• “College is a place where the professor’s lecture notes go straight to the students’ lecture notes, without passing through the brains of either.” – Mark Twain

“Some people talk in their sleep. Lecturers talk while other people sleep.” – Albert Camus
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 UNIX in assembly (tedious & not portable due to specific ISA)
• 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 a limited number of types: `int` and `char`
• Two types of Bitwise operators
  – Unary bitwise operators
  – Binary bitwise operators
• Most unsigned `int`’s use 32 bits of memory. Then `X` is actually represented as
  – \( X = x_{31}x_{30}x_{29} \ldots x_0 \)
Note about signed and unsigned

• The first bit (most significant bit - MSB) in signed int’s/char’s is used a sign bit
  • For e.g., x = 1000 0010
    – unsigned char x = 82
    – signed char x = -2
• Similarly, for 32-bit/64-bit int’s the first bit (MSB) denotes the sign (0 = +ve, 1 = -ve)
• Therefore, signed char’s range = -128 to 127 & unsigned char’s range = 0 to 255
Bitwise Operators

• Only one unary operator – NOT
• Binary bitwise operators
  – AND (&)
    • x & y
  – OR (|)
    • x | y
  – XOR (^)
    • x ^ y
• Demo bitwise example
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 or
  – x >> n
• x can be any kind of int variable or char variable and n can be any kind of int variable
Bitshift Operators

• Operator $\ll$
  – $x \ll n$ shifts the bits for $x$ leftward by $n$ bits
• Eg: $x = 50 = 0011\ 0010$ followed by $x\ll4$
• 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).
Bitshift Operators

• 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 \texttt{\textgreater\textgreater} Operator

- Creates few problems regardless of data type
- Shifting does \textbf{NOT} change values
  
  \begin{verbatim}
  x = 3 ; n = 2 ;
  x \ll n ;
  printf("%d", x);  // Prints 3 NOT 12
  \end{verbatim}

- Shifting right using \texttt{\textgreater\textgreater} 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 it?

• 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.
For 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.
• 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.
Bit operations

• 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., $x = 1111\ 0111$, $m = 0000\ 1000 \implies x \& m$

• How do you create a mask?
Creating a Mask

• unsigned char mask = 1 << i ;
  – Causes i\textsuperscript{th} bit to be set to 1. Since we are testing if bit i is set

• To use the mask on 8-bit data,

```c
unsigned char isBitISet( 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

```c
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).

```c
unsigned char setBit( unsigned char ch, int i )
{
    unsigned char mask = 1 << i;
    return mask ^ ch; // using bitwise XOR
}
// And NOT works for any bit
```
Homework Exercises

• Write a function that sets bit from $b_{\text{high}}...b_{\text{low}}$ to all 1's, while leaving the remaining bits unchanged.

• Write a function that clears bits from $b_{\text{high}}...b_{\text{low}}$ to all 0's, while leaving the remaining bits unchanged.
Bit operators

• Is Any Bit Set Within a Range?

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

• This function returns true if any bit within \( b_{\text{high}} \ldots b_{\text{low}} \) had a value of 1.

• Assume that \( \text{low} \leq \text{high} \).

• 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 ( isBitIsSet( 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 $b_{\text{low}}$ and $b_{\text{high}}$
  – 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

```
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.
Homework Exercise

• Write a function to find out whether an integer is a power of two.
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