Motivation

- High-level languages, for the most part, try to make you as unaware of the hardware as possible
- Not entirely true, because efficiency is still a major consideration for some programming languages.
- C was created to make it easier to write operating systems.

Motivation

- Writing operating systems requires the manipulation of data at addresses, and this requires manipulating individual bits or groups of bits.
- Rather than write an Operating System in assembly(tedious and not portable due to specific computer architecture)
- Goal - language that provided good control-flow, some abstractions (structures, function calls), and could be efficiently compiled and run quickly

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
- If an `unsigned int` x uses 32 bits of memory, then x is actually represented as $x = x_{31}x_{30}x_{29}...x_0$
- An `unsigned char` c will be represented as $c_7c_6\dots c_0$. Similarly for `short`, `long` etc.

Note about signed and unsigned

- The first bit (most significant bit - MSB) in signed types is used as a sign bit
- For e.g., `x = 1000 0010`
  - `unsigned char x = 130`
  - `signed char x = -126`
- Similarly, for 32-bit/64-bit ints the first bit (MSB) denotes the sign ($0 =$ positive, $1 =$ negative)
- Therefore, signed chars range = -128 to 127 and unsigned chars range = 0 to 255

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`
- Demo bitwise example
- *In-class exercise*: define XOr in terms of NOT, AND and OR

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 takes 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 $\gg$ 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).
- Demo bitshift operators

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.)

- Checking whether 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., `x= 1111 0111, m = 0000 1000 => x&m`
- How do you create a mask?

Creating a Mask

- `unsigned char mask = 1 << i;`
- Causes i^{th} bit to be set to 1. Since we are testing if bit i is set
- To use the mask on 8-bit data,

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

Setting a bit

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

```
unsigned char setBit( unsigned char ch, int i )
{
    unsigned char mask = 1 << i;
    return mask | ch; // using bitwise OR
}
```

In-class exercise

- Write a function that clears bit *i* (i.e., makes bit *i*'s value 0).

```
unsigned char clearBit(unsigned char ch, int i)
{
    unsigned char mask = 1 << i ;
    return ~mask & ch ; // using & for clearing
}
// How about using mask ^ ch ?
```

Bit operators

- Is any bit set within a range?
  `bool isBitSetInRange(char ch, int low, int high );`
- This function returns true if any bit within $b_{high}...b_{low}$ had 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

- Create a Mask with 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

- Method 2: No loops
  - 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 int mask = 1 << k;
mask = mask - 1; // mask -=1;
```

- Think about it

Computing the Mask

- Get the low and high bits

```
unsigned char maskHigh = (1 << (high + 1)) - 1;
unsigned char maskLow = (1 << low) - 1;
unsigned char mask = maskHigh - maskLow;
```

- Function now looks like

```
bool isBitSetInRange(char ch, int low, int high)
{
    unsigned char maskHigh = (1 << (high + 1)) - 1;
    unsigned char maskLow = (1 << low) - 1;
    unsigned char mask = maskHigh - maskLow;
    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.
- Read Chapter 3 (Control-flow) 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.