Битовые операции

Методическая разработка к уроку

http://inf1.info – Планета информатики
Автор: С. Шапошникова (plustilino)

Во многих языках программирования допустимы логические операции над битами целых чисел. В отличие от обычных логических операций, результатом выполнения которых является логический тип данных, битовые логические операции просто изменяют целое число согласно определенным правилам. Точнее битовые операции изменяют отдельные биты двоичного представления числа, в результате чего изменяется его десятичное значение.

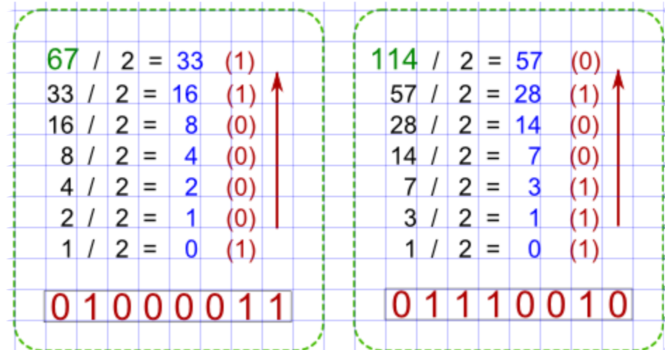
Например, в языке программирования Паскаль обычные логические операции и логические операции над битами обозначают с помощью одних и тех же ключевых слов: **`not, and, or, xor`**. Компилятор определяет, что имелось в виду в зависимости от контекста использования этих слов. Обычные логические операции объединяют два и более простых логических выражения. Например, **`(a > 0) and (c != b)`**, **`(c < a) or (not b)`** и т.п. В свою очередь побитовые логические операции выполняются исключительно над целыми числами (или переменными, которые их содержат). Например, **`a and b, a or 8, not 247`**.

Как понять побитовые операции

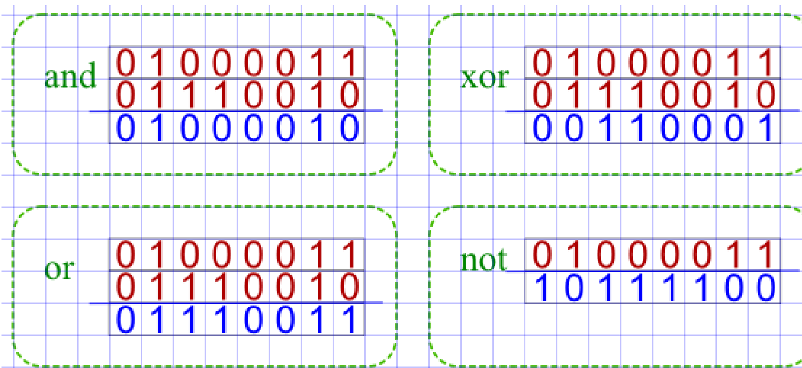
1. Переведем пару произвольных целых чисел до 256 (один байт) в двоичное представление.

$67_{10} = 0100\ 0011_2$

$114_{10} = 0111\ 0010_2$



2. Теперь расположим биты второго числа под соответствующими битами первого и выполним обычные логические операции к цифрам, стоящим в одинаковых разрядах первого и второго числа. Например, если в последнем (младшем) разряде одного числа стоит 1, а другого числа — 0, то логическая операция **`and`** вернет 0, а **`or`** вернет 1. Операцию **`not`** применим только к первому числу.



3. Переведем результат в десятичную систему счисления.

> $01000010 = 2^6 + 2^1 = 64 + 2 = 66$
>
> $01110011 = 2^6 + 2^5 + 2^4 + 2^1 + 2^0 = 64 + 32 + 16 + 2 + 1 = 115$
>
> $00110001 = 2^5 + 2^4 + 2^0 = 32 + 16 + 1 = 49$
>
> $10111100 = 2^7 + 2^5 + 2^4 + 2^3 + 2^2 = 128 + 32 + 16 + 8 + 4 = 188$

4. Итак, в результате побитовых логических операций получилось следующее:

> 67 and 114 = 66
>
> 67 or 114 = 115
>
> 67 xor 114 = 49
>
> not 67 = 188

Вот еще один пример выполнения логических операций над битами. Проверьте его правильность самостоятельно.

> 5 and 6 = 4
>
> 5 or 6 = 7
>
> 5 xor 6 = 3
>
> not 5 = 250

Зачем нужны побитовые логические операции

Глядя на результат побитовых операций, не сразу можно уловить закономерности в их результате. Поэтому непонятно, зачем нужны такие операции. Однако, они находят свое применение. В байтах не всегда хранятся числа. Байт или ячейка памяти может хранить набор флагов (установлен — сброшен), представляющих собой информацию о состоянии чего-либо. С помощью битовых логических операций можно проверить, какие биты в байте установлены в единицу, можно обнулить биты или, наоборот, установить в единицу. Также существует возможность сменить значения битов на противоположные.

Проверка битов

Проверка битов осуществляется с помощью битовой логической операции **`and`**.

Представим, что имеется байт памяти с неизвестным нам содержимым. Известно, что логическая операция **`and`** возвращает 1, если только оба операнда содержат 1. Если к неизвестному числу применить побитовое логическое умножение (операцию **`and`**) на число 255 (что в двоичном представлении 1111 1111), то в результате мы получим неизвестное число. Обнулятся те единицы двоичного представления числа 255, которые будут умножены на разряды неизвестного числа, содержащие 0. Например, пусть неизвестное число 38 (0010 0110), тогда проверка битов будет выглядеть так:

and	0?	0?	1?	0?	0?	1?	1?	0?
	1	1	1	1	1	1	1	1
	0	0	1	0	0	1	1	0

Другими словами, **`x and 255 = x`**.

Обнуление битов

Чтобы обнулить какой-либо бит числа, нужно его логически умножить на 0.

and
0 0 1 0 0 1 1 0
1 1 1 1 1 0 1 1
0 0 1 0 0 0 1 0

Обратим внимание на следующее:

> 1111 1110 = 254 = 255 - 1 = 255 - 2^0
>
> 1111 1101 = 253 = 255 - 2 = 255 - 2^1
>
> 1111 1011 = 251 = 255 - 4 = 255 - 2^2
>
> 1111 0111 = 247 = 255 - 8 = 255 - 2^3
>
> 1110 1111 = 239 = 255 - 16 = 255 - 2^4
>
> 1101 1111 = 223 = 255 - 32 = 255 - 2^5
>
> 1011 1111 = 191 = 255 - 64 = 255 - 2^6
>
> 0111 1111 = 127 = 255 - 128 = 255 - 2^7

Т.е. чтобы обнулить четвертый с конца бит числа **`x`**, надо его логически умножить на 247 или на (255 - 2^3).

Установка битов в единицу

Для установки битов в единицу используется побитовая логическая операция **`or`**. Если мы логически сложим двоичное представление числа **`x`** с 0000 0000, то получим само число **`x`**. Но вот если мы в каком-нибудь бите второго слагаемого напишем единицу, то в результате в этом бите будет стоять единица.

or
0 0 1 0 0 1 1 0
0 0 0 0 0 0 0 0
0 0 1 0 0 1 1 0

or
0 0 1 0 0 1 1 0
0 0 0 0 1 0 0 0
0 0 1 0 1 1 1 0

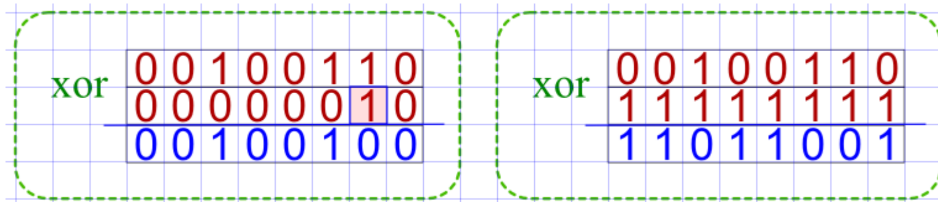
Отметим также, что:

> 0000 0001 = 2^0 = 1
>
> 0000 0010 = 2^1 = 2
>
> 0000 0100 = 2^2 = 4
>
> 0000 1000 = 2^3 = 8
>
> 0001 0000 = 2^4 = 16
>
> 0010 0000 = 2^5 = 32
>
> 0100 0000 = 2^6 = 64
>
> 1000 0000 = 2^7 = 128

Поэтому, например, чтобы установить второй по старшинству бит числа **`x`** в единицу, надо его логически сложить с 64 (**`x or 64`**).

Смена значений битов

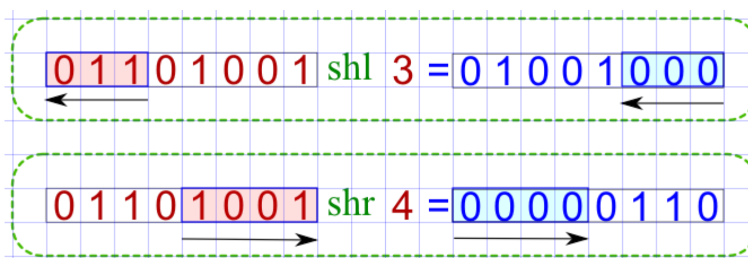
Для смены значений битов на противоположные используется битовая операция **`xor`**. Чтобы инвертировать определенный бит числа **`x`**, в такой же по разряду бит второго числа записывают единицу. Если же требуется инвертировать все биты числа **`x`**, то используют побитовую операцию исключающего ИЛИ (**`xor`**) с числом 255 (1111 1111).



Операции побитового циклического сдвига

Помимо побитовых логических операций во многих языках программирования предусмотрены битовые операции циклического сдвига влево или вправо. Например, в языке программирования Паскаль эти операции обозначаются **`shl`** (сдвиг влево) и **`shr`** (сдвиг вправо).

Первым операндом операций сдвига служит целое число, над которым выполняется операция. Во втором операнде указывается, на сколько позиций сдвигаются биты первого числа влево или вправо. Например, **`105 shl 3`** или **`105 shr 4`**. Число 105 в двоичном представлении имеет вид 0110 1001.



При сдвиге влево теряются старшие биты исходного числа, на их место становятся младшие. Освободившиеся младшие разряды заполняются нулями.

При сдвиге вправо теряются младшие биты исходного числа, на их место становятся старшие. Освободившиеся старшие разряды заполняются нулями, если исходное число было положительным.