

Conceitos e Técnicas de Programação

Prof. David Déharbe

2011

Sumário

0.1	Bits e bytes	1
0.2	Representação binária de números	2
0.2.1	Representação de números inteiros positivos	2
0.2.2	Representação de números inteiros com sinal	3
0.3	Os tipos inteiros da linguagem C	3
0.4	Manipulação de bits	5
0.5	Composição bit a bit	6
0.5.1	Negação	7
0.5.2	Conjunção	7
0.5.3	Disjunção	8
0.5.4	Disjunção exclusiva	9

Lista de Figuras

1	Dados sobre a evolução dos micro-processadores	1
2	Definição das operações de composição de bits da linguagem C.	7

Escovando bits

0.1 Bits e bytes

Computadores são máquinas de calcular construídas com uma enorme quantidade de componentes eletrônicos extremamente simples. Esses componentes eletrônicos são combinados de forma a utilizar correntes elétricas para representar e manipular informações. O *transistor* é provavelmente o componente elementar o mais usado: ele possui pelo menos três conexões tais que, quando uma corrente é aplicada a uma delas, uma corrente flui entre o outro par de conexões. A figura 1 providencia dados acerca da evolução da quantidade e do tamanho de transistores.

ano	modelo	núm. transistores	processo
1970	Intel 4004	2.300	10^{-5}m
1980	Motorola 68000	40.000	$3,5 \times 10^{-6}\text{m}$
1982	Intel 80286	134.000	$1,5 \times 10^{-6}\text{m}$
1999	Intel Pentium III Coppermine	29.000.000	$1,8 \times 10^{-7}\text{m}$
2006	STI Cell	241.000.000	$9 \times 10^{-8}\text{m}$
2011	Intel Core i7 Extreme Edition	1.300.000.000	$3,2 \times 10^{-8}\text{m}$

Figura 1: Dados sobre a evolução dos micro-processadores

Programar uma máquina da complexidade dos micro-processadores não pode ser realizado efetivamente se algumas camadas de abstração não fossem aplicadas.

A primeira abstração é a *digitalização*. Assim, a unidade de informação básica não é a voltagem ou a intensidade de uma corrente elétrica, e sim o *bit*. Um bit pode assumir dois valores: zero ou um; diz-se que é um dígito binário (*bit* é a abreviação de *binary digit*).

Mesmo o bit é uma informação muito básica, e geralmente os micro-processadores não manipulam diretamente bits e sim agrupamentos de bits. É comum qualificar processadores com o tamanho dos agrupamentos de bits que eles podem manipular de uma vez só: 64 bits, por exemplo. Entre os diversos tamanhos possíveis, o número de 8 bits tem se tornado o mais comum, e é hoje chamado de octeto, ou *byte*. Um byte é composto de oito bits; cada bit dentro de um byte é situado em uma posição. São portanto oito posições, que denotaremos com índices de 7 a 0, da direita para a esquerda. Seja *b* um byte, denotaremos $b_7, b_6, b_5, b_4, b_3, b_2, b_1$ e b_0 seus bits. Eventualmente usaremos uma tabela horizontal para ilustrar o valor de um byte:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	1	0	0	1	1	0	1

Para um byte b , cada bit b_i pode assumir dois valores. Portanto o número total de valores possíveis para um byte é de $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^8 = 256$.

Veremos na sequência como podemos codificar números com um ou vários bytes, quais a relação entre bytes e diferentes tipos da linguagem C, e enfim operadores da linguagem C que permitem combinar e manipular agrupamentos de bits.

0.2 Representação binária de números

0.2.1 Representação de números inteiros positivos

Inicialmente, consideramos o caso da representação de um número binário com um único byte, ou seja oito bits.

Seja b um byte, com $b_7 \cdots b_0$ seus oito bits. Usando a representação posicional, o valor representado é:

$$\begin{aligned} \sum_{i=0}^7 b_i \times 2^i &= b_7 \times 2^7 + b_6 \times 2^6 + b_5 \times 2^5 + b_4 \times 2^4 + b_3 \times 2^3 + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0 \\ &= b_7 \times 128 + b_6 \times 64 + b_5 \times 32 + b_4 \times 16 + b_3 \times 8 + b_2 \times 4 + b_1 \times 2 + b_0 \times 1 \end{aligned}$$

Nesta representação, o bit b_7 tem o maior peso e é chamado de *bit mais significativo*; similarmente, o bit b_0 é chamado de *bit menos significativo*.

O menor número representável com essa codificação é quando todos os bits são iguais a zero, o qual é:

$$0 \times 128 + 0 \times 64 + 0 \times 32 + 0 \times 16 + 0 \times 8 + 0 \times 4 + 0 \times 2 + 0 \times 1 = 0$$

O maior número representável com essa codificação é quando todos os bits são iguais a um, o qual é:

$$1 \times 128 + 1 \times 64 + 1 \times 32 + 1 \times 16 + 1 \times 8 + 1 \times 4 + 1 \times 2 + 1 \times 1 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

Assim, com um byte, podemos representar todos os números inteiros positivos na faixa $[0; 255]$.

Exercício 0.1 Qual é o número com a seguinte representação binária?

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	1	0	0	1	1	0	1

Exercício 0.2 Qual a representação binária, de um byte, para representar os valores seguintes?

- 47;
- 5;

- 150;
- 234.

Exercício 0.3 Seja n um número inteiro, tal que $0 \leq n \leq 255$. Usando operações aritméticas como quociente e resto da divisão inteira é possível determinar o valor de cada um dos bits da representação binária de n .

Determine como calcular a representação binária, em um byte, de n . Comece determinando como calcular o bit mais significativo até o menos significativo. Escreva um texto que detalhe passo a passo como calcular cada um desses bits. Verifique seu procedimento com diferentes valores para n .

Escreva uma sub-rotina que, dada um valor inteiro entre 0 e 255, imprime sua representação binária de um byte.

Use esta rotina para escrever um programa que lê um número inteiro entre 0 e 255, e imprime a sua representação binária.

Para representar uma faixa maior de números é necessário empregar uma quantidade maior de bytes na representação de cada valor. Assim, é comum ver representações com quatro bytes, ou seja 32 bits. Seja b^3, b^2, b^1, b^0 quatro bytes usados para representar números inteiros positivos. O valor inteiro representando é igual a:

$$\underbrace{\left(\sum_{i=0}^7 b_i^3 \times 2^i \right)}_{b^3} \times 2^{24} + \underbrace{\left(\sum_{i=0}^7 b_i^2 \times 2^i \right)}_{b^2} \times 2^{16} + \underbrace{\left(\sum_{i=0}^7 b_i^1 \times 2^i \right)}_{b^1} \times 2^8 + \underbrace{\left(\sum_{i=0}^7 b_i^0 \times 2^i \right)}_{b^0} \times 2^0.$$

ou, de forma equivalente, se denotamos $b_7 \cdots b_0$ os bits de b^0 , $b_{15} \cdots b_8$ os bits de b^1 , $b_{23} \cdots b_{16}$ os bits de b^2 , e $b_{31} \cdots b_{24}$ os bits de b^3 :

$$\sum_{i=0}^{31} b_i \times 2^i.$$

Com 4 bytes, podemos representar $2^{4 \times 8} = 2^{32}$ valores diferentes. No caso de inteiros sem sinal, isto permite representar o intervalo de $[0; 2^{32} - 1]$.

0.2.2 Representação de números inteiros com sinal

Não entraremos em detalhes da representação binária dos números inteiros com sinal. Um bit é reservado para representar o sinal do número, e os demais bits são usados para a parte absoluta. No entanto, e embora a parte absoluta seja um inteiro positivo, a forma de codificação pode ser diferente daquela apresentada na seção 0.2.1.

0.3 Os tipos inteiros da linguagem C

A linguagem C possui vários tipos inteiros. Cada tipo representa valores dentro de uma certa faixa de números. Esses tipos variam de acordo com dois fatores:

- **Sinal:** Há tipos com sinal e sem sinal. Nos tipos com sinal, o primeiro bit da representação binária indica o sinal do número: positivo ou negativo. Nos tipos sem sinal, todos os valores representados são positivos (ou nulos).

Nos tipos sem sinal, o valor representado corresponde à codificação binária apresentada na seção anterior. Os tipos sem sinal são rotulados com a palavra **unsigned**.

Nos tipos com sinal, o valor representado utiliza uma codificação binária diferente (existe várias codificações possíveis). Por default, os tipos tem sinal, esta informação pode ser indicada explicitamente com a palavra **signed**.

- **Tamanho:** O número de bytes empregados para representar os valores varia de acordo com o tipo. Obviamente, quanto maior o tamanho, maior é a faixa de valores representada. Em C, há cinco indicadores de tamanho pré-definidos: **char**, **short**, **int**, **long**, **long long**.

É importante destacar que, para um mesmo tipo, o tamanho da sua representação pode variar de uma plataforma computacional para outra; o padrão da linguagem apenas define limites mínimos.

A tabela seguinte contem a totalidade dos tipos inteiros disponíveis, em ordem crescente de tamanho:

tamanho(*)	com sinal	sem sinal
1	char	unsigned char
2	short	unsigned short
4 ou 8	int	unsigned int
8	long	unsigned long
16	long long	unsigned long long

(*) valores típicos

A tabela seguinte fornece informações auxiliares quanto a estes tipos: qual a diretiva de formatação para as sub-rotinas da biblioteca padrão (como `printf` e `scanf`), e nomes para os valores máximos e mínimos (para os tipos com sinal) definidos no arquivo cabeçalho `limits.h`.

tipo	formato	valor máx.	valor mín.
char	%hhi	CHAR_MAX	CHAR_MIN
unsigned char	%hhu	UCHAR_MAX	
short	%hi	SHRT_MAX	SHRT_MIN
unsigned short	%hhu	USHRT_MAX	
int	%i	INT_MAX	INT_MIN
unsigned int	%u	UINT_MAX	
long	%li	LONG_MAX	LONG_MIN
unsigned long	%lu	ULONG_MAX	
long long	%lli	LLONG_MAX	LLONG_MIN
unsigned long long	%llu	ULLONG_MAX	

O operador C **sizeof** permite ao programador saber o tamanho da representação de um tipo. A expressão **sizeof(T)**, onde **T** é um tipo, é a quantidade de bytes usada para representar um valor do tipo **T**. O tipo do resultado deste operador é **unsigned long**.

De posse destas informações, estamos agora com os recursos para escrever um programa que imprime as características de todos os tipos inteiros da linguagem C:

```

1 #include <limits.h>
2 #include <stdio.h>
3
4 int main ()
5 {
6     printf("tipos inteiros com sinal\n");
7     printf(" signed char - size: %lu, min = %hhi, max = %hhi\n",
8           sizeof(signed char), SCHAR_MIN, SCHAR_MAX);
9     printf(" short int - size: %lu, min = %hi, max = %hi\n",
10          sizeof(short int), SHRT_MIN, SHRT_MAX);
11     printf(" int - size: %lu, min = %i, max = %i\n",
12          sizeof(int), INT_MIN, INT_MAX);
13     printf(" long int - size: %lu, min = %li, max = %li\n",
14          sizeof(long int), LONG_MIN, LONG_MAX);
15     printf(" long long int - size: %lu, min = %lli, max = %lli\n",
16          sizeof(long long int), LLONG_MIN, LLONG_MAX);
17
18     printf("tipos inteiros sem sinal\n");
19     printf(" unsigned char - size: %lu, max = %hhu\n",
20          sizeof(unsigned char), UCHAR_MAX);
21     printf(" unsigned short int - size: %lu, max = %hu\n",
22          sizeof(unsigned short), USHRT_MAX);
23     printf(" unsigned int - size: %lu, max = %u\n",
24          sizeof(unsigned int), UINT_MAX);
25     printf(" unsigned long int - size: %lu, max = %lu\n",
26          sizeof(unsigned long int), ULONG_MAX);
27     printf(" unsigned long long int - size: %lu, max = %llu\n",
28          sizeof(unsigned long long int), ULLONG_MAX);
29
30     return 0;
31 }

```

0.4 Manipulação de bits

Vimos na seção 0.1 que os valores manipulados pelos programas de computadores podem ser vistos como agrupamentos de bits compostos por um ou mais bytes. No caso de valores inteiros sem sinal há uma relação direta entre a codificação binária e a representação posicional na base 2. A linguagem C disponibiliza vários tipos inteiros sem sinal; podemos usar eles nos nossos programas para manipular conjuntos de bits.

A forma mais simples para definir um conjunto de bits é através da notação hexadecimal (notação posicional em base 16). Cada dígito hexadecimal corresponde a quatro bits. A tabela seguinte mostra a codificação em quatro bits de cada dígito hexadecimal.

0	0	0	0	0	8	1	0	0	0
1	0	0	0	1	9	1	0	0	1
2	0	0	1	0	A	1	0	1	0
3	0	0	1	1	B	1	0	1	1
4	0	1	0	0	C	1	1	0	0
5	0	1	0	1	D	1	1	0	1
6	0	1	1	0	E	1	1	1	0
7	0	1	1	1	F	1	1	1	1

Um número de dois dígitos hexadecimais pode ser codificado com exatamente um byte. Por exemplo, a seguinte declaração define variáveis que de diversos tamanhos e que têm todos os seus bits iguais a um.

```

1 unsigned char c = 0xFF;
2 unsigned short s = 0xFFFF;
3 unsigned int i = 0xFFFFFFFF; /* ou 0xFFFFFFFFFFFFFFFF se a arquitetura for 64 bits */
4 unsigned long long ll = 0xFFFFFFFFFFFFFFFF;

```

A linguagem C proporciona diversos operadores para manipular e compor a codificação binária de valores. Esses operadores têm efeito diretamente em nível de bits. Os operadores de composição são apresentados na seção 0.5.

Há dois operadores de manipulação de bits. Ambos realizam um *deslocamento* dos bits de um certo número de casas. O operador `>>` realiza um deslocamento para a direita, e o operador `<<` efetua um deslocamento para a esquerda.

O resultado da aplicação do operador de *deslocamento para a esquerda* é dada pela seguinte equação:

$$b_{m-1}b_{m-2}\cdots b_1b_0 \ll n = b_{m-1-n}b_{m-2-n}\cdots b_0 \underbrace{0\cdots 0}_{n \text{ vezes}}.$$

Seja m o número de bits de b . No resultado de $b \ll n$, os n bits mais a direita são 0, e todos os demais $m - n$ bits são resultado do deslocamento dos $m - n$ últimos bits de b de n posições para a esquerda. Por exemplo $10010011 \ll 2$ é 01001100 .

Esse operador possui as seguintes propriedades:

$$\begin{aligned} m \ll 0 &= m; \\ m \ll n \ll p &= m \ll (n + p); \\ m \ll n &= m \times 2^n, \text{ se } m \text{ for um inteiro sem sinal.} \end{aligned}$$

O resultado da aplicação do operador de *deslocamento para a direita* é dada pela seguinte equação:

$$b_{m-1}b_{m-2}\cdots b_1b_0 \gg n = \underbrace{0\cdots 0}_{n \text{ vezes}} b_{m-1}b_{m-2}\cdots b_n.$$

Seja m o número de bits de b . No resultado de $b \gg n$, os n bits mais a esquerda são 0, e todos os demais $m - n$ bits são resultado do deslocamento dos $m - n$ primeiros bits de b de n posições para a direita. Por exemplo $10010011 \gg 2$ é 00100100 .

Esse operador possui as seguintes propriedades:

$$\begin{aligned} m \gg 0 &= m; \\ m \gg n \gg p &= m \gg (n + p); \\ m \gg n &= \frac{m}{2^n}, \text{ se } m \text{ for um inteiro sem sinal.} \end{aligned}$$

0.5 Composição bit a bit

Os operadores de composição combinam bit por bit seus arguments usando operações lógicas (ver figura 2).

Negação		Conjunção		
argumento	resultado	argumento 1	argumento 2	resultado
0	1	0	0	0
1	0	0	1	0
		1	0	0
		1	1	1

Disjunção			Disjunção exclusiva		
argumento 1	argumento 2	resultado	argumento 1	argumento 2	resultado
0	0	0	0	0	0
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

Figura 2: Definição das operações de composição de bits da linguagem C.

0.5.1 Negação

O operador `~` (til) é o operador de negação bit a bit. É um operador unário e ele inverte todos os bits de seu argumento. O programa seguinte ilustra o uso deste operador: um valor inteiro na base dez é lido. Em seguida o programa imprime uma linha com a representação hexadecimal com oito dígitos deste número e da sua negação bit a bit. As duas representações estão prefixadas com `0x` e complementadas com zeros a esquerda caso haja menos de oito dígitos significativos.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main ()
5 {
6     unsigned int n;
7     scanf("%u", &n);
8     printf("~ 0x%08x = 0x%08x\n", n, ~n);
9     return EXIT_SUCCESS;
10 }
```

Exercício 0.4 A codificação binária da expressão `1 << i` tem todos os bits iguais a 0, com exceção do bit na posição `i`, que é igual a 1.

Escreva uma expressão cuja codificação binária tem todos os bits iguais a 1, com exceção do bit na posição `i`, que deve ser igual a 0.

0.5.2 Conjunção

O operador `&` (e comercial) é o operador de conjunção em nível de bits. É um operador binário, que calcula a conjunção bit por bit de seus argumentos. O i -ésimo bit do resultado é a conjunção dos i -ésimos bits de seus argumentos.

O programa seguinte lê dois números inteiros. Em seguida imprime uma linha com a representação hexadecimal dos dois números lidos e da conjunção bit a bit deles.

```

1 #include <stdio.h>
```

```

2 #include <stdlib.h>
3
4 int main ()
5 {
6     unsigned int n, m;
7     scanf("%u %u", &n, &m);
8     printf("0x%08x & 0x%08x = 0x%08x\n", n, m, n&m);
9     return EXIT_SUCCESS;
10 }

```

O programa seguinte lê dois números inteiros n e m , e imprime uma mensagem informando se o bit na posição m de n é 0 ou 1.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main ()
5 {
6     unsigned int n, m;
7     printf("numero: ");
8     scanf("%u", &n);
9     printf("bit: ");
10    scanf("%u", &m);
11    if (n & (1 << m))
12        printf("O bit %u de %u é 1.\n", m, n);
13    else
14        printf("O bit %u de %u é 0.\n", m, n);
15    return EXIT_SUCCESS;
16 }

```

Exercício 0.5 Com base o programa anterior, defina uma sub-rotina cuja interface é `int test_bit(unsigned int n, unsigned int m)`; e que testa se o bit m do número n é 1 ou 0.

Exercício 0.6 Defina uma sub-rotina cuja interface é `void print_bit(unsigned int n, unsigned int m)`; e que imprime o bit na posição m do número n .

Exercício 0.7 Defina uma sub-rotina cuja interface é `void print_unsigned_int_bit(unsigned int n)`; imprime a codificação binária de n .

Exercício 0.8 Defina uma sub-rotina Com base o programa anterior, defina uma sub-rotina cuja interface é `unsigned int unset_bit(unsigned int n, unsigned int m)`; e é tal que:

- o bit na posição m do resultado deve ser 0.
- para cada i diferente de m , o bit na posição i do resultado deve ser igual ao bit na posição i de n .

0.5.3 Disjunção

O operador `|` (barra vertical) é o operador de disjunção sobre bits. É um operador binário, que calcula a disjunção bit por bit de seus argumentos. O i -ésimo bit do resultado é a disjunção dos i -ésimos bits de seus argumentos.

O programa seguinte lê dois números inteiros e , em seguida, imprime uma linha com a representação hexadecimal dos dois números lidos e da disjunção bit a bit deles.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main ()
5 {
6     unsigned int n, m;
7     scanf("%u %u", &n, &m);
8     printf("0x%08x | 0x%08x = 0x%08x\n", n, m, n | m);
9     return EXIT_SUCCESS;
10 }
```

Exercício 0.9 Defina uma sub-rotina cuja interface é `unsigned int set_bit(unsigned int n, unsigned int m)`; e é tal que:

- o bit na posição m do resultado deve ser 1.
- para cada i diferente de m , o bit na posição i do resultado deve ser igual ao bit na posição i de n .

0.5.4 Disjunção exclusiva

O operador \wedge (acento circunflexo) é o operador de disjunção exclusiva sobre bits. É um operador binário, que calcula a disjunção exclusiva bit por bit de seus argumentos. O i -ésimo bit do resultado é a disjunção exclusiva dos i -ésimos bits de seus argumentos.

O programa seguinte lê dois números inteiros e , em seguida, imprime uma linha com a representação hexadecimal dos dois números lidos e da disjunção exclusiva bit a bit deles.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main ()
5 {
6     unsigned int n, m;
7     scanf("%u %u", &n, &m);
8     printf("0x%08x | 0x%08x = 0x%08x\n", n, m, n | m);
9     return EXIT_SUCCESS;
10 }
```

Exercício 0.10 Defina uma sub-rotina cuja interface é `unsigned int eq_bit(unsigned int n, unsigned int m)`; e é tal que:

- o bit na posição i do resultado deve ser 1 se os bits na posição i de n e de m são iguais.
- o bit na posição i do resultado deve ser 0 se os bits na posição i de n e de m são diferentes.