Programming Languages

Bibhas Adhikari

IIT Kharagpur

November 12, 2020

Operation on Bits: why to use?

- Compression: Occasionally, you may want to implement a large number of Boolean variables, without using a lot of space. A 32-bit int can be used to store 32 Boolean variables. Normally, the minimum size for one Boolean variable is one byte. All types in C must have sizes that are multiples of bytes. However, only one bit is necessary to represent a Boolean value.
- Set operations: You can also use bits to represent elements of a (small) set. If a bit is 1, then element i is in the set, otherwise it's not. You can use bitwise AND to implement set intersection, bitwise OR to implement set union.
- Encryption: swapping the bits of a string for e.g. according to a predefined shared key will create an encrypted string
- Device communication: When reading input from a device - Each bit may indicate a status for the device or it may be one bit of control for that device

Operation on Bits

- C provides a host of operators specifically designed for performing operations on individual bits. For instance, 01100100 represents a string of eight binary digits, as called byte
- The rightmost bit of a byte is known as the least significant or low-order bit, whereas the leftmost bit is known as the most significant or high-order bit.
- If a string of bits represents an integer, the rightmost bit of the preceding byte represents 20 or 1, the bit immediately to its left represents 21 or 2, the next bit 22 or 4, and so on. For example, 01100100 represents the decimal number (integer)
  $2^2 + 2^5 + 2^6 = 4 + 32 + 64 = 100$.

Operation on Bits

- However, negative numbers is handled slightly differently. The leftmost bit represents the sign bit. If this bit is 1, the number is negative; otherwise, the bit is 0 and the number is positive
- A convenient way to convert a negative number from decimal to binary is to first add 1 to the value, express the absolute value of the result in binary, and then "complement" all the bits; that is, change all 1s to 0s and 0s to 1s. So, for example, to convert -5 to binary, 1 is added, which gives -4; 4 expressed in binary is 00000100, and complementing the bits produces 11111011
- To convert a negative number from binary back to decimal, first complement all of the bits, convert the result to decimal, change the sign of the result, and then subtract 1

Operation on Bits

- The largest positive number that can be stored into n bits is $2^{n-1}-1$ (Home Work)
- On most of today's processors, integers occupy four contiguous bytes, or 32 bits, in the computer's memory.The largest positive value that can, therefore, be stored into such an integer is $2^{31}-1$ or $2,147,483,647$, whereas the smallest negative number that can be stored is $-2,147,483,648$.

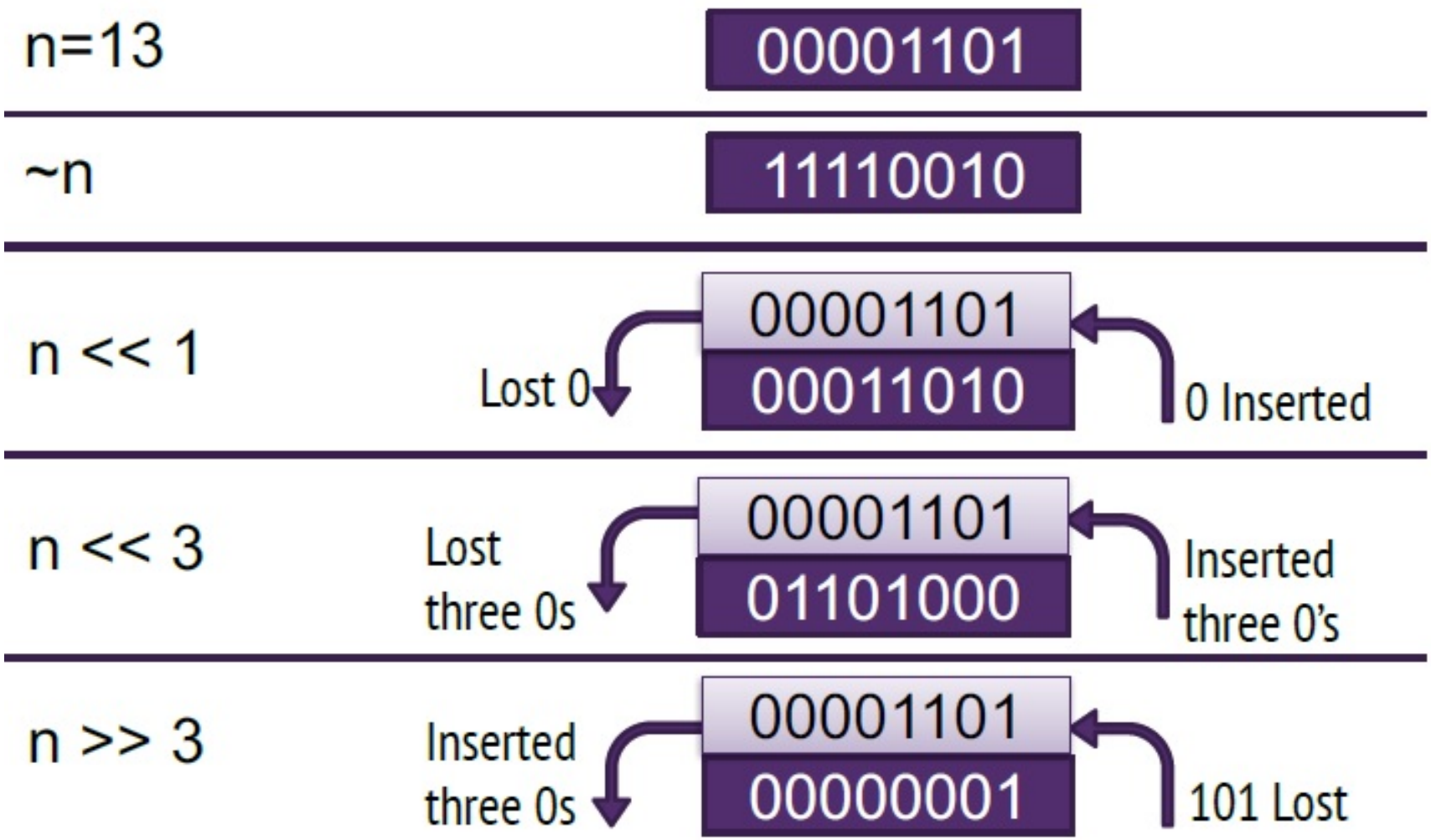
Symbol	Operation
&	Bitwise AND
|	Bitwise Inclusive-OR
^	Bitwise Exclusive-OR
~	Ones complement
<<	Left shift
>>	Right shift

- All the operators listed in the Table, with the exception of the ones complement operator~, are binary operators

Operation on Bits

&	bitwise AND	Compares its two operands bit by bit. The bits in the result are set to 1 if the corresponding bits in the two operands are *both* 1.
|	bitwise inclusive OR	Compares its two operands bit by bit. The bits in the result are set to 1 if *at least one* of the corresponding bits in the two operands is 1.
^	bitwise exclusive OR (also known as bitwise XOR)	Compares its two operands bit by bit. The bits in the result are set to 1 if the corresponding bits in the two operands are different.
<<	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from the right with 0 bits.
>>	right shift	Shifts the bits of the first operand right by the number of bits specified by the second operand; the method of filling from the left is machine dependent when the left operand is negative.
~	complement	All 0 bits are set to 1 and all 1 bits are set to 0.

Operation on Bits

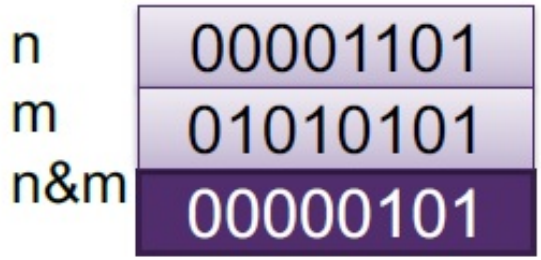


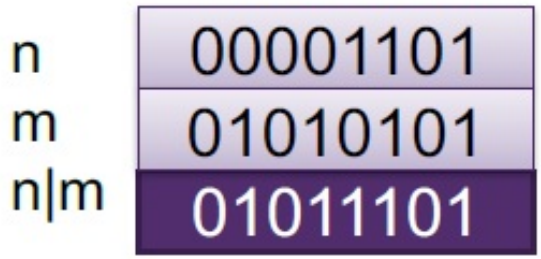
Operation on Bits

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
0	1	0
1	0	0
1	1	1

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
0	1	1
1	0	1
1	1	1

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
0	1	1
1	0	1
1	1	0





Operation on Bits

- Bit operations can be performed on any type of integer value in C and on characters, but cannot be performed on floating-point values.
- For example, 25 & 77 = 0000000000011001 & 0000000001001101 = 0000000000001001 = 9
- Recall that, the operation $x << n$ shifts the value of x left by n bits. Homework: Show that shifting left on an unsigned int by n bits, this is equivalent to multiplying by 2^n
- Shifting does not change value:

```
int x = 3;
int n = 2;
x << n;
printf("%d\n", x);
```

Operation on Bits

- How do you save the change after shifting?
  $x = x << n;$
  $printf(\text{“}\%d\backslash n\text{”}, x);$
- Otherwise use:
  $x <<= n;$
  $printf(\text{“}\%d\backslash n\text{”}, x);$

Operation on Bits: properties of XOR ($\oplus$)

XOR on Boolean values

- Let x, y be two bits, that is, $x, y \in \{0, 1\}$. Then $x \oplus y$ is true if exactly one of x and y is true
- $x_1 \oplus x_2 \oplus \ldots \oplus x_n$, $x_i \in \{0, 1\}$, $i = 1, \ldots, n$ is true if the number of variables with the value true is odd (and is false if the number of variables with the value true is even).

bitwise XOR or XOR on Boolean values

- $\mathbf{x} \oplus \mathbf{0} = \mathbf{x}$
- $\mathbf{x} \oplus \mathbf{1} = \tilde{}\mathbf{x}$
- $\mathbf{x} \oplus \mathbf{x} = \mathbf{0}$
- $(\mathbf{x} \oplus \mathbf{y}) \oplus \mathbf{z} = \mathbf{x} \oplus (\mathbf{y} \oplus \mathbf{z})$ (associative)
- $\mathbf{x} \oplus \mathbf{y} = \mathbf{y} \oplus \mathbf{y}$ (commutative)