Juggling Bits

Marco Gallotta

28 February, 2009

Introduction

- Most useful optimisations are high-level
- Bit manipulation is one of the most effective low-level optimisations
- Potential order-of-magnitude improvement in speed and size
- Can simplify code

Outline

1. Introduction
2. Sets
3. Bit Tricks
4. Bitset
5. Sample Problems

Truth Table

- Truth table for `!` (not), `&&` (and), `||` (or) and `^` (xor):

A	B	!A	A && B	A || B	A ^B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

- Bit-wise operations do the same, but operate on each bit separately
- If A is 1100 and B is 1001, then:
  - `~A` = 11110011 (assuming A is one byte)
  - `A & B` = 1000
  - `A | B` = 1101
  - `A ^B` = 0101

Truth Table

- Truth table for `!` (not), `&&` (and), `||` (or) and `^` (xor):

A	B	!A	A && B	A || B	A ^B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

- Bit-wise operations do the same, but operate on each bit separately
- If A is 1100 and B is 1001, then:
  - `~A` = 11110011 (assuming A is one byte)
  - `A & B` = 1000
  - `A | B` = 1101
  - `A ^B` = 0101

Shift Operators

- Two more operators: `A « B` and `A » B`
- Shift all bits in A and shifts them B positions to the left («) / right (»)
- Non-negatives are padded with zeros
- Equivalent to multiplyication («) / integer division (») by 2^B
- Most common use is `1 « X`, which is a number with only bit X set

Bit-Sets

- We can use an integer to represent a subset of a set of up to 32 values
- A 1 bit represents a member in the subset, a 0 bit a member that is absent
- We have the following simple operations on subsets:

Union	`A	B`	Intersection	`A & B`
Subtraction	`A & ~B`	Negation	`ALL_BITS ^A`
Set bit	`A |= 1 « bit`	Clear bit	`A &= ~(1 « bit)`
Test bit	`(A & 1 « bit) != 0`		

Bit-Sets

- We can use an integer to represent a subset of a set of up to 32 values
- A 1 bit represents a member in the subset, a 0 bit a member that is absent
- We have the following simple operations on subsets:

Union	`A	B`	Intersection	`A & B`
Subtraction	`A & ~B`	Negation	`ALL_BITS ^A`
Set bit	`A |= 1 « bit`	Clear bit	`A &= ~(1 « bit)`
Test bit	`(A & 1 « bit) != 0`		

All Subsets

- Every N-bit value represents some subset of an N-element set
- Easy to iterate over all subsets
- The bit representation of a subset is less than that of the set
- `i = (i - 1) & A` to iterate to next subset of the subset A

All Subsets

- Every N-bit value represents some subset of an N-element set
- Easy to iterate over all subsets
- The bit representation of a subset is less than that of the set
- `i = (i - 1) & A` to iterate to next subset of the subset A

Extracting Bits

- Value of lowest bit: `x & ~(x - 1)`
- Index of highest/lowest bit: looping requires only two iterations on average
- GCC built-in functions:
  - `__builtin_ctz` (count trailing zeros)
  - `__builtin_clz` (count leading zeros)
  - *Undefined for zero*
- Check if number is a power of 2: clear lowest bit and check if result is 0
- `__builtin_popcount` counts number of bits set

Extracting Bits

- Value of lowest bit: `x & ~(x - 1)`
- Index of highest/lowest bit: looping requires only two iterations on average
- GCC built-in functions:
  - `__builtin_ctz` (count trailing zeros)
  - `__builtin_clz` (count leading zeros)
  - *Undefined for zero*
- Check if number is a power of 2: clear lowest bit and check if result is 0
- `__builtin_popcount` counts number of bits set

Extracting Bits

- Value of lowest bit: `x & ~(x - 1)`
- Index of highest/lowest bit: looping requires only two iterations on average
- GCC built-in functions:
  - `__builtin_ctz` (count trailing zeros)
  - `__builtin_clz` (count leading zeros)
  - *Undefined for zero*
- Check if number is a power of 2: clear lowest bit and check if result is 0
- `__builtin_popcount` counts number of bits set

Gotchas

- Very easy to make mistakes with bits
- `A << B` and `A >> B` use only the last 5 bits of `B`
    - Shifting by 32 bits does nothing!
- `&` and `|` operators have lower precedence than comparison operators

Bitset

- STL offers a conveniant data structure, `bitset<N>` in header `bitset`
- Optimised for space: each element occupies only one bit
- Advantages:
  - Easier than array of integers when more than 64 bits required
  - Some handy methods
- Disadvantages:
  - Need to know size in advance: template parameter `N`
  - No iterators: `++` and `--`
- Operators: `&`, `|`, `^`, «, » and their `&=` equivalents
- Operators: `~`, `==`, `!=` and `[]` to access a bit
- Methods: `set`, `flip`, `any`, `none`, `to_ulong`
- Documentation: `http://www.sgi.com/tech/stl/bitset.html`

Bitset

- STL offers a conveniant data structure, bitset<N> in header bitset
- Optimised for space: each element occupies only one bit
- Advantages:
  - Easier than array of integers when more than 64 bits required
  - Some handy methods
- Disadvantages:
  - Need to know size in advance: template parameter N
  - No iterators: ++ and --
- Operators: &, |, ^, «, » and their &= equivalents
- Operators: ~, ==, != and [] to access a bit
- Methods: set, flip, any, none, to_ulong
- Documentation: http://www.sgi.com/tech/stl/bitset.html

Bitset

- STL offers a conveniant data structure, `bitset<N>` in header `bitset`
- Optimised for space: each element occupies only one bit
- Advantages:
  - Easier than array of integers when more than 64 bits required
  - Some handy methods
- Disadvantages:
  - Need to know size in advance: template parameter `N`
  - No iterators: `++` and `--`
- Operators: `&`, `|`, `^`, «, » and their `&=` equivalents
- Operators: `~`, `==`, `!=` and `[]` to access a bit
- Methods: `set`, `flip`, `any`, `none`, `to_ulong`
- Documentation: `http://www.sgi.com/tech/stl/bitset.html`

Bitset

- STL offers a conveniant data structure, `bitset<N>` in header `bitset`
- Optimised for space: each element occupies only one bit
- Advantages:
  - Easier than array of integers when more than 64 bits required
  - Some handy methods
- Disadvantages:
  - Need to know size in advance: template parameter `N`
  - No iterators: `++` and `--`
- Operators: `&`, `|`, `^`, «, » and their `&=` equivalents
- Operators: `~`, `==`, `!=` and `[]` to access a bit
- Methods: `set`, `flip`, `any`, `none`, `to_ulong`
- Documentation: `http://www.sgi.com/tech/stl/bitset.html`

Example

```
#include <bitset>
#include <iostream>
using namespace std;
int main() {
  bitset<8> a(100ul), b(string("1001110"));
  cout << a << " " << b << endl;
  cout << (a&b) << " " << (a|b) << " " << (a^b)
      << endl;
  a[0] = 1; cout << a << endl;
}
```

produces:

```
01100100 01001110
01000100 01101110 00101010
```

Sample Problems

- Bit-sets and bit count: `http://www.topcoder.com/stat?c=problem_statement&pm=6725&rd=10100`
- All subsets: `http://www.topcoder.com/stat?c=problem_statement&pm=6095&rd=9917`
- Bit-sets with trick iteration: `http://www.topcoder.com/stat?c=problem_statement&pm=6475&rd=9988`
- No adjacent bits: `http://www.topcoder.com/stat?c=problem_statement&pm=6400&rd=10000`

Questions

?