

Manipulação de bits

Operações Binárias

Prof. Edson Alves - UnB/FGA

2018

1. Operações Binárias

Operações Binárias

Operações bit a bit

- As operações *bit a bit* se comportam da mesma maneira do que suas equivalentes da lógica booleana, considerando o valor 0 (zero) como falso e 1 (um) como verdadeiro
- As representações binárias dos operandos devem estar alinhadas (com o mesmo número de dígitos) antes da operação (zeros à esquerda podem ser necessários)
- A operação $\&$ (e, *and*) resulta em verdadeiro somente quando os dois valores são verdadeiros
- A operação $|$ (ou, *or*) resulta em falso somente quando os dois valores são falsos
- A operação \wedge (ou exclusivo, *xor*) resulta em falso somente quando ambos valores são iguais
- A operação \sim (negação, *not*) é unária, e inverte todos os *bits* do operando

Visualização das operações bit a bit

$$\begin{array}{rcl} 10100111 & (167) \\ \& 01101110 & (110) \\ \hline 00100110 & (38) \end{array}$$

$$\begin{array}{rcl} 10100111 & (167) \\ | 01101110 & (110) \\ \hline 11101111 & (239) \end{array}$$

$$\begin{array}{rcl} 10100111 & (167) \\ \wedge 01101110 & (110) \\ \hline 11001001 & (201) \end{array}$$

$$\begin{array}{rcl} \sim 01101110 & (110) \\ \hline 10010001 & (145) \end{array}$$

Deslocamentos binários

- O operador \ll (deslocamento à esquerda, *left shift*) adiciona o número indicado (k) de zeros à esquerda do número
- A operação equivale à uma multiplicação por 2^k , levando em conta um possível *overflow*
- O operador \gg (deslocamento à direita, *right shift*) adiciona o número indicado (k) de zeros à direita do número
- A mesma quantidade de *bits* à direita são desprezados
- Se o sinal é propagado, a operação é denominada deslocamento à direita aritmético; caso contrário, deslocamento à direita binário
- No caso do operador aritmético, a operação equivale à uma divisão inteira euclidiana por 2^k
- Em C/C++, o operador \gg é aritmético, e a divisão inteira ($/$) não é euclidiana (é a divisão de menor resto)

Exemplo de deslocamentos binários

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 int main()
6 {
7     int a = 12345, b = -98;
8
9     cout << (a << 2) << endl;    // 49380
10    cout << (a >> 3) << endl;    // 1543
11
12    cout << (b << 2) << endl;    // -392
13    cout << (b >> 3) << endl;    // -13
14
15    cout << b / 8 << endl;        // -12
16
17    return 0;
18 }
```

Máscaras binárias

- Uma máscara binária é um padrão binário que permite a localização, extração ou alteração de determinados *bits* de uma representação binária
- A máscara $(1 \ll k)$ corresponde a todos os *bits* iguais a zero, exceto o k -ésimo *bit*, que é igual a um
- Esta máscara permite a leitura do k -ésimo *bit* de um número através do operador $\&$
- Esta mesma máscara permite ligar o k -ésimo *bit* de um número através do operador $|$
- A negação desta máscara $(\sim(1 \ll k))$ permite desligar o k -ésimo *bit* de um número com o operador $\&$
- A máscara $((1 \ll k) - 1)$ permite a extração dos k *bits* menos significativos de um número através do operador $\&$

Exemplo de uso de máscaras binárias

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 unsigned long rotate_right(unsigned long n, int k)
6 {
7     unsigned long left = (n >> k);
8     unsigned right = n & ((1 << k) - 1);
9
10    return right << (8*sizeof(unsigned long) - k) | left;
11 }
12
13 int main() {
14     unsigned long n = 0x12345678;
15
16     printf("0x%08lx\n", rotate_right(n, 8)); // 0x78123456
17     printf("0x%08lx\n", rotate_right(n, 16)); // 0x56781234
18     printf("0x%08lx\n", rotate_right(n, 3)); // 0x02468acf
19
20     return 0;
21 }
```

Bit menos significativo

- O *bit* menos significativo (*least significant bit* – *LSB*) de um inteiro n pode ser extraído em $O(1)$
- Basta fazer a conjunção de n com seu simétrico $-n$
- Em termos de código, $\text{LSB}(n) = n \ \& \ -n$
- É possível desligar o LSB com a expressão $(n \ \& \ \sim(1 \ll \text{LSB}(n)))$
- Porém a expressão $\text{CLSB}(n) = n \ \& \ (n - 1)$ gera o mesmo resultado usando uma sintaxe mais simples e eficiente
- A rotina $\text{CLSB}(n)$ pode ser usada para contar o número de *bits* ligados de n em $O(\log n)$

Exemplo de rotinas com LSB em C++

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 int LSB(int n) { return n & -n; }
6 int CLSB(int n) { return n & (n - 1); }
7
8 int bit_count(int n)
9 {
10     int count = 0;
11
12     while (n)
13     {
14         ++count;
15         n &= (n - 1);
16     }
17
18     return count;
19 }
20
```

Exemplo de rotinas com LSB em C++

```
21 int main()
22 {
23     int n = 14;
24
25     cout << LSB(n) << '\n';           // 2
26     cout << CLSB(n) << '\n';         // 12
27
28     cout << bit_count(n) << '\n';     // 3
29 }
```

Funções do GCC

- O GCC oferece uma série de funções de baixo nível para manipulação binária
- A função `__builtin_popcount(x)` retorna o número de *bits* ligados de x
- A função `__builtin_clz(x)` o número de zeros à esquerda na representação binária de x (*clz – count leading zeroes*)
- A função `__builtin_ctz(x)` o número de zeros à direita na representação binária de x (*ctz – count trailing zeroes*)
- As duas funções anteriores tem comportamento indefinido se x é igual a zero
- A função `__builtin_ffs(x)` retorna 1 mais o índice do *bit* menos significativo de x , ou zero, se x é igual a zero

Exemplo de uso das funções do GCC

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 int main()
6 {
7     int x = 1968;          // 123 x 16 = 11110110000
8
9     cout << __builtin_popcount(x) << '\n';      // 6
10    cout << __builtin_ffs(x) << '\n';           // 5
11    cout << __builtin_clz(x) << '\n';           // 21
12    cout << __builtin_ctz(x) << '\n';           // 4
13
14    return 0;
15 }
```

1. **HALIM**, Felix; **HALIM**, Steve. *Competitive Programming 3*, 2010.
2. **LAAKSONEN**, Antti. *Competitive Programmer's Handbook*, 2018.
3. **SKIENA**, Steven S.; **REVILLA**, Miguel A. *Programming Challenges*, 2003.