



# PROGRAMAÇÃO BÁSICA DE BAIXO NÍVEL EM C



ESTE APÊNDICE apresenta tópicos de programação de baixo nível que podem ser necessários para completo entendimento do texto principal, notadamente aquele dos capítulos 7 e 8.

## B.1 Bases Numéricas

Como as operações apresentadas aqui envolvem manipulações de sequências de bits que são um tanto incômodas de ser escritas em formato binário, usualmente, o formato hexadecimal é utilizado para representá-las. Assim, antes de prosseguir, é importante que o leitor adquira a habilidade de transformar sequências de bits entre esses formatos. A **Tabela B-1** serve como referência auxiliar para essa tarefa.

BASE NUMÉRICA		
DECIMAL	BINÁRIA	HEXADECIMAL
0	0000	0x0
1	0001	0x1
2	0010	0x2
3	0011	0x3
4	0100	0x4
5	0101	0x5
6	0110	0x6

TABELA B-1: CONVERSÕES ENTRE BASES NUMÉRICAS

BASE NUMÉRICA		
DECIMAL	BINÁRIA	HEXADECIMAL
7	0111	0x7
8	1000	0x8
9	1001	0x9
10	1010	0xA
11	1011	0xB
12	1100	0xC
13	1101	0xD
14	1110	0xE
15	1111	0xF

TABELA B-1: CONVERSÕES ENTRE BASES NUMÉRICAS

Em programação de baixo nível, que envolve manipulação de bits, o formato mais comumente utilizado é o hexadecimal. Para converter uma sequência de bits do formato binário para hexadecimal, separe a sequência dada em sequências menores de quatro bits, completando, se for o caso, a primeira sequência de quatro bits com zeros à esquerda. Por exemplo, a sequência de bits **110101** pode ser separada em **0011** e **0101** (note que a primeira sequência foi completada com zeros à esquerda). Após fazer essa separação, escreva os números hexadecimais correspondentes a cada sequência de quatro bits utilizando a **Tabela B-1**. Na sequência de bits do exemplo, **0011** corresponde a **3** e **0101** corresponde a **5**. Portanto **110101** corresponde a **35** em hexadecimal, que se escreve em C como **0x35**.

Para converter de hexadecimal para binário, transforme cada dígito hexadecimal numa sequência de quatro bits utilizando a **Tabela B-1**. Por exemplo: **0xA52B** corresponde a **1010010100101011**, pois **0xA** corresponde a **1010**, **0x5** corresponde a **0101**, **0x2** corresponde a **0010** e **0xB** corresponde a **1011**.

## B.2 Operadores Lógicos de Bits

Existem quatro operadores lógicos de bits, que são resumidamente apresentados na **Tabela B-2**.

OPERADOR	SÍMBOLO	ARIDADE	PRECEDÊNCIA	ASSOCIATIVIDADE
Operador de complemento	~	1	Alta ↓ Baixa	À direita
Conjunção de bits	&	2		À esquerda
Disjunção exclusiva de bits	^	2		À esquerda
Disjunção de bits		2		À esquerda

TABELA B-2: OPERADORES LÓGICOS BINÁRIOS DE BITS

Todos os operadores de bits requerem que seus operandos sejam inteiros. Com exceção do operador de complemento, que é unário, cada um dos demais operadores de bits requer dois operandos inteiros e as operações são executadas sobre cada par de bits correspondentes nos dois operandos. Isto é, o primeiro bit do primeiro operando é avaliado com o primeiro bit do segundo operando, o segundo bit do primeiro operando é avaliado com o segundo bit do segundo operando, e assim por diante até que todos os bits sejam avaliados. O resultado

dessas avaliações para cada operador binário lógico de bits é apresentado na **Tabela B-3**. Nessa tabela,  $b_1$  e  $b_2$  representam um bit do primeiro operando e o bit correspondente do segundo operando, respectivamente.

$b_1$	$b_2$	$b_1 \& b_2$	$b_1   b_2$	$b_1 \wedge b_2$
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

**TABELA B-3: RESULTADOS DE OPERAÇÕES LÓGICAS BINÁRIAS DE BITS**

O operador de complemento, representado pelo símbolo  $\sim$  (til), é um operador unário cujo efeito é o de inverter os bits de seu operando.

## B.3 Operadores de Deslocamento de Bits

Existem dois operadores de deslocamento de bits em C, que são resumidamente apresentados na **Tabela B-4**.

OPERADOR	SÍMBOLO
Deslocamento de bits à esquerda	<<
Deslocamento de bits à direita	>>

**TABELA B-4: OPERADORES DE DESLOCAMENTO**

Cada um desses operadores de deslocamento requer dois operandos inteiros: o primeiro operando representa uma sequência de bits a ser deslocada, enquanto o segundo operando representa o número de deslocamentos que cada bit do primeiro operando irá experimentar. O valor do segundo operando não deverá ser maior do que o número de bits utilizados para representar o primeiro operando. A operação de deslocamento à esquerda faz com que os bits do primeiro operando sejam deslocados para a esquerda no número de posições indicado pelo segundo operando. Os bits mais à esquerda do primeiro operando, em número igual ao valor do segundo operando, serão perdidos na operação, enquanto os bits mais à direita do primeiro operando, também em número igual ao valor do segundo operando, serão preenchidos com 0.

Operadores de deslocamento possuem a mesma precedência, que é maior do que a precedência dos demais operadores binários de bits. A associatividade dos operadores de deslocamento é à esquerda.

É importante salientar que qualquer das operações de deslocamento apresentará um comportamento imprevisível e que não é portátil quando o segundo operando é negativo ou maior do que o tamanho do tipo do valor deslocado.

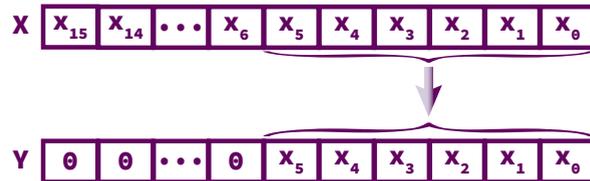
A operação de deslocamento à direita é similar à operação de deslocamento à esquerda quando o primeiro operando é sem sinal (i.e., **unsigned**), mas agora os bits são deslocados para a direita. Aliás, é sempre recomendável que o primeiro operando do operador de deslocamento à direita seja de fato **unsigned**, pois, se esse não for o caso e se o primeiro operando do operador de deslocamento à direita for um número negativo, o resultado do deslocamento é dependente de implementação.

## B.4 Mascaramento

Provavelmente, os usos práticos mais comuns de programação de baixo nível envolvem **mascaramento**, que é uma operação na qual uma dada sequência de bits é transformada em outra sequência desejada por meio de uma operação lógica de bits. Nessa operação, a sequência dada é um dos operandos do operador de bits e o

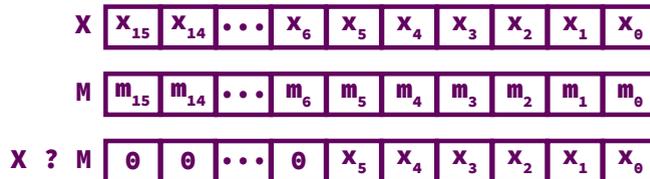
outro operando é uma sequência denominada **máscara**. Mais precisamente, uma máscara é um valor que, usado em conjunto com uma dada operação de bits, é empregado para extrair ou alterar o conteúdo de alguma variável. Numa operação de mascaramento, a máscara e a operação lógica são escolhidas de modo que a operação resulte na sequência de bits desejada.

Existem vários tipos de operações de mascaramento. Por exemplo, parte de uma variável pode ser copiada para outra variável, enquanto o restante da nova variável é preenchida com zeros, conforme mostra a **Figura B-1**.



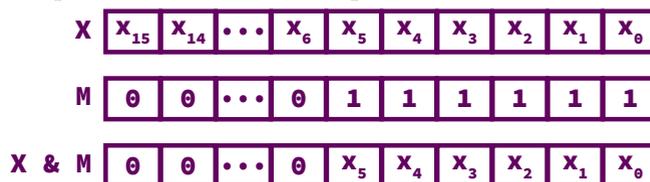
**FIGURA B-1: EXEMPLO DE MASCARAMENTO: COPIANDO BITS DE UMA VARIÁVEL 1**

Suponha que  $X$  seja uma variável do tipo **int** e que se deseje extrair os seis bits mais à direita de  $X$  e atribuí-los a uma variável  $Y$ , como mostra o diagrama da **Figura B-1**. Os bits restantes de  $Y$  devem ser preenchidos com zeros. A questão é: como escolher o operador lógico de bits e a máscara para essa operação de mascaramento? Para uma melhor visualização do problema, a operação de mascaramento é ilustrada na **Figura B-2**. Nessa figura, é feita a suposição de que o tipo **int** ocupa 16 bits, mas isso é apenas comodidade e não constitui uma limitação.



**FIGURA B-2: EXEMPLO DE MASCARAMENTO: COPIANDO BITS DE UMA VARIÁVEL 2**

Na **Figura B-2**,  $X$  é uma variável qualquer do tipo **int** e, conseqüentemente, os bits  $x_i$  de  $X$  também são arbitrários. Os bits que compõem a máscara  $M$  devem ser escolhidos adequadamente, de modo a resultar na sequência de bits dada por  $X \text{ ? } M$ , em que  $?$  representa um operador de bits a ser convenientemente escolhido. Examinando-se a **Tabela B-3** dos operadores de bits apresentada na **Seção B.2**, pode-se concluir que esse operador deve ser **&** e que a máscara deve ser constituída por uma sequência de zeros, correspondente à porção do resultado contendo apenas zeros (i.e., os bits numerados de 6 a 31) e uma sequência de uns, correspondente à porção do resultado contendo os valores originais do operando dado (i.e., os bits numerados de 0 a 5). Levando isso em consideração, o resultado da operação de mascaramento pode ser visualizado como é ilustrado na **Figura B-3**.



**FIGURA B-3: EXEMPLO DE MASCARAMENTO: COPIANDO BITS DE UMA VARIÁVEL 3**

Portanto a máscara da operação de mascaramento do exemplo acima tem a seguinte representação binária:

0000 0000 0011 1111<sub>2</sub>

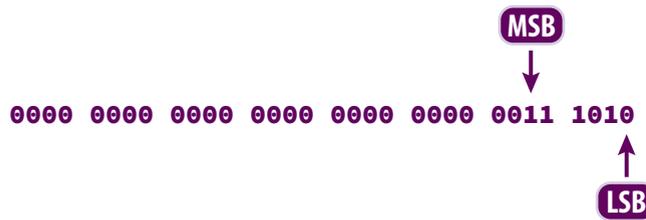
que corresponde, em formato hexadecimal, a **0x3F**.

É interessante notar que a máscara do último exemplo é independente do tamanho do tipo **int** utilizado na operação, pois seus bits mais à esquerda são todos zeros. Por exemplo, se o tipo **int** ocupasse 32 bits, o valor da

máscara continuaria sendo  $0x3F$ . Nesse caso, a única diferença seria que a representação binária conteria mais zeros à esquerda, mas o valor da máscara seria o mesmo. Entretanto máscaras cujos bits mais à esquerda são iguais a 1 não são independentes do tamanho do tipo inteiro utilizado na operação de mascaramento.

## B.5 Extraíndo Bits Significativos

O bit menos significativo (**LSB**<sup>[1]</sup>) de um número inteiro em base binária é aquele que se encontra mais à direita no número (i. e, ele é o bit que ocupa a posição zero). Por sua vez, o bit mais significativo (**MSB**<sup>[2]</sup>) de um número inteiro diferente de zero em base binária é o bit mais à esquerda do número, desde que ele seja diferente de 0 e não seja um bit de sinal. Por exemplo, o bit mais significativo e o bit menos significativo do número  $58_{10}$  representado em base binária como um inteiro de 32 bits é apresentado na **Figura B-4**.



**FIGURA B-4: BIT MAIS SIGNIFICATIVO (MSB) E BIT MENOS SIGNIFICATIVO (LSB)**

Note que o MSB e o LSB apresentam as seguintes propriedades:

- ❑ O LSB de um número inteiro ocupa sempre a posição 0, mas seu valor depende do valor do número.
- ❑ O MSB de um número inteiro diferente de zero é sempre 1, mas sua posição depende do valor do número.

Alguns programas, tal como a implementação de dispersão extensível apresentada na **Seção 8.2**, requerem que  $n$  bits menos ou mais significativos sejam extraídos de um número inteiros e esta seção mostrará como esses bits podem ser obtidos por meio de operações de mascaramento (v. **Seção B.4**).

### B.5.1 Bits Menos Significativos (LSBs)

Como foi visto na **Seção B.4**, para obter os seis bits menos significativos de um número inteiro, usa-se uma máscara cujos seis bits mais à direita são iguais a 1 e cujos demais bits são iguais a 0. Após essa máscara ter sido obtida, aplica-se o operador de conjunção de bits para extrair os bits desejados. Intuitivamente, pode-se generalizar esse raciocínio para a extração dos  $n$  bits menos significativos de um número inteiro: basta considerar uma máscara semelhante que, em vez de seis, tenha  $n$  bits iguais a 1 e usar esse mesmo operador. O problema agora é como obter essa máscara.

Suponha que se deseje extrair os quatro bits menos significativos de um número inteiro. Então, seguindo o raciocínio exposto acima, a máscara a ser utilizada será<sup>[3]</sup>:

$$0000\ 0000\ 0000\ 1111_2$$

Se a essa máscara for acrescentado 1, o resultado obtido é aquele apresentado na **Figura B-5**.

$$\begin{array}{r} \text{Máscara} \rightarrow 0000\ 0000\ 0000\ 1111 \\ \phantom{\text{Máscara}} \phantom{\rightarrow} \phantom{0000\ 0000\ 0000\ } + 1 \\ \hline 1 \ll 4 \rightarrow 0000\ 0000\ 0001\ 0000 \end{array}$$

**FIGURA B-5: EXTRAÇÃO DE BITS MENOS SIGNIFICATIVOS**

[1] A sigla *LSB* é derivada de *least significant bit* em inglês.

[2] A sigla *MSB* é derivada de *most significant bit* em inglês.

[3] Novamente, para facilitar o entendimento, serão usados inteiros de 16 bits, mas o mesmo raciocínio pode ser aplicado para inteiros de qualquer largura.

Agora, se somando-se  $1$  à máscara obtém-se o resultado da operação  $1 \ll 4$ , subtraindo-se  $1$  dessa última expressão, obtém-se a máscara desejada. Essa conclusão pode ser generalizada numa fórmula em C para obtenção de uma máscara para extração dos  $n$  bits menos significativos de um número inteiro como:

```
mascara = (1 << n) - 1
```

A função `ObtemLSBs()` a seguir retorna os bits menos significativos de um número inteiro e seus parâmetros são:

- `chave` (entrada) — o número inteiro do qual os bits serão extraídos
- `nBits` (entrada) — o número de bits menos significativos que serão extraídos da chave

```
int ObtemLSBs(int chave, int nBits)
{
    int mascara = (1 << nBits) - 1;
    return chave & mascara;
}
```

### B.5.2 Bits Mais Significativos (MSB)

Para a extração dos  $n$  bits mais significativos de um número inteiro não negativo, pode-se usar a mesma fórmula para máscara apresentada na [Seção B.5.1](#), mas a extração desses bits é uma tarefa um tanto sutil, pois, em princípio, não se sabe onde eles se encontram.

Suponha, por exemplo, que se deseja extrair os quatro bits mais significativos de  $2568_{10}$ , cuja representação em base binária é:

```
0000 1010 0000 10002
```

Agora, usando-se máscara para extração de quatro bits obtida na [Seção B.5.1](#), a presente situação é aquela mostrada na [Figura B-6](#).

Note que, como mostra a [Figura B-6](#), não há nenhum operador de bits que possa ser aplicado aos números em questão para resultar nos bits almejados. Então a ideia é fazer com que o número sofra sucessivos deslocamentos de bits para a direita até que os bits desejados coincidam com a porção da máscara contendo apenas bits iguais a  $1$ , como mostra a [Figura B-7](#).



A questão agora é: como saber se os bits mais significativos estão alinhados da maneira preconizada? Bem, como se observa na [Figura B-7](#), nessa situação, todos os bits que antecedem os bits desejados e aqueles que antecedem a porção de bits iguais a  $1$  da máscara são iguais a  $0$ , de modo que, no máximo, o número será igual à máscara (na referida figura, o número é menor do que a máscara).

A função `ObtemMSBs()` abaixo retorna os bits mais significativos de um número inteiro positivo seguindo o arrazoado apresentado no último parágrafo. Os parâmetros dessa função são:

- `chave` (entrada) — o número inteiro positivo do qual os bits serão extraídos
- `nBits` (entrada) — o número de bits mais significativos que serão extraídos da chave

```
int ObtemMSBs(int chave, int nBits)
{
    int mascara = (1 << nBits) - 1;
    ASSEGURA(chave >= 0, "A chave nao pode ser negativa");
    /* Desloca a chave para a direita, um bit de cada vez, até que */
    /* seu valor seja menor do que ou igual ao valor da máscara */
    while (chave > mascara)
        chave >>= 1;
    return chave;
}
```

## B.6 Ligando e Desligando Bits

**Ligar um bit** significa torná-lo igual a  $1$  e **desligar um bit** significa torná-lo igual a  $0$ .

Para ligar um bit que se encontra numa posição  $j$ , usa-se uma máscara contendo  $1$  nessa posição e  $0$  nas demais posições e, então, usa-se o operador de disjunção de bits `|`, como ilustra a **Figura B-8**. Para obter a máscara desejada, a ela deve ser atribuído o valor da expressão  $1 \ll j$ , como mostra essa mesma figura.

Para desligar um bit que se encontra numa posição  $j$ , usa-se uma máscara contendo  $0$  nessa posição e  $1$  nas demais posições e, então, usa-se o operador de conjunção de bits `&`, como ilustra a **Figura B-9**. Para obter a máscara desejada, obtém-se o valor da expressão  $1 \ll j$  e depois aplica-se a ele o operador de negação de bits, como mostra a **Figura B-9**.

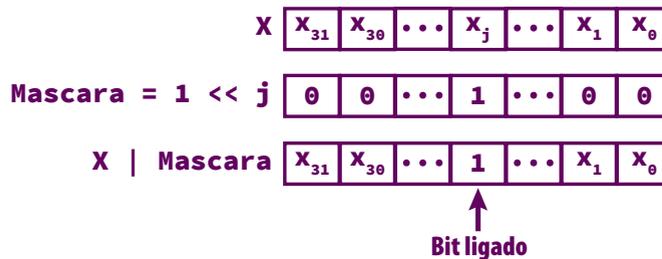


FIGURA B-8: LIGANDO UM BIT

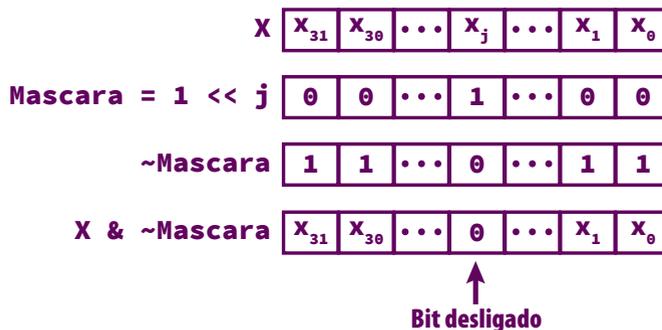


FIGURA B-9: DESLIGANDO UM BIT

As funções `LigaBit()` e `DesligaBit()` apresentadas a seguir são usadas, respectivamente, para ligar e desligar um bit.

```

int LigaBit(int valor, int pos)
{
    int mascara = 1 << pos;
    return valor | mascara;
}

int DesligaBit(int valor, int pos)
{
    int mascara = 1 << pos;
    return valor & ~mascara;
}

```

## B.7 Invertendo um Bit

Para inverter o valor de um bit, usa-se a mesma máscara usada para ligar um bit (v. Seção B.4) em conjunto com o operador de disjunção exclusiva de bits  $\wedge$ , como mostra a Figura B-10.

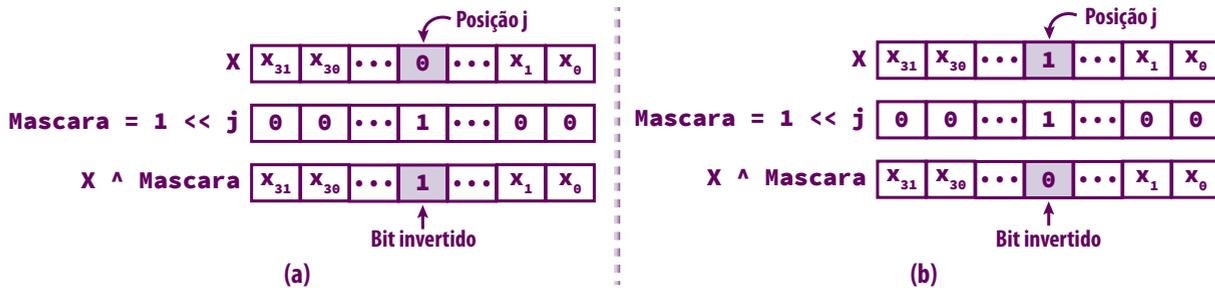


FIGURA B-10: INVERTENDO UM BIT

A função `InverteBit()` retorna o valor resultante da inversão de um bit de um número inteiro recebido como parâmetro.

```

int InverteBit(int valor, int pos)
{
    int mascara = 1 << pos;
    return valor ^ mascara;
}

```

## B.8 Consultando um Bit

Para verificar se o valor de um bit de um número inteiro é 0 (desligado) ou 1 (ligado), usa-se uma máscara com valor igual a 1. Então deslocam-se os bits do número para a direita por um valor igual à posição do número que se deseja consultar. Finalmente, aplica-se o operador de conjunção de bits  $\&$ , como mostra a Figura B-11.

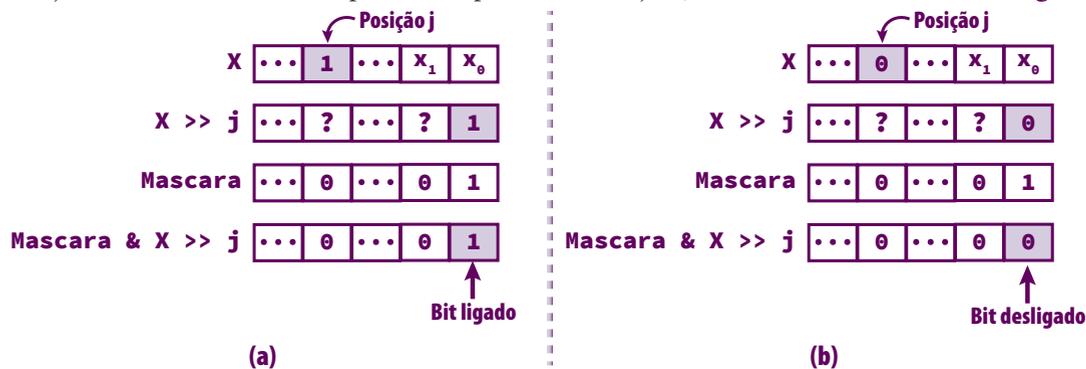


FIGURA B-11: CONSULTANDO UM BIT

A função `ConsultaBit()` pode ser usada para consultar o bit de um número inteiro que se encontra numa determinada posição.

```
int ConsultaBit(int valor, int pos)
{
    return 1 & valor >> pos;
}
```

## B.9 Exemplos de Programação

### B.9.1 Representação Binária

Esta seção apresenta uma função denominada `RepresentacaoBinaria()`, que exhibe, no meio de saída padrão, a sequência de bits correspondente à representação binária de um valor do tipo `int` passado como parâmetro para a função.

```
void RepresentacaoBinaria(int numero)
{
    int          i, bit, numeroDeBits;
    unsigned int mascara; /* A máscara deve ser unsigned */

    /* Cada byte contém um número de bits igual a CHAR_BIT */
    numeroDeBits = CHAR_BIT*sizeof(int);

    /* Coloca 1 na posição mais à esquerda da */
    /* máscara. Todos os outros bits terão 0s. */
    mascara = 0x1 << (numeroDeBits - 1);

    for (i = 1; i <= numeroDeBits; i++) { /* Exibe cada bit a partir da esquerda */
        bit = (numero & mascara) ? 1 : 0;

        printf("%d", bit); /* Apresenta um bit na tela */

        /* Separa sequências de bytes com espaços em branco */
        /* para melhor visualização da representação */
        if (i % CHAR_BIT == 0)
            printf(" ");

        mascara >>= 1; /* Move bit com valor 1 da máscara uma posição para a direita */
    }
}
```

A função `RepresentacaoBinaria()` merece alguns comentários:

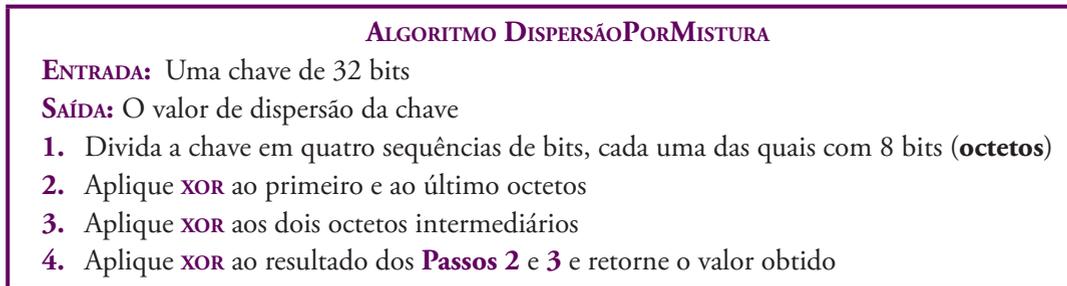
- ❑ A constante `CHAR_BIT` é utilizada para calcular o número de bits utilizados pelo valor que será representado. Essa constante está definida no arquivo de cabeçalho `<limits.h>`<sup>[4]</sup>.
- ❑ A máscara utilizada é representada pela variável `mascara`. Em qualquer instante, essa variável possui apenas um bit `1` e os demais bits são todos zeros. A posição do bit com valor `1` na máscara corresponde à posição do bit que se está testando se é `0` ou `1` no número recebido como parâmetro. Inicialmente, o bit mais à esquerda da máscara recebe o valor `1`. Esse bit é deslocado uma posição para a direita, utilizando o operador `>>`, a cada execução do corpo do laço `for`.
- ❑ No interior do laço, a expressão `x & y` resulta num valor diferente de zero quando o bit do número a ser representado na posição correspondente ao bit com valor `1` na máscara for também igual a `1`; caso contrário, esse bit deve ser `0`.
- ❑ A instrução `if` no interior do laço serve apenas para inserir alguns espaços em branco separando os bits da representação em bytes para facilitar a visualização.

[4] A constante `CHAR_BIT` existe porque a linguagem C não especifica quantos bits um byte deve possuir. Mas mesmo que você não conheça nenhum computador cujo número de bits num byte seja diferente de 8, é melhor utilizar essa constante por questão de legibilidade.

A função `RepresentacaoBinaria()` não é apenas um exemplo interessante de manipulação de bits. Ela é bastante útil na depuração de programas que processam bits.

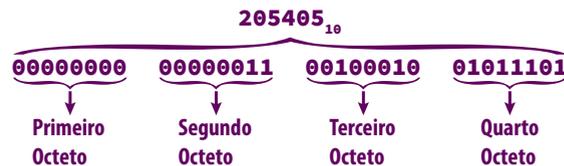
### B.9.2 Dispersão por Mistura

Suponha que se deseja elaborar uma função de dispersão que resulta num índice entre 0 e 255 e a representação interna de chaves do tipo `int` é uma sequência binária de 32 bits. Nesse caso, são usados 8 bits para representar os 256 valores de índices (pois  $2^8$  é igual a 256). Um algoritmo de dispersão por mistura usado para criar essa função de dispersão é mostrado na **Figura B-12**.



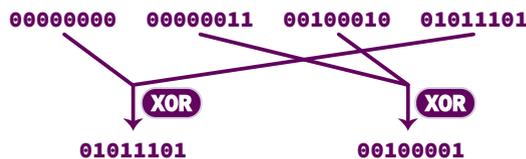
**FIGURA B-12: ALGORITMO DE DISPERSÃO POR MISTURA**

Esse algoritmo é exemplificado usando-se a chave 205405, cuja representação binária é apresentada na **Figura B-13**.



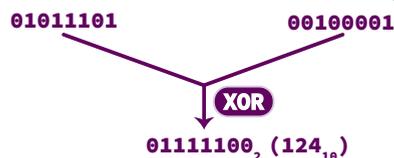
**FIGURA B-13: PRIMEIRO PASSO DE DISPERSÃO POR MISTURA**

O segundo passo do algoritmo da **Figura B-12** consiste em aplicar **XOR** ao primeiro e ao último octetos, enquanto o terceiro passo efetua a mesma operação com o segundo e o terceiro octetos. Os resultados dessas operações são apresentados na **Figura B-14**.



**FIGURA B-14: SEGUNDO E TERCEIRO PASSOS DE DISPERSÃO POR MISTURA**

O último passo do algoritmo da **Figura B-12** aplica **XOR** ao resultado dos dois passos anteriores. O resultado dessa última operação é apresentado na **Figura B-15**. Note que o número binário obtido é equivalente ao número decimal 124, de modo que a chave 205405 resulta no valor de dispersão 124.



**FIGURA B-15: QUARTO PASSO DE DISPERSÃO POR MISTURA**

A implementação de uma função de dispersão por mistura pode não ser tão trivial quanto aparenta ser se o leitor não possuir treinamento adequado em programação de baixo nível. Espera-se que as seções anteriores deste

apêndice tenham provido o conhecimento mínimo necessário para o leitor acompanhar a implementação do algoritmo de dispersão por mistura delineado na **Figura B-12**.

O maior problema com esse algoritmo é que ele não detalha como obter os octetos de bytes com os quais se efetuam as operações **XOR**. Mais precisamente, esse algoritmo não especifica como obter os octetos que compõem o valor inteiro que representa a chave. E esse é exatamente o passo mais complicado desse algoritmo.

A **Figura B-16** apresenta dois exemplos de extração de octetos por meio de mascaramento, que é uma operação que foi discutida na **Seção B.4**. A **Figura B-16 (a)** mostra a extração do segundo octeto da chave do exemplo apresentado acima, ao passo que a **Figura B-16 (b)** mostra a extração do terceiro octeto dessa mesma chave.

Infelizmente, os octetos obtidos com essas operações de mascaramento não produzem o resultado desejado quando for aplicada a operação **XOR** sobre eles. Por exemplo, considerando os octetos obtidos na **Figura B-16**, o resultado obtido quando se aplica o operador **XOR** (representado por **^** em C) é aquele mostrado na **Figura B-17**, que não corresponde àquilo que se esperava obter.

Em contraste com o que mostra a **Figura B-17**, esperava-se obter como resultado apenas um único octeto, em vez de dois. Mais precisamente, esperava-se obter o octeto **00100001**, conforme foi visto acima (v. **Figura B-14**). O resultado inesperado é advindo do fato de os octetos sobre os quais incide a operação **XOR** estarem desalinhados. Para alinhar esses octetos, aplica-se um deslocamento à direita a cada um deles, de modo que eles ocupem os bits de menor ordem, como mostra a **Figura B-18**.

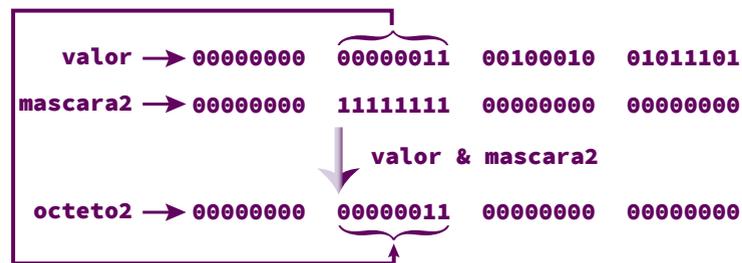
Note que o uso de parênteses nas expressões:

```
(valor & mascara2) >> 2*8
```

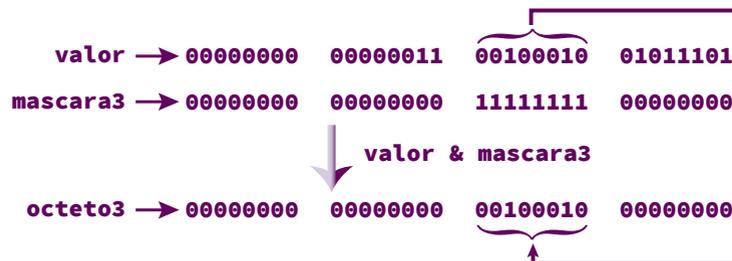
e

```
(valor & mascara2) >> 1*8
```

na **Figura B-18** são de fato necessários porque o operador **>>** tem precedência maior do que o operador **&**. A **Figura B-19** mostra como, agora, o resultado desejado é obtido.



(a) Extração do Segundo Octeto



(b) Extração do Terceiro Octeto

**FIGURA B-16: EXTRAÇÃO INCORRETA DE OCTETOS EM DISPERSÃO POR MISTURA**

```

octeto2 → 00000000 00000011 00000000 00000000
octeto3 → 00000000 00000000 00100010 00000000
                                     ↓ octeto2 ^ octeto3
00000000 00000011 00100010 00000000

```

FIGURA B-17: RESULTADO INCORRETO DE DISPERSÃO POR MISTURA

```

valor → 00000000 00000011 00100010 01011101
mascara2 → 00000000 11111111 00000000 00000000
                                     ↓ (valor & mascara2) >> 2*8
octeto2 → 00000000 00000000 00000000 00000011
                                     ↑

```

(a) Extração do Segundo Octeto com Deslocamento

```

valor → 00000000 00000011 00100010 01011101
mascara3 → 00000000 00000000 11111111 00000000
                                     ↓ (valor & mascara3) >> 1*8
octeto3 → 00000000 00000000 00000000 00100010
                                     ↑

```

(b) Extração do Terceiro Octeto com Deslocamento

FIGURA B-18: EXTRAÇÃO CORRETA DE OCTETOS COM DESLOCAMENTO

```

octeto2 → 00000000 00000000 00000000 00000011
octeto3 → 00000000 00000000 00000000 00100010
                                     ↓ octeto2 ^ octeto3
00000000 00000000 00000000 00100001

```

FIGURA B-19: RESULTADO CORRETO DE DISPERSÃO POR MISTURA

O valor resultante de cada operação **XOR** acima cabe num byte, o que significa dizer que esse resultado cabe numa variável do tipo **char**. Uma consequência dessa última conclusão é que o método de mistura descrito aqui só pode ser usado para tabelas com tamanho máximo de 256 elementos, o que constitui uma limitação desse método. Mas, de qualquer modo, as técnicas de manipulação de bits que você deve ter aprendido aqui servirão para compreender outros métodos de dispersão e ser capaz de desenvolver seus próprios algoritmos de cálculo de valores de dispersão.

A função `DispersaoMisturaJunior()` calcula o valor de dispersão de uma chave usando o método de mistura e seus parâmetros são a chave e o tamanho da tabela de dispersão. Essa função não é a melhor implementação para o algoritmo descrito nesta seção. Ou seja, ela é dirigida para leitores iniciantes em programação de baixo nível em C.

```

unsigned DispersaoMisturaJunior(int chave)
{
    unsigned mascara[4] = { 0xFF000000, /* 11111111 00000000 00000000 00000000 */
                           0x00FF0000, /* 00000000 11111111 00000000 00000000 */
                           0x0000FF00, /* 00000000 00000000 11111111 00000000 */
                           0x000000FF, /* 00000000 00000000 11111111 00000000 */
    };
}

```

```

char    octetos[4] = { (chave & mascara[0]) >> 3*8,
                      (chave & mascara[1]) >> 2*8,
                      (chave & mascara[2]) >> 1*8,
                      (chave & mascara[3]) >> 0*8
                      };
unsigned i, xor1, xor2;
xor1 = octetos[0] ^ octetos[3];
xor2 = octetos[1] ^ octetos[2];
return xor1 ^ xor2;
}

```

O array `mascara[]` contém os valores que serão empregados na extração das sequências de bits. A [Tabela B-5](#) ajuda a entender os valores usados na iniciação desse array.

HEXADECIMAL	BINÁRIO
0xFF000000	11111111 00000000 00000000 00000000
0x00FF0000	00000000 11111111 00000000 00000000
0x0000FF00	00000000 00000000 11111111 00000000
0x000000FF	00000000 00000000 00000000 11111111

**TABELA B-5: MÁSCARAS EM FORMATOS HEXADECIMAL E BINÁRIO**

O array `octetos[]` armazena os octetos extraídos da chave e, se você realmente entendeu o arrazoado apresentado acima, não terá dificuldade em compreender como esse array é iniciado.

Uma versão um pouco mais sofisticada da função `DispersaoMisturaJunior()` leva em consideração o fato de, com exceção da primeira máscara, as demais máscaras poderem ser obtidas por meio da aplicação do operador de deslocamento à direita, de modo que não é necessário um array para armazená-las: basta uma única variável do tipo **unsigned**, como mostra a função `DispersaoMisturaMedio()` a seguir.

```

unsigned DispersaoMisturaMedio(int chave)
{
    unsigned mascara = 0xFF000000,
            i, xor1, xor2;
    char    octetos[4];

    for (i = 0; i < 4; ++i) {
        octetos[i] = (chave & mascara) >> ((3 - i)*8);
        mascara = mascara >> 8;
    }

    xor1 = octetos[0] ^ octetos[3];
    xor2 = octetos[1] ^ octetos[2];

    return xor1 ^ xor2;
}

```

A função `DispersaoMisturaMedio()` ainda está longe de ser a melhor implementação para a situação corrente, mas se você tinha deficiência em programação de baixo nível e entendeu o que foi exposto nesta seção, parabéns! Foi uma grande evolução. No site dedicado ao livro na internet você encontrará uma versão ainda mais sofisticada de função que implementa dispersão por mistura.

