantes um código de 7 bits, chamado ASCII (as iniciais de American Standard Code for Information Interchange – Código Padrão Americano para Intercâmbio de Informações). Com 7 bits é possível codificar 128 valores distintos, número mais que suficiente para todas as letras minúsculas e maiúsculas do alfabeto da língua inglesa, caracteres de pomtuação, bem como alguns caracteres especiais para movimentação da cabeça de impressão: avanço de linha, avanço de página, retorno à primeira coluna, etc. A tabela abaixo mostra uma pequena parte da codificação ASCII de 7 bits.

Esta codificação é suficiente para a impressão de textos em inglês em impressoras antigas tipo máquina de escrever. No entanto, em linguagens como português, onde várias letras podem ser acentuadas, a codificação ASCII de 7 bits não é tão adequada. Para imprimir uma letra acentuada como *ã*, por exemplo, é necessário fazer uma sobre-impressão. Ou seja, é necessário enviar para a impressora o caractere 'a' seguido do caractere de controle *volta-um-espaço* seguido do caractere '~'. E ainda assim, a qua-

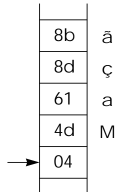
lidade gráfica do resultado é sofrível. Por isto, mais recentemente foi estabelecido um novo código de 8 bits, onde cada caractere acentuado tem uma codificação própria. Desta forma, para imprimir um caractere como *ã*, basta enviar à impressora o código ISO correspondente, que a impressora se encarrega de imprimi-lo com a melhor qualidade possível para aquele tipo de impressora.

Uma tabela da codificação ISO completa é fornecida no Apêndice x.

O fato de os caracteres serem representados com códigos de 8 bits facilita a representação de cadeias de caracteres na memória do computador, composta de um conjunto de bytes. Por exemplo, a seqüência de caracteres "Maçã" pode ser representada pela seqüência de bytes mostrada ao lado (o início é assinalado pela flecha, e os endereços crescem de cima para baixo). Para indicar o final de uma cadeia, podemos utilizar um caractere especial que por (nossa) convenção significa fim de cadeia. Na linguagem C, por exemplo, o final de cadeia é marcado com um byte 00h, como na figura ao lado (a seta indica o início da cadeia na memória).



Uma outra maneira de reconhecer o final de uma cadeia é armazenar o seu tamanho em caracteres em algum lugar. Na linguagem Pascal, é comum utilizar o primeiro byte de uma cadeia de caracteres para guardar o comprimento da cadeia (daí a restrição de comprimento máximo de 255 caracteres para o tipo "cadeia de caracteres" em algumas implementações).



Estes exemplos de representação de cadeias de caracteres são interessantes também para ressaltar um ponto importante, que às vezes provoca uma certa confusão: a diferença entre a representação da cadeia de caracteres '10' (seqüência de bytes 31h e 30h) e a representação do número 10 em binário (seqüência de bytes 00h e 0Ah na representação binária com 16 bits).

2.4 Exercícios

1. Converta os seguintes números binários para hexa:

   a) 10011111001 b) 0101010011
   c) 111101000101101 d) 110111000100101

2. Converta os seguintes números hexa para binário:

   a) 5A b) F81

   c) E3C0 d) B976

3. Converta os seguintes números decimais para hexa e binário:

   a) 267 b) 555
   c) 937201 d) 7346352

4. Converta os seguintes números hexa para decimal:

   a) 1001 b) 222
   c) A73E d) 8B10D

5. Efetue as operações diretamente em hexa e confira os resultados efetuando as operações também em decimal (todos os números estão em hexa):

   a) 3FF + 4872 b) 89A4 + C067
   c) 37891 – FA85 d) E75A – 18D
   e) 8A $\times$ 4 f) 7FF $\times$ 8
   g) 16 $\times$ 100 h) ABC / 2

6. Determine as representações em sinal e magnitude, complemento-de-um e complemento-de-dois dos seguintes números decimais, supondo números de 16 bits (mostre o resultado em binário e hexa):

   a) –183 b) 37671

7. Determine o valor em decimal dos seguintes números hexa (16 bits) em complemento-de-dois:

   a) 7C8F b) 003A
   d) FF01 d) 8AC5

8. Descreva uma maneira de determinar se houve estouro de campo em uma operação de adição em complemento-de-dois. Sugestão: considere o vai-um das casas relativas ao bit de sinal e ao bit mais significativo.

9. Descreva uma maneira de determinar se um número é maior que outro em complemento-de-dois.