Bitwise Operators

CS 253

Department of Computer Science
College of Engineering
Boise State University

August 25, 2017

Motivation

- Most programming tasks can be implemented using abstractions (e.g. representing data as an `int`, `long`, `char`, ...)
- However, some programming tasks require manipulation of data that is not a standard length, so we need to work at the bit level.
- Some of these tasks include
  - low-level device control,
  - error detection and correction algorithms,
  - data compression,
  - encryption algorithms,
  - and other optimizations.
- Instead of writing solutions in assembly (tedious and not portable due to specific computer architecture), we use a programming language that provides good abstraction, but still allows for low-level control.

Note about signed and unsigned integers

- If the size of an unsigned int, `x`, is 4 bytes, then it is stored in memory as a 32-bit binary number of the form

$$x_{base2} = b_{31}b_{30}...b_n...b_1b_0 \ , \quad where\ b_n \in \{0,1\}$$

- And the decimal value of `x` is calculated as follows.

$$x_{base10} = b_{31} * 2^{31} + b_{30} * 2^{30} + ... + b_n * 2^n + ... + b_1 * 2^1 + b_0 * 2^0$$

- In the case where $x = 277$, x may be stored as 00000000000000000000000100010101.
- Similarly, an 8-bit unsigned char, `c` would be stored in memory as

$$c_{base2} = b_7b_6...b_n...b_0 \ , \quad where\ b_n \in \{0,1\}$$

- The same applies for all unsigned integer types (`long`, `short`, etc).

Note about signed and unsigned integers

- In a k-bit signed type, the Most Significant Bit (MSB) (b_{k-1}), is used as a sign bit ($0 =$ positive, $1 =$ negative).
- For example, if $x = 10000010_2$, then
  - `unsigned char x =` $2^7 + 2^1 = 128 + 2$ `= 130`
  - `signed char x =` $-(2^7 - 2^1) = -(128 - 2)$ `= -126`
- Therefore, signed `chars` range = -128 to 127 and `unsigned chars` range = 0 to 255.
- The same logic applies for 32-bit/64-bit `ints`.

Bitwise and Bitshift

- Two sets of operators are useful:
  - bitwise operators
  - bitshift operators
- Bitwise operators allow you to read and manipulate bits in variables of certain types.
- Available in C, C++, Java, C#

Bitwise Operators

- Bitwise operators only work on integer types: `char`, `short`, `int` and `long`
- Two types of bitwise operators
  - Unary bitwise operators
  - Binary bitwise operators

Bitwise Operators

- Only one unary operator: `NOT` (`~`)
  - (1's complement) Flips every bit. 1's become 0's and 0's become 1's.
  - `~x`
- Binary bitwise operators
  - `AND` (`&`)
    - Similar to boolean `&&`, but works on the bit level.
    - `x & y`
  - `OR` (`|`)
    - Similar to boolean `||`, but works on the bit level.
    - `x | y`
  - `XOR` (`^`) (eXclusive-OR)
    - *Only* returns a 1 if one bit is a 1 and the other is 0. Unlike `OR`, `XOR` returns 0 if both bits are 1.
    - `x ^ y`

Examples and exercises

- Example: C-examples/bitwise-operators/simple.c
  - Run with GDB.
  - Print binary values using `print \t x`.
- Exercise: Define `XOR` in terms of `NOT`, `AND` and `OR`.

What can we do with bitwise operators?

- You can represent 32 boolean variables very compactly. You can use an `int` variable (assuming it has 4 bytes) as a 32 bit Boolean array.
- Unlike the `bool` type in C++, which presumably requires one byte, you can make your own Boolean variable using a single bit.
- However, to do so, you need to access individual bits.

What can we do with bitwise operators? (contd.)

- 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.
- Bit manipulation is considered really low-level programming.
- Many higher level languages hide the operations from the programmer
- However, languages like C are often used for systems programming where data representation is quite important.

What can we do with bitwise operators? (contd.)

- Setting, un-setting, or checking whether a specific bit, i, is set.
- Ideas?
- Masks
  - If you need to check whether a specific bit is set, then you need to create an appropriate mask and use bitwise operators. For e.g.,

    ```
    #define MSB_MASK 0x80  // 1000 0000
    unsigned char x = 77;  // 0111 0111
    ```
- How would we set MSB? `x = x | mask`

Exercise

- Exercise: Assume we have a device with the following status fields. Each status will be stored as a bit of an `unsigned char` (8 bits).

ENABLE	READY	ERROR	RUNNING	LOADED	LOCKED	(null)	(null)
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

- Create a bit mask for each of the status bits.
- How would you ENABLE the device and set it to READY?
- How would you indicate an error? (if there is an error, then it won't be ready anymore)
- How would you clear all bits?

Bitshift Operators

- The << and >> operators have different meanings in C and C++
  - In C, they are bitshift operators
  - In C++, they are stream insertion and extraction operators
- The bitshift operators take two arguments
  - `x << n`
  - `x >> n`
- x can be any kind of int variable or char variable and n can be any kind of int variable

Left shift operator

- Left shift operator <<
  - `x << n` shifts the bits for x leftward by n bits, filling the vacated bits by zeroes
- Eg : x = 50 = 0011 0010 followed by `x<<4`
- Think about what shifting left means?
- Left shifting by k bits = multiplying by 2^k
- Each bit contributes 2^i to the overall value of the number

Issues with « Operator

- For unsigned int, when the first "1" falls off the left edge, the operation has overflowed.
- It means that you have multiplied by 2^k such that the result is larger than the largest possible unsigned int.
- Shifting left on signed values also works, but overflow occurs when the most significant bit changes values (from 0 to 1, or 1 to 0).

Right shift operator

- Right shift operator >>
  - `x >> n` shifts the bit rightward by n bits
- Example: x $=$ 1011 0010 and `x>>4`
- What does shifting right mean?
- For unsigned int, and sometimes for signed int, shifting right by k bits is equivalent to dividing by 2^k (using integer division).

Issues with » Operator

- Creates few problems regardless of data type
- Shifting does NOT change values

```
x = 3 ; n = 2 ;
x << n ;
printf("%d", x); // Prints 3 NOT 12
```

- Shifting right using >> for signed numbers is implementation dependent. It may shift in the sign bit from the left, or it may shift in 0's (it makes more sense to keep shifting in the sign bit).

Example

- Add bitshift operators to C-examples/bitwise-operators/simple.c
  - Run with GDB.

Creating a Dynamic Mask

- Recall: If you need to check whether a specific bit is set, then you need to create an appropriate mask and use bitwise operators. For e.g.,

```
unsigned char mask = 128; // 1000 0000
unsigned char x = 77;     // 0111 0111
```

- But how would you create dynamic bit mask?

```
unsigned char mask = 1 << i;
```

- Causes i^{th} bit to be set to 1.
  - If `i = 4`, then the mask would be `00010000`.

Checking if a bit is set

- Create a mask.
- Followed by using a bitwise AND operator

```
unsigned char isBitSet( unsigned char ch, int i )
{
    unsigned char mask = 1U << i;
    return mask & ch;
}
```

- If return value is anything other than 0, then bit `i` was set.

Setting a bit

- Create a mask
- Followed by using a bitwise OR operator

```
unsigned char setBit( unsigned char ch, int i )
{
    unsigned char mask = 1U << i;
    return mask | ch;
}
```

exercise

- Write a function that clears bit i. (i.e., makes bit i's value 0 no matter what).

```
unsigned char clearBit(unsigned char ch, int i)
{

}
```

Is Any Bit Set Within a Range (1)

- Is any bit set within a range?

```
bool isBitSetInRange(unsigned char ch, int low, int high );
```

- This function returns true if any bit within $b_{high} \ldots b_{low}$ has a value of 1.
- Assume that `low <= high`.
- All bits could be 1, or some bits could be 1, or exactly 1 bit in the range could be 1, and they would all return true.
- Return false only when all the bits in that range are 0.

Is Any Bit Set Within a Range (2)

- How do we check for a bit set within a range?
- Method 1:
  - Write a for loop that checks if bit i is set

```
for (int i = low; i <= high; i++)
{
    if (isBitSet(ch, i)
        return true;
}
return false;
```

Is Any Bit Set Within a Range (3)

- Method 2: No loops. Define a mask over the specified range.
  - Combination of bitwise operation and subtraction.
  - Need a mask with 1's between b_{low} and b_{high}.
  - How can you get k 1's?
  - One method:

```
unsigned char mask = 1 << k;  // if k = 3, 00001000
mask = mask - 1;              //           00000111
```

- Think about it...

Is Any Bit Set Within a Range (4)

- How do we use this to get get mask for range from b_{low} to b_{high}?
- Method 1:

```
int k = (high - low) + 1;      // e.g. high = 4, low = 2, k = 3
unsigned char mask = 1 << k;   // 00001000
mask = mask - 1;               // 00000111
mask = mask << low;            // 00011100
```

- Method 2:

```
unsigned char maskHigh = (1 << (high + 1)) - 1; // 00011111
unsigned char maskLow = (1 << low) - 1;         // 00000011
unsigned char mask = maskHigh ^ maskLow;        // 00011100
```

Is Any Bit Set Within a Range (5)

- Function now looks like

```
bool isBitSetInRange(unsigned char ch, int low, int high)
{
    unsigned char maskHigh = (1 << (high + 1)) - 1;
    unsigned char maskLow = (1 << low) - 1;
    unsigned char mask = maskHigh ^ maskLow; // 00011100
    return ch & mask;
}
```

- As long as at least one bit is 1, then the result is non-zero, and thus, the return value is true.

Another example

Write a function `getbits(x, p, n)` that returns the (right adjusted) `n`-bit field of `x` that begins at position `p`. Assume that bit position 0 is at the right end and that `n` and `p` are sensible positive values.

```
unsigned int getbits (unsigned int x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

See example C-examples/bitwise-operators/getbits.c.

References

- The C Programming Language Kernighan and Ritchie
- Computer Organization & Design: The Hardware/Software Interface, David Patterson & John Hennessy, Morgan Kaufmann
- http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/index.html

Exercises

- Read Section 2.9 from the C book.
- Write a function that computes the parity bit for a given unsigned integer. The parity bit is 1 if the number of 1 bits in the integer is even. The parity bit is 0 if the number of 1 bits in the integer is odd. This is known as odd-parity. We can also compute the even-parity, which is the opposite of odd-parity. Parity bits are used for error detection in data transmission.
- Write a function that sets bit from $b_{high}...b_{low}$ to all 1's, while leaving the remaining bits unchanged.
- Write a function that clears bits from $b_{high}...b_{low}$ to all 0's, while leaving the remaining bits unchanged.