Binary Numbers

COS 217
Goals of Today’s Lecture

• Binary numbers
  o Why binary?
  o Converting base 10 to base 2
  o Octal and hexadecimal

• Integers
  o Unsigned integers
  o Integer addition
  o Signed integers

• C bit operators
  o And, or, not, and xor
  o Shift-left and shift-right
  o Function for counting the number of 1 bits
  o Function for XOR encryption of a message
Why Bits (Binary Digits)?

• **Computers are built using digital circuits**
  - Inputs and outputs can have only two values
  - True (high voltage) or false (low voltage)
  - Represented as 1 and 0

• **Can represent many kinds of information**
  - Boolean (true or false)
  - Numbers (23, 79, …)
  - Characters (‘a’, ‘z’, …)
  - Pixels
  - Sound

• **Can manipulate in many ways**
  - Read and write
  - Logical operations
  - Arithmetic
  - …
Base 10 and Base 2

- **Base 10**
  - Each digit represents a power of 10
  - $4173 = 4 \times 10^3 + 1 \times 10^2 + 7 \times 10^1 + 3 \times 10^0$

- **Base 2**
  - Each bit represents a power of 2
  - $10110 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22$

**Divide repeatedly by 2 and keep remainders**

- $12/2 = 6 \quad R = 0$
- $6/2 = 3 \quad R = 0$
- $3/2 = 1 \quad R = 1$
- $1/2 = 0 \quad R = 1$

Result = **1100**
Writing Bits is Tedious for People

• Octal (base 8)
  o Digits 0, 1, …, 7
  o In C: 00, 01, …, 07

• Hexadecimal (base 16)
  o Digits 0, 1, …, 9, A, B, C, D, E, F
  o In C: 0x0, 0x1, …, 0xf

<table>
<thead>
<tr>
<th>Binary</th>
<th>Octal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Thus the 16-bit binary number 1011 0010 1010 1001 converted to hex is B2A9.
Representing Colors: RGB

• Three primary colors
  o Red
  o Green
  o Blue

• Strength
  o 8-bit number for each color (e.g., two hex digits)
  o So, 24 bits to specify a color

• In HTML, on the course Web page
  o Red: <font color="#FF0000"><i>Symbol Table Assignment Due</i></font>
  o Blue: <font color="#0000FF"><i>Fall Recess</i></font>

• Same thing in digital cameras
  o Each pixel is a mixture of red, green, and blue
Storing Integers on the Computer

• Fixed number of bits in memory
  o Short: usually 16 bits
  o Int: 16 or 32 bits
  o Long: 32 bits

• Unsigned integer
  o No sign bit
  o Always positive or 0
  o All arithmetic is modulo $2^n$

• Example of unsigned int
  • 00000001 $\rightarrow$ 1
  • 00001111 $\rightarrow$ 15
  • 00010000 $\rightarrow$ 16
  • 00100001 $\rightarrow$ 33
  • 11111111 $\rightarrow$ 255
Adding Two Integers: Base 10

- From right to left, we add each pair of digits
- We write the sum, and add the carry to the next column

\[
\begin{array}{c}
\text{+} & 1 & 9 & 8 \\
\text{2} & 6 & 4 \\
\hline
\text{Sum} & 4 & 6 & 2 \\
\text{Carry} & 0 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{c}
\text{+} & 0 & 1 & 1 \\
\text{0} & 0 & 1 \\
\hline
\text{Sum} & 1 & 0 & 0 \\
\text{Carry} & 0 & 1 & 1 \\
\end{array}
\]
## Binary Sums and Carries

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### XOR

\[
\begin{array}{c}
0100 \\
+ 0110 \\
\hline
1010
\end{array}
\]

### AND

\[
\begin{array}{c}
0101 \\
1111 \\
\hline
1100
\end{array}
\]

- XOR: 69
- AND: 172
Modulo Arithmetic

• Consider only numbers in a range
  o E.g., five-digit car odometer: 0, 1, …, 99999
  o E.g., eight-bit numbers 0, 1, …, 255

• Roll-over when you run out of space
  o E.g., car odometer goes from 99999 to 0, 1, …
  o E.g., eight-bit number goes from 255 to 0, 1, …

• Adding $2^n$ doesn’t change the answer
  o For eight-bit number, n=8 and $2^n=256$
  o E.g., $(37 + 256) \mod 256$ is simply 37

• This can help us do subtraction…
  o Suppose you want to compute $a - b$
  o Note that this equals $a + (256 -1 - b) + 1$
One’s and Two’s Complement

• One’s complement: flip every bit
  o E.g., b is 01000101 (i.e., 69 in base 10)
  o One’s complement is 10111010
  o That’s simply 255-69

• Subtracting from 11111111 is easy (no carry needed!)

```
  1111 1111
- 0100 0101
  1011 1010
```

  b

  one’s complement

• Two’s complement
  o Add 1 to the one’s complement
  o E.g., (255 – 69) + 1 ➞ 1011 1011
Putting it All Together

• Computing “a – b” for unsigned integers
  o Same as “a + 256 – b”
  o Same as “a + (255 – b) + 1”
  o Same as “a + onecomplement(b) + 1”
  o Same as “a + twocomplement(b)”

• Example: 172 – 69
  o The original number 69: 0100 0101
  o One’s complement of 69: 1011 1010
  o Two’s complement of 69: 1011 1011
  o Add to the number 172: 1010 1100
  o The sum comes to: 0110 0111
  o Equals: 103 in base 10
Signed Integers

• **Sign-magnitude representation**
  o Use one bit to store the sign
    – Zero for positive number
    – One for negative number
  o Examples
    – E.g., 0010 1100 \(\Rightarrow\) 44
    – E.g., 1010 1100 \(\Rightarrow\) -44
  o Hard to do arithmetic this way, so it is rarely used

• **Complement representation**
  o One’s complement
    – Flip every bit
    – E.g., 1101 0011 \(\Rightarrow\) -44
  o Two’s complement
    – Flip every bit, then add 1
    – E.g., 1101 0100 \(\Rightarrow\) -44
Overflow: Running Out of Room

- Adding two large integers together
  - Sum might be too large to store in the number of bits allowed
  - What happens?

- Unsigned numbers
  - All arithmetic is “modulo” arithmetic
  - Sum would just wrap around

- Signed integers
  - Can get nonsense values
  - Example with 16-bit integers
    - Sum: 10000+20000+30000
    - Result: -5536
  - In this case, fixable by using “long”…
Bitwise Operators: AND and OR

- Bitwise AND (&)

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- Mod on the cheap!
  - E.g., h = 53 & 15;

  53 \[\begin{array}{cccccc}
  0 & 0 & 1 & 1 & 0 & 1 \\
  \end{array}\] & 15 \[\begin{array}{cccccc}
  0 & 0 & 0 & 0 & 1 & 1 \\
  \end{array}\]  

  \[\begin{array}{cccccc}
  0 & 0 & 0 & 0 & 1 & 0 \\
  \end{array}\]

- Bitwise OR (|)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>


Bitwise Operators: Not and XOR

• One’s complement (\(\sim\))
  - Turns 0 to 1, and 1 to 0
  - E.g., set last three bits to 0
    \[-x = x \& \sim 7;\]

• XOR (^)
  - 0 if both bits are the same
  - 1 if the two bits are different

\[
\begin{array}{c|cc}
^ & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]
Bitwise Operators: Shift Left/Right

- **Shift left (<<):** Multiply by powers of 2
  - Shift some # of bits to the left, filling the blanks with 0

  53 \[0 0 1 1 0 1 0 0\]
  
  53 << 2 \[1 1 0 1 0 0 0 0\]

- **Shift right (>>):** Divide by powers of 2
  - Shift some # of bits to the right
    - For unsigned integer, fill in blanks with 0
    - What about signed integers? Varies across machines…
      - Can vary from one machine to another!

  53 \[0 0 1 1 0 1 0 0\]
  
  53 >> 2 \[0 0 0 0 1 1 0 1\]
Count Number of 1s in an Integer

• Function bitcount(unsigned x)
  o Input: unsigned integer
  o Output: number of bits set to 1 in the binary representation of x

• Main idea
  o Isolate the last bit and see if it is equal to 1
  o Shift to the right by one bit, and repeat

```c
int bitcount(unsigned x) {
    int b;
    for (b = 0; x != 0; x >>= 1)
        if (x & 1)
            b++;
    return b;
}
```
XOR Encryption

• Program to encrypt text with a key
  o Input: original text in stdin
  o Output: encrypted text in stdout

• Use the same program to decrypt text with a key
  o Input: encrypted text in stdin
  o Output: original text in stdout

• Basic idea
  o Start with a key, some 8-bit number (e.g., 0110 0111)
  o Do an operation that can be inverted
    – E.g., XOR each character with the 8-bit number

\[
\begin{array}{cc}
0100 & 0101 \\
\wedge & 0110 & 0111 \\
\hline
0110 & 0111 \\
0010 & 0010 \\
\end{array}
\quad
\begin{array}{cc}
0010 & 0010 \\
\wedge & 0110 & 0111 \\
\hline
0100 & 0101 \\
0100 & 0101 \\
\end{array}
\]
XOR Encryption, Continued

• But, we have a problem
  o Some characters are control characters
  o These characters don’t print

• So, let’s play it safe
  o If the encrypted character would be a control character
  o … just print the original, unencrypted character
  o Note: the same thing will happen when decrypting, so we’re okay

• C function `iscntrl()`
  o Returns true if the character is a control character
#define KEY ‘&’
int main() {
    int orig_char, new_char;

    while ((orig_char = getchar()) != EOF) {
        new_char = orig_char ^ KEY;
        if (iscntrl(new_char))
            putchar(orig_char);
        else
            putchar(new_char);
    }
    return 0;
}
Next Week

• Wednesday lecture time
  o Midterm exam
  o Open book and open notes
  o Practice exams online
Stupid Programmer Tricks

• Where do I use bitwise & most?
  o Bit vectors

• What’s a bit vector?
  o Lots of booleans packed into an int/long
  o Often used to indicate some condition(s)
  o Less storage space than lots of fields
  o More explicit storage than compiled-defined bit fields

• Your compiler can do this?
  typedef struct blah {
    int b_onoff:1;
    int b_temperature:7;
    char b_someChar;
  }

### Example From Real Code

- `#define DONTCACHE_REQNOSTORE` 0x000001
- `#define DONTCACHE_AUTHORIZED` 0x000002
- `#define DONTCACHE_MISSINGVARIANTHDR` 0x000004
- `#define DONTCACHE_USERORPASS` 0x000008
- `#define DONTCACHE_BYPASSFILTER` 0x000010
- `#define DONTCACHE_NONCACHEMETHOD` 0x000020
- `#define DONTCACHE_CTLPRIVATE` 0x000040
- `#define DONTCACHE_CTLNOSTORE` 0x000080
- `#define DONTCACHE_ISQUERY` 0x000100
- `#define DONTCACHE_EARLYEXPIRE` 0x000200
- `#define DONTCACHE_NOLASTMOD` 0x000400
- `#define DONTCACHE_NONEGCACHING` 0x000800
- `#define DONTCACHE_INSTANTEXPIRE` 0x001000
- `#define DONTCACHE_FILETOOBIG` 0x002000
- `#define DONTCACHE_FILEGREWTOOBIG` 0x004000
- `#define DONTCACHE_ICPPROXYONLY` 0x008000
- `#define DONTCACHE_LARGEFILEBLAST` 0x010000
- `#define DONTCACHE_PERSISTLOGLOADING` 0x020000
- `#define DONTCACHE_NEWERCOPYEXISTS` 0x040000
- `#define DONTCACHE_BADVARYFIELDS` 0x080000
- `#define DONTCACHE_SETCOOKIE` 0x100000
- `#define DONTCACHE_HTTPSTATUSCODE` 0x200000
- `#define DONTCACHE_OBJECTINCOMPLETE` 0x400000
Conclusions

• Computer represents everything in binary
  o Integers, floating-point numbers, characters, addresses, …
  o Pixels, sounds, colors, etc.

• Binary arithmetic through logic operations
  o Sum (XOR) and Carry (AND)
  o Two’s complement for subtraction

• Binary operations in C
  o AND, OR, NOT, and XOR
  o Shift left and shift right
  o Useful for efficient and concise code, though sometimes cryptic