APÊNDICE A
PROGRAMAÇÃO DE BAIXO NÍVEL EM C BÁSICA

Este apêndice apresenta tópicos de programação de baixo nível que podem ser necessários para completo entendimento do texto principal, notadamente o **Capítulo 8**.

A.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 A–1** serve como referência auxiliar para essa tarefa.

DECIMAL	BINÁRIO	HEXADECIMAL
0	0000	0x0
1	0001	0x1
2	0010	0x2
3	0011	0x3
4	0100	0x4
5	0101	0x5
6	0110	0x6
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 A–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 A–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 A–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`.

A.2 Operadores Lógicos de Bits

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

Operador	Símbolo	Aridade	Precedência	Associatividade
Operador de complemento	~	1	Alta	À direita
Conjunção de bits	&	2		À esquerda
Disjunção exclusiva de bits	^	2		À esquerda
Disjunção de bits	|	2	Baixa	À esquerda

Tabela A–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 A–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 ^ b_2
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

Tabela A–3: Resultados de Operações Lógicas Binárias de Bits

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

A.3 Operadores de Deslocamento de Bits

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

Operador	Símbolo
Deslocamento à esquerda	`<<`
Deslocamento à direita	`>>`

Tabela A–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 que 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 que 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 não-portável 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.

A.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 que o restante da nova variável é preenchida com zeros, conforme mostra a **Figura A–1**.

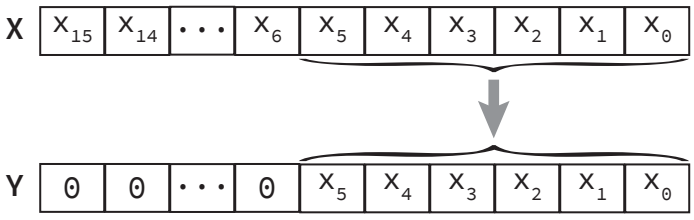

FIGURA A–1: OPERAÇÃO DE MASCARAMENTO 1

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 A–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 A–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.

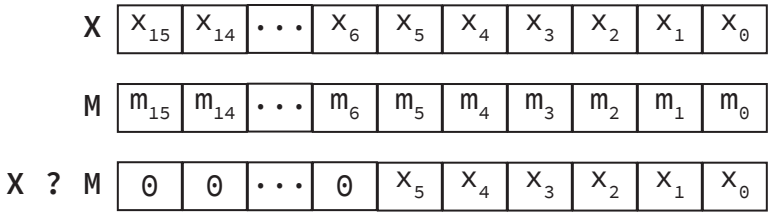

FIGURA A–2: OPERAÇÃO DE MASCARAMENTO 2

Na **Figura A–2**, `X` é uma variável qualquer do tipo **int** e, consequentemente, 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 ? M`, em que `?` representa um operador de bits a ser convenientemente escolhido. Examinando-se a **Tabela A–3** dos operadores de bits apresentada na **Seção A.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 A–3**.

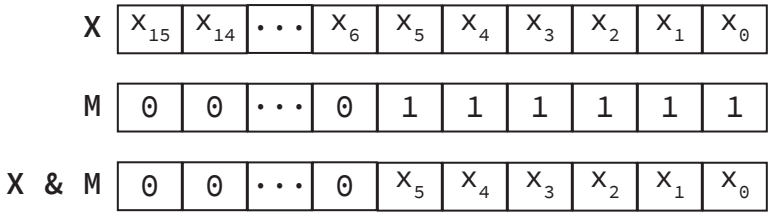

FIGURA A–3: OPERAÇÃO DE MASCARAMENTO 3

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

$$\texttt{0000 0000 0011 1111}_2$$

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.

A.5 Extraindo Bits Significativos

O bit menos significativo (**LSB**[1]) 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**[2]) 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 A–4**.

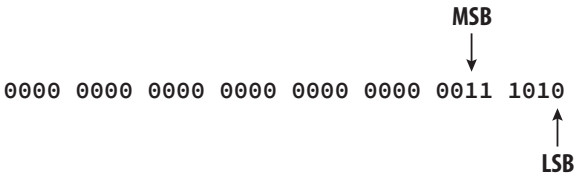

Figura A–4: Bit Mais Significativo (MSB) e Bit Menos Significativo (LSB)

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 8.7**, 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 A.4**).

A.5.1 Extração de Bits Menos Significativos (LSBs)

Como foi visto na **Seção A.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

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

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

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[3]:

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

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

```
Máscara ⟶  0000 0000 0000 1111
                          + 1
           ---------------------
1 << 4 ⟶   0000 0000 0001 0000
```

Figura A–5: Extração de Bits Menos Significativos

Agora, se somando-se *1* à máscara obtém-se o resultado da operação *1 << 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;
}
```

A.5.2 Extração de 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 A.5.1**, mas a extração desse 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:

$$0000\ 1010\ 0000\ 1000_2$$

Agora, usando-se máscara para extração de quatro bits obtida na **Seção A.5.1**, a presente situação é aquela mostrada na **Figura A–6**.

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

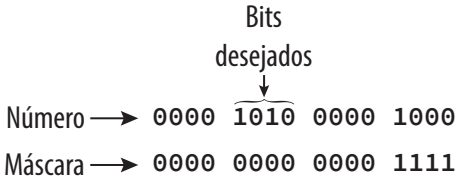

Figura A–6: Extração de Bits Mais Significativos 1

Note que, como mostra a **Figura A–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 A–7**.

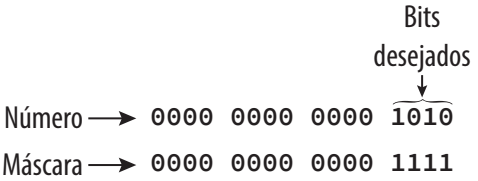

Figura A–7: Extração de Bits Mais Significativos 2

A questão agora é: como saber se os bits mais significativos estão alinhados da maneira preconizada? Bem, como se observa na **Figura A–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;
}
```

A.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 A–8**. Para obter a máscara desejada, a ela deve ser atribuído o valor da expressão *1 << j*, como mostra essa mesma figura.

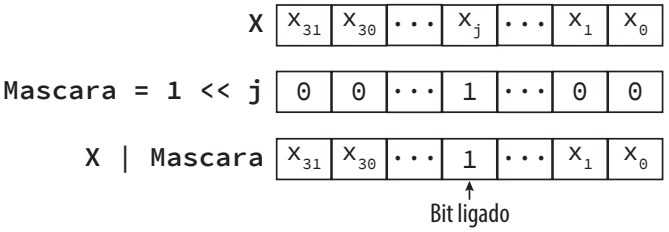

FIGURA A–8: LIGANDO UM BIT

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 A–9**. Para obter a máscara desejada, obtém-se o valor da expressão *1 << j* e depois aplica-se a ele o operador de negação de bits, como mostra a **Figura A–9**.

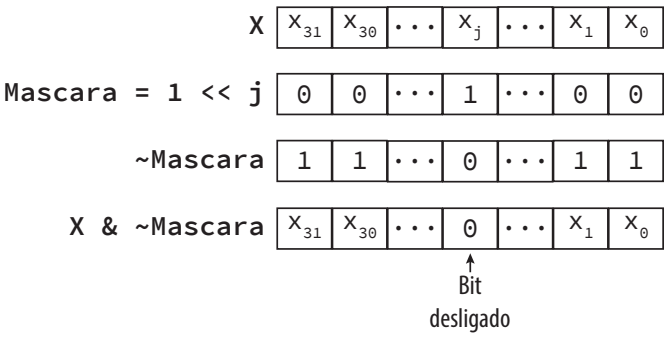

FIGURA A–9: DESLIGANDO UM BIT

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;
}
```

A.7 Invertendo um Bit

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

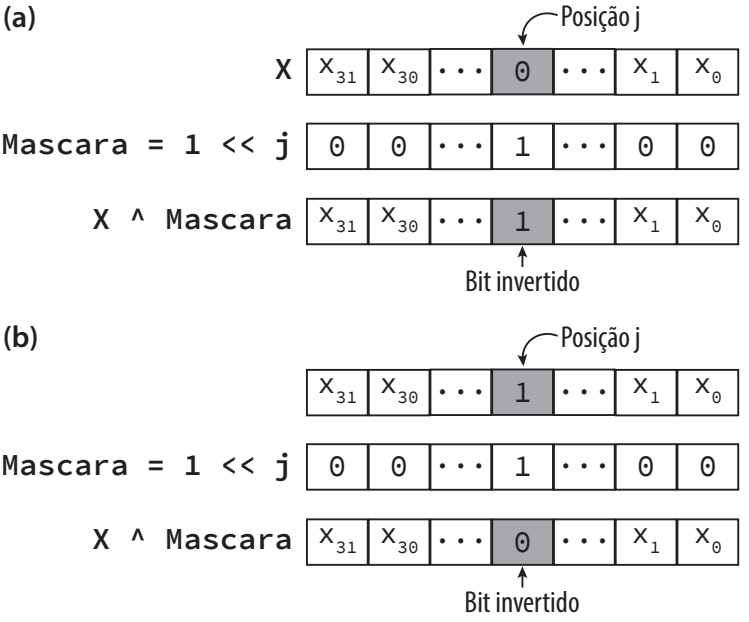

FIGURA A–10: INVERTENDO UM BIT

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;
}
```

A.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 A–11**.

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

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

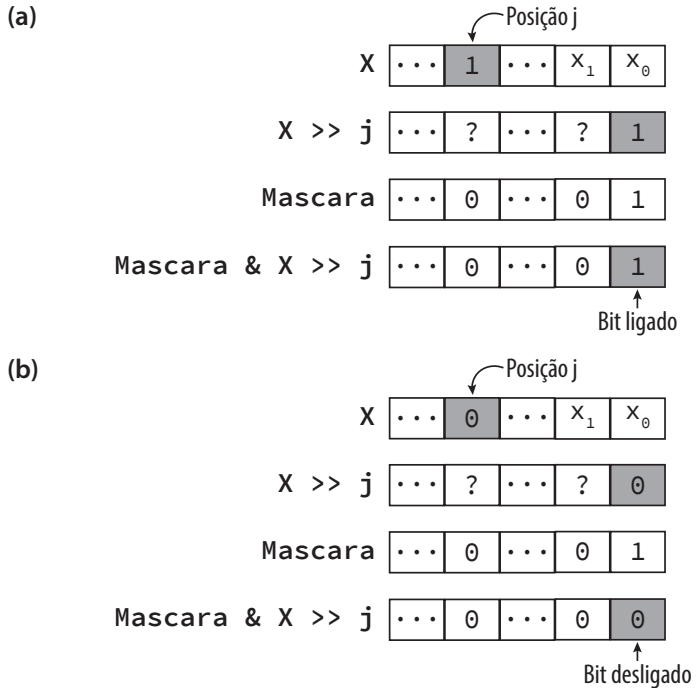

FIGURA A–11: CONSULTANDO UM BIT

A.9 Exemplo: Representação Binária

Esta seção apresenta uma função denominada `RepresentacaoBinaria()`, que exibe, 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);

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

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

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

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

```
    }
}
```

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>`[4].
- 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.

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

A.10 Exemplo: 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 um string binário de 32 bits. Sabe-se que se usam 8 bits para representar os *256* valores de índices (pois 2^8 é igual a *256*). Um algoritmo de mistura para criar essa função de dispersão poderia ser o seguinte:

1. Divida a chave em quatro strings de bits, cada um dos quais com 8 bits
2. Aplique XOR ao primeiro e ao último strings de bits
3. Aplique XOR aos dois strings de bits intermediários
4. Aplique XOR ao resultado dos passos 2 e 3 para obter o índice do array de 8 bits

Esse algoritmo é ilustrado usando-se a chave *205405*, cuja representação binária é apresentada a seguir com espaços em branco separando seus agrupamentos de 8 bits:

```
00000000  00000011  00100010  01011101
```

Logo têm-se os seguintes strings de 8 bits:

- Primeiros 8 bits: `00000000`
- Próximos 8 bits: `00000011`
- Próximos 8 bits: `00100010`
- Últimos 8 bits: `01011101`

[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.

O próximo passo é aplicar XOR ao primeiro e ao último strings de bits. O resultado da operação XOR sobre dois bits é `0` se os dois bits são os mesmos, e `1` se eles são diferentes (v. **Tabela A–3**). Para aplicar XOR a dois strings de bits, essa regra é aplicada a pares correspondentes de bits. Portanto tem-se que:

```
00000000 ^ 01011101
```

resulta em:

```
01011101
```

Então aplica-se XOR aos dois strings de bits intermediários:

```
00000011 ^ 00100010
```

que resulta em:

```
00100001
```

Finalmente, aplica-se XOR ao resultado dos dois passos anteriores, o que resulta em:

```
01011101 ^ 00100001
```

que é igual a:

```
01111100
```

Esse número binário é equivalente ao número decimal `124`, de modo que a chave `205405` resulta no índice `124`. A **Figura A–12** ilustra o método de cálculo de dispersão por mistura.

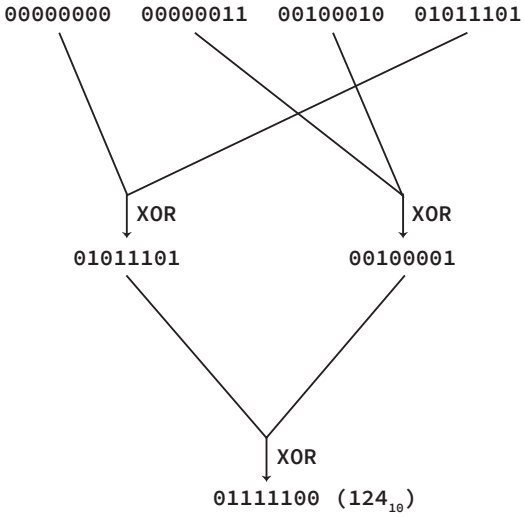

FIGURA A–12: MÉTODO DE DISPERSÃO POR MISTURA

A implementação dessa 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, o que é comum entre estudantes de disciplinas de Estruturas de Dados. Portanto serão apresentadas duas versões dessa função. A primeira delas, denominada `DispersaoMisturaJunior()`, é dirigida para esses alunos que não apresentam bom conhecimento em programação de baixo nível em C. Se você julga que tem conhecimento

suficiente para acompanhar a versão sênior que será apresentada mais adiante, você pode pular a discussão sobre a versão júnior apresentada a seguir.

Antes de discutir a implementação do mencionado algoritmo em si, o leitor será provido com algum conhecimento sobre operações de bits em C.

O maior problema com o algoritmo apresentado acima 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 bytes que compõem o valor inteiro que representa a chave. E esse é exatamente o passo do algoritmo que é mais complicado, pois ele requer o uso de duas operações de baixo nível:

1. Mascaramento, que é uma operação de baixo nível discutida na **Seção A.4**. A **Figura A–13** apresenta dois exemplos de extração de bits por meio de mascaramento. A **Figura A–13 (a)** mostra a extração do segundo octeto da chave do exemplo apresentado acima, enquanto que a **Figura A–13 (b)** mostra a extração do terceiro octeto dessa mesma chave.
2. Infelizmente, os octetos obtidos na operação de mascaramento não produzem o resultado desejado quando for aplicada a operação XOR sobre eles. Por exemplo, considerando os octetos obtidos na **Figura A–13**, o resultado obtido quando se aplica o operador XOR (representado por `^` em C) é aquele mostrado na **Figura A–14**, que não corresponde àquilo que se esperava obter. Quer dizer, 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. O resultado inesperado é advindo do fato de os octetos sobre os quais incide a operação XOR estarem desalinhados. Para alinhar esses octetos, usa-se um deslocamento à direita a cada um deles de modo que eles ocupem os bits de menor ordem, como mostra a **Figura A–15**. Por sua vez, a **Figura A–16** mostra como, agora, o resultado desejado é obtido.

É importante notar o uso de parênteses nas expressões `(valor & mascara2) >> 2*8` e `(valor & mascara2) >> 1*8` na **Figura A–15**. Esses parênteses são de fato necessários porque o operador `>>` tem precedência maior do que o operador **&**. Note ainda que o valor resultante das operações XOR cabem 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.

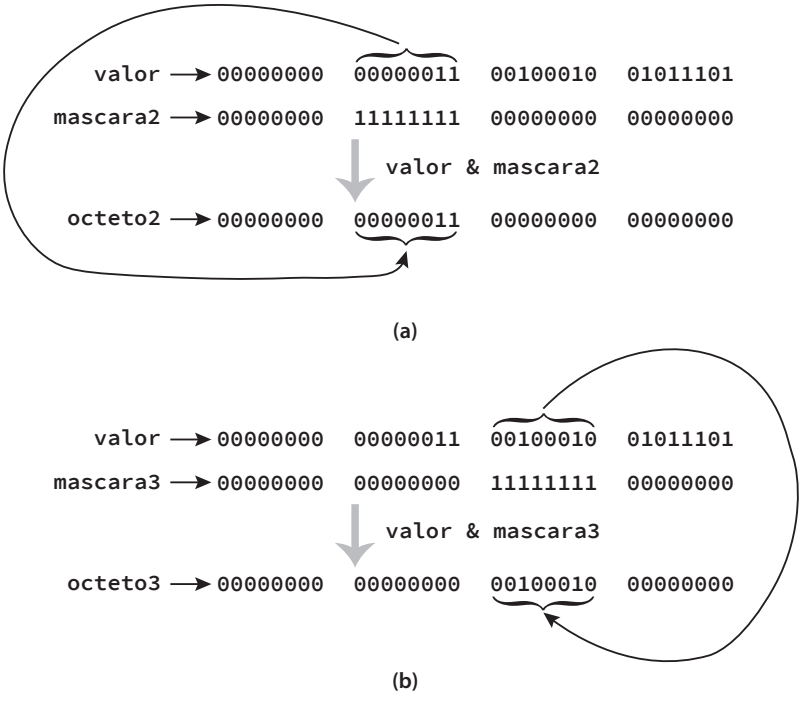

Figura A–13: Dispersão por Mistura: Extração de Bits

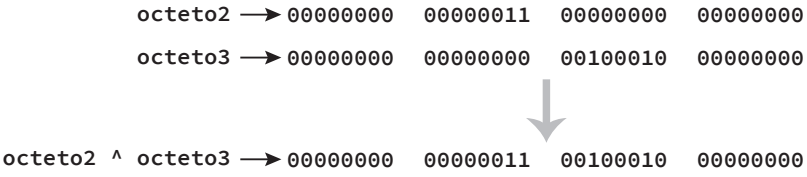

Figura A–14: Dispersão por Mistura: Resultado Incorreto

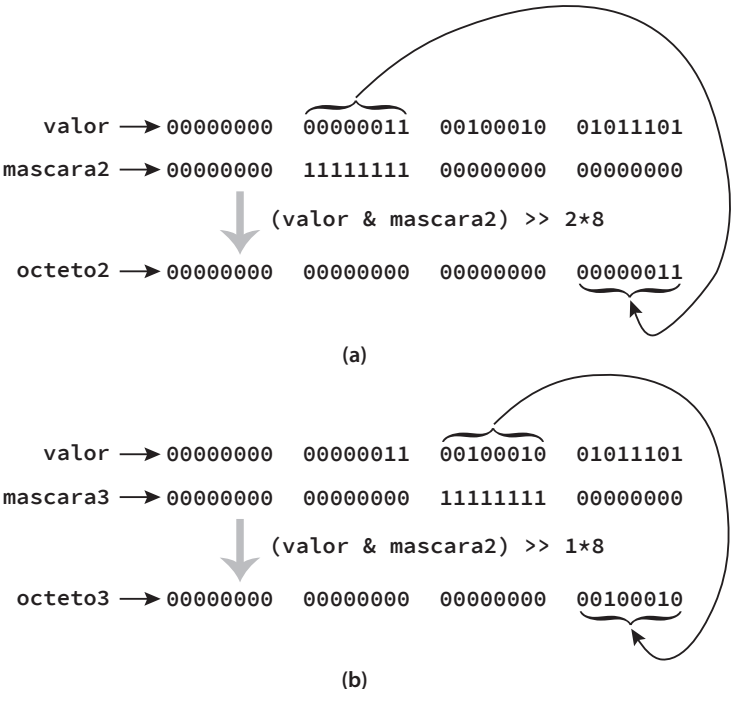

FIGURA A–15: DISPERSÃO POR MISTURA: EXTRAÇÃO DE BITS COM DESLOCAMENTO

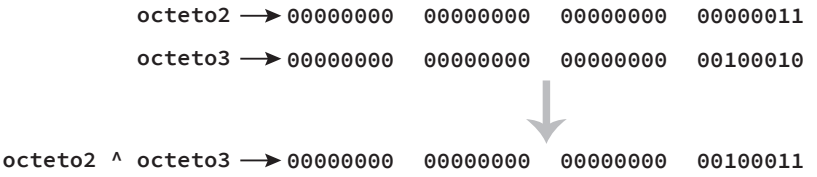

FIGURA A–16: DISPERSÃO POR MISTURA: RESULTADO CORRETO

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

```
int DispersaoMisturaJunior(int chave, int tamTabela)
{
   unsigned mascara[4] = { 0xFF000000,  0x00FF0000,
                           0x0000FF00, 0x000000FF,
                         };
   char     octetos[4] = { (chave & mascara[0]) >> 3*8,
                           (chave & mascara[1]) >> 2*8,
                           (chave & mascara[2]) >> 1*8,
                           (chave & mascara[3]) >> 0*8
                         };
   int      i, xor1, xor2;

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

   return (xor1 ^ xor2)/tamTabela;
}
```

O array `mascara[]` contém os valores a ser usados para extração das sequências de bits. A **Tabela A–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 A–5: Máscaras em Formato Hexadecimal e Binário

O array `octetos[]` armazena os bytes 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.

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

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

      printf("\n\n>>> Mascara: %X", mascara);
      printf("\n>>> Octeto %d: ", i + 1);
      RepresentacaoBinaria(octetos[i]);

      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.