Побитовые операции

Побитовые операции (англ. *bitwise operations*) — операции, производимые над цепочками битов. Выделяют два типа побитовых операций: логические операции и побитовые сдвиги.

Содержание

Принцип работы

Логические побитовые операции

Битовые операторы И , ИЛИ , НЕ и исключающее ИЛИ используют те же таблицы истинности, что и их логические эквиваленты.

Побитовое И

Побитовое И используется для выключения битов. Любой бит, установленный в , вызывает установку соответствующего бита результата также в .

&	11001010 11100010
	11000010

Побитовое ИЛИ

Побитовое ИЛИ используется для включения битов. Любой бит, установленный в , вызывает установку соответствующего бита результата также в .

|	11001010 11100010
	11101010

Побитовое НЕ

Побитовое НЕ инвертирует состояние каждого бита исходной переменной.

~	11001010
	00110101

Побитовое исключающее ИЛИ

Исключающее ИЛИ устанавливает значение бита результата в , если значения в соответствующих битах исходных переменных различны.

^	11001010 11100010
	00101000

Побитовые сдвиги

Операторы сдвига и сдвигают биты в переменной влево или вправо на указанное число. При этом на освободившиеся позиции устанавливаются нули (кроме сдвига вправо отрицательного числа, в этом случае на свободные позиции устанавливаются единицы, так как числа представляются в двоичном дополнительном коде и необходимо поддерживать знаковый бит).

Сдвиг влево может применяться для умножения числа на два, сдвиг вправо — для деления.

```
x = 7         // 00000111 (7)
x = x >> 1    // 00000011 (3)
x = x << 1    // 00000110 (6)
x = x << 5    // 11000000 (-64)
x = x >> 2    // 11110000 (-16)
```

В языке программирования Java существует также оператор беззнакового битового сдвига вправо . При использовании этого оператора на освободившиеся позиции всегда устанавливаются нули.

```
x = 7          // 00000111 (7)
x = x << 5     // 11100000 (-32)
x = x >>> 2    // 00111000 (56)
```

Применение

Сложные операции

Определение знака числа

Пусть дано число . Поскольку при сдвиге вправо на освобождающиеся позиции устанавливается бит знака, знак числа можно определить, выполнив сдвиг вправо на всю длину переменной:

```
int32 getSign(x: int32):
    if x != 0:
        mask = 1
    else:
        mask = 0

    return mask | (x >> 31)    // результатом будет -1, 0, или +1
                               // для отрицательного, равного нулю и положительного числа x соответственно
```

Используя побитовые операции можно также узнать, различны ли знаки двух переменных и . Если числа имеют различный знак, то результат операции XOR, произведенной над их знаковыми битами, будет единицей. Поэтому неравенство будет верно в том случае, если числа и разного знака.

Вычисление модуля числа без использования условного оператора

Пусть дано число . Если положительно, то , и . В случае, если отрицательно, . Тогда получается, что мы работаем с числом так, как будто оно представлено в коде со сдвигом с тем отличием, что у нас знаковый бит принимает значение для отрицательных чисел, а — для положительных.

```
int32 abs1(x: int32):
    mask = x >> 31
    return (x + mask) XOR mask

int32 abs2(x: int32):
    mask = x >> 31
    return (x + mask) XOR mask
```

Нахождение минимума и максимума из двух чисел без использования условного оператора

Этот способ корректен только если можно утверждать, что величина лежит между граничными значениями типа int.

Пусть даны числа и разрядности . Тогда если , то , а если , то . Выражение принимает значение , если , и , если .

```
int32 min(x, y: int32):
    return y + ((x - y) & ((x - y) >> 31))

int32 max(x, y: int32):
    return x - ((x - y) & ((x - y) >> 31))
```

Проверка на то, является ли число степенью двойки

Пусть дано число . Тогда, если результатом выражения является единица, то число — степень двойки.

Правая часть выражения будет равна единице, только если число равно или является степенью двойки. Если число является степенью двойки, то в двоичной системе счисления оно представляется следующим образом: , где — показатель степени. Соответственно, выражение будет иметь вид , и равно .

Операция логического И в данном выражении отсекает тот случай, когда и не является степенью двойки, но при этом правая часть равна единице.

Нахождение младшего единичного бита

Пусть дано число и необходимо узнать его младший единичный бит.

Применим к числу побитовое отрицание, чтобы инвертировать значения всех его бит, а затем прибавим к полученному числу единицу. У результата первая часть (до младшего единичного бита) не совпадает с исходным числом , а вторая часть совпадает. Применив побитовое И к этим двум числам, получим степень двойки, соответствующую младшему единичному биту исходного числа .

К такому же результату можно прийти, если сначала отнять от числа единицу, чтобы обнулить его младший единичный бит, а все последующие разряды обратить в , затем инвертировать результат и применить побитовое И с исходным числом .

Нахождение старшего единичного бита

Пусть дано число и необходимо узнать его старший единичный бит.

Рассмотрим некоторое число, представим его как , где — любое значение бита. Тогда, если совершить битовый сдвиг этого числа вправо на и произвести побитовое ИЛИ результата сдвига и исходного числа, мы получим результат . Если мы повторим эту последовательность действий над полученным числом, но устроим сдвиг на , то получим . При каждой следующей операции будем увеличивать модуль сдвига до следующей степени двойки. После некоторого количества таких операций (зависит от разрядности числа) мы получим число вида . Тогда результатом выполнения действий будет число, состоящее только из старшего бита исходного числа.

```
int32 greatestBit(x: int32):
    power = 1
    for i = 1 :
        x |= x >> power
        power <<= 1
    return x - (x >> 1)
```

Циклический сдвиг

Пусть дано число и надо совершить циклический сдвиг его битов на величину . Желаемый результат можно получить, если объединить числа, полученные при выполнении обычного битового сдвига в желаемую сторону на и в противоположном направлении на разность между разрядностью числа и величиной сдвига. Таким образом, мы сможем поменять местами начальную и конечную части числа.

```
int32 rotateLeft(x, d: int32):
    return (x << d) | (x >>> (32 - d))

int32 rotateRight(x, d: int32):
    return (x >>> d) | (x << (32 - d))
```

Подсчет количества единичных битов

Для подсчета количества единичных битов в числе можно воспользоваться следующим алгоритмом:

```
// Для чисел других разрядностей необходимо использовать соответствующие константы.
int16 setBitsNumber(x: int16):
    x = x - ((x >>> 1) & 0x5555)
    x = (x & 0x3333) + ((x >>> 2) & 0x3333)
```

```
    x = (x + (x >>> 4)) & 0x0F0F
    return (x * 0x0101) >>> 8
```

Поскольку равно , результатом операции является число, в котором все нечетные биты соответствуют нечетным битам числа . Аналогично, результатом операции является число, в котором все нечетные биты соответствуют четным битам . Четные биты результата в обоих случаях равны нулю.

Мысленно разобьем двоичную запись нашего числа на группы по бита. Результатом операции будет такое число, что если разбить его двоичную запись на группы по два бита, значение каждой группы соответствует количеству единичных битов в соответствующей паре битов числа .

Аналогично, число равно и операция , примененная к результату, полученному на первом этапе, выполняет подсчет количества единичных битов в блоках по . В свою очередь, число равно и операция позволяет подсчитать число единичных бит в блоках по .

Теперь необходимо просуммировать числа, записанные в блоках по битов, чтобы получить искомую величину. Это можно сделать, домножив результат на . Ответ на задачу будет находиться в первых восьми битах произведения. Выполнив сдвиг вправо на (для шестнадцатибитных чисел), мы получим долгожданный ответ.

Подведем итог:

```
int16 setBitsNumber(x: int16):
    x = (x & 0x5555) + ((x >>> 1) & 0x5555)
    x = (x & 0x3333) + ((x >>> 2) & 0x3333)
    x = (x & 0x0F0F) + ((x >>> 4) & 0x0F0F)
    return (x * 0x0101) >>> 8
```

Заметим, что операция равносильна операции , в чем легко убедиться, рассмотрев все числа из двух бит.

В свою очередь, операцию можно заменить на . Эта замена не повлияет на результат, так как максимальное значение в любой группе из четырех битов данного числа равно четырем, то есть требует только трех битов для записи, и выполнение суммирования не повлечет за собой переполнения и выхода за пределы четверок.

Таким образом, мы получили код, приведенный в начале раздела.

Разворот битов

Чтобы получить биты числа , записанные в обратном порядке, применим следующий алгоритм.

```
// Для чисел других разрядностей нужны соответствующие константы.
int16 reverseBits(x: int16):
    x = ((x & 0x5555) << 1) | ((x >>> 1) & 0x5555)  // Четные и нечетные биты поменялись местами.
    x = ((x & 0x3333) << 2) | ((x >>> 2) & 0x3333)  // Биты "перетасовываются" группами по два.
    x = ((x & 0x0F0F) << 4) | ((x >>> 4) & 0x0F0F)  // Биты "перетасовываются" группами по четыре.
    x = ((x & 0x00FF) << 8) | ((x >>> 8) & 0x00FF)  // Биты "перетасовываются" группами по восемь.
    return x
```

Более подробно про то, что за константы выбраны для данного алгоритма, можно прочитать в разделе подсчет количества единичных битов.

Применение для решения задач

Работа с битовыми масками

Для работы с подмножествами удобно использовать битовые маски. Применяя побитовые операции легко сделать следующее: найти дополнение , пересечение , объединение множеств, установить бит по номеру , снять бит по номеру .

Битовые маски используются, например, при решении некоторых задач[1] динамического программирования.

Алгоритм Флойда

Алгоритм Флойда–Уоршелла (англ. *the Floyd–Warshall algorithm*) — алгоритм для нахождения длин кратчайших путей между всеми парами вершин во взвешенном ориентированном графе. Работает корректно, если в графе нет циклов отрицательной величины, а если же такой цикл есть, позволяет найти хотя бы один такой цикл. Асимптотическая сложность алгоритма , также требует памяти.

Дерево Фенвика

Дерево Фенвика (англ. *Binary indexed tree*) — структура данных, которая может выполнять следующие операции:

- изменять значение любого элемента в массиве,
- выполнять некоторую ассоциативную, коммутативную, обратимую операцию на отрезке .

Данная структура требует памяти, а выполнение каждой операции происходит за .

Функция, позволяющая делать операции вставки и изменения элемента за , задается следующей формулой . Пусть дан массив . Деревом Фенвика называется массив из элементов: , где и — функция, которую мы определили ранее.

См. также

- Определение булевой функции
- Сумматор
- Триггеры

Примечания

1. Динамическое программирование по подмножествам (по маскам)

Источники информации

- Онлайн справочник программиста на C и C++ (http://www.c-cpp.ru/books/bitovye-operatory)
- Побитовые операторы (http://developer.alexanderklimov.ru/android/java/bitwise.php)
- Bit Twiddling Hacks by Sean Eron Anderson (https://graphics.stanford.edu/~seander/bithacks.html)
- Habrahabr — Алгоритмы поиска старшего бита (https://habrahabr.ru/post/93172/)
- STP's blog — Counting the number of set bits in an integer (https://yesteapea.wordpress.com/2013/03/03/counting-the-number-of-set-bits-in-an-integer/)

Источник — «http://neerc.ifmo.ru/wiki/index.php?title=Побитовые_операции&oldid=85773»

- Эта страница последний раз была отредактирована 4 сентября 2022 в 19:40.