Goals of Today’s Lecture

• Representations
  ◦ Why binary?
  ◦ Converting base 10 to base 2
  ◦ Octal and hexadecimal

• Integers
  ◦ Unsigned integers
  ◦ Integer addition, subtraction
  ◦ Signed integers

• C bit operators
  ◦ And, or, not, and xor
  ◦ Shift-left and shift-right
  ◦ Function for counting the number of 1 bits
  ◦ Function for XOR encryption of a message
Radiohead - OK Computer CD

3 Miles of Music

Pits and Lands

Transition represents a bit state (1/on/red/female/heads)
No change represents other state (0/off/white/male/tails)

Interpretation

As Music:

01110101₂ = 117/256 position of speaker

As Number:

01110101₂ = 1 + 4 + 16 + 32 + 64 = 117₁₀ = 75₁₆
(Get comfortable with base 2, 8, 10, and 16.)

As Text:

01110101₂ = 117th character in the ASCII codes = “u”
Interpretation – ASCII

Computer Science Building West Wall

Interpretation: Code and Data (Hello World!)

- Programs consist of Code and Data
- Code and Data are Encoded in Bits

IA-64 Binary (objdump)
Interpretation:
Numbers

• 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

\[
\begin{align*}
12/2 &= 6 \quad R = 0 \\
6/2 &= 3 \quad R = 0 \\
3/2 &= 1 \quad R = 1 \\
1/2 &= 0 \quad R = 1 \\
\end{align*}
\]

Result = 1100

Writing Bits is Tedious for People

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

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

\[
\begin{align*}
0000 &= 0 \\
0001 &= 1 \\
0010 &= 2 \\
0011 &= 3 \\
0100 &= 4 \\
0101 &= 5 \\
0110 &= 6 \\
0111 &= 7 \\
1000 &= 8 \\
1001 &= 9 \\
1010 &= A \\
1011 &= B \\
1100 &= C \\
1101 &= D \\
1110 &= E \\
1111 &= F \\
\end{align*}
\]

Thus the 16-bit binary number

\(1011\ 0010\ 1010\ 1001\)

converted to hex is

B2A9

Interpretation:
Colors

• Three primary colors
  - Red
  - Green
  - Blue

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

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

• Same thing in digital cameras
  - Each pixel is a mixture of red, green, and blue
Binary Representation of Integers

- **Fixed number of bits in memory**
  - char: 8 bits
  - short: usually 16 bits
  - int: 16 or 32 bits
  - long: 32 bits
  - long long: 64 bits

- **Unsigned integers**
  - Always positive or 0
  - All arithmetic is modulo $2^n$
    - unsigned char
    - unsigned short
    - unsigned int
    - unsigned long
    - unsigned long long

### Table: Binary vs Decimal

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$1(n)$</td>
<td>$2^n-1$</td>
</tr>
</tbody>
</table>

Size and Overflow in Unsigned Integers

<table>
<thead>
<tr>
<th>Bits</th>
<th>Integer Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0 - 255</td>
</tr>
<tr>
<td>16</td>
<td>0 - 65,535</td>
</tr>
<tr>
<td>32</td>
<td>0 - 4,294,967,295</td>
</tr>
<tr>
<td>64</td>
<td>0 - 18,446,744,073,709,551,615</td>
</tr>
</tbody>
</table>

Number of bits determines unsigned integer range

**Overflow:**
- 8-bit integer $\rightarrow$ 11111111₂ (255₁₀)
- Add 1
- What happens?

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Carry</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Carry</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
### Binary Sums and Carries

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

**XOR**

| 0100 | 0101 | ← 69 |
| +    | 0110 | ← 103 |
| 1010 | 1100 | ← 172 |

### Overflow in Unsigned Addition

Operands: \( w \) bits

True Sum: \( w + 1 \) bits

Discard Carry: \( w \) bits

\[
\begin{align*}
\text{UAdd}_w(u,v) &= \begin{cases} 
    u + v & u + v < 2^w \\
    u + v - 2^w & u + v \geq 2^w 
\end{cases}
\end{align*}
\]

Modulo Arithmetic: \( \text{UAdd}_w(u, v) = u + v \mod 2^w \)

### Detecting Unsigned Overflow

- **Task:**
  - Given \( s = \text{UAdd}_w(u, v) \)
  - Determine if \( s = u + v \)

- **Claim:**
  - Overflow iff \( s < u \)
  - \( \text{ovf} = (s < u) \)
  - By symmetry iff \( s < v \)

- **Proof:**
  - \( 0 \leq v < 2^w \)
  - No overflow \( \Rightarrow s = u + v \geq u + 0 = u \)
  - Overflow \( \Rightarrow s = u + v - 2^w < u + 0 = u \)
Modulo Arithmetic

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

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

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

- This can help us do subtraction…
  - Suppose you want to compute $a - b$
  - Note that this equals $a + (256 - 1 - b) + 1$

Modulo Addition Forms an Abelian Group

- Closed under addition
  - $0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1$

- Commutative
  - $\text{UAdd}_w(u, v) = \text{UAdd}_w(v, u)$

- Associative
  - $\text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v)$

- 0 is additive identity
  - $\text{UAdd}_w(u, 0) = u$

- Every element has additive inverse
  - Let $U\text{Comp}_w(u) = 2^w - u$
  - $\text{UAdd}_w(u, U\text{Comp}_w(u)) = 0$
What about Negative Numbers?

- We have been looking at unsigned numbers
- What about negative or signed numbers?
- Need new interpretation of bits
- Some patterns interpreted as negative numbers

<table>
<thead>
<tr>
<th>Bits</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>65,536</td>
</tr>
<tr>
<td>32</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>64</td>
<td>18,446,744,073,709,551,616</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1{n}</td>
<td>2^n</td>
</tr>
</tbody>
</table>

Key Standard Pattern Assignments

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Sign Magnitude</th>
<th>One’s Complement</th>
<th>Two’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>+0</td>
<td>+0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>010</td>
<td>+2</td>
<td>+2</td>
<td>+2</td>
</tr>
<tr>
<td>011</td>
<td>+3</td>
<td>+3</td>
<td>+3</td>
</tr>
<tr>
<td>100</td>
<td>-0</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>101</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
</tr>
<tr>
<td>110</td>
<td>-2</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>111</td>
<td>-3</td>
<td>-0</td>
<td>-1</td>
</tr>
</tbody>
</table>

- Which one is best?
  - Balance
  - Zeros
  - Ease of operations

Most Common: Two’s Complement

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Two’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>+1</td>
</tr>
<tr>
<td>010</td>
<td>+2</td>
</tr>
<tr>
<td>011</td>
<td>+3</td>
</tr>
<tr>
<td>100</td>
<td>-4</td>
</tr>
<tr>
<td>101</td>
<td>-3</td>
</tr>
<tr>
<td>110</td>
<td>-2</td>
</tr>
<tr>
<td>111</td>
<td>-1</td>
</tr>
</tbody>
</table>

- “Invert and Add 1” to negate
- Sign Bit
- Zeros, Range
- What about arithmetic?
Unsigned and Two’s Complement

- **Unsigned Values**
  - \( B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \)
  - \( UMin = 0 \)
  - \( UMax = 2^w - 1 \)

- **Two’s Complement**
  - \( B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \)
  - \( TMin = -2^{w-1} \)
  - \( TMax = 2^{w-1} - 1 \)

Values for \( W = 16 \)

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

Representation Relationship

Sizes and C Data Types

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>MIPS, x86</th>
<th>Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8 bits</td>
<td>8 bits</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>int</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>long int</td>
<td>32 bits</td>
<td>64 bits</td>
</tr>
</tbody>
</table>

- char, short, int, long int
  - Refer to number of bits of integer
  - Most machines: signed two’s complement
  - unsigned <type>
    - Same number of bits as signed counterparts
    - Unsigned integer
Sign Extension

```java
char minusFour = -4;
short moreBits;
moreBits = (short) minusFour;
```

Given a $w$ bit signed integer, return the equivalent $w+k$ bit signed integer.

**Sign Extend:**

\[
\begin{array}{c}
\quad X \\
\quad \vdots \\
\quad X' \\
\end{array}
\]

\[
\begin{array}{c}
\quad k \\
\quad w \\
\end{array}
\]

Sign Extension

**Proof of Correctness Outline**

- Prove Correctness by Induction on $k$
- Induction Step: extending by single bit maintains value

Two’s Complement Addition

- TAdd and UAdd have identical Bit-Level Behavior!

<table>
<thead>
<tr>
<th>Operands: $w$ bits</th>
<th>$u$</th>
<th>+</th>
<th>$v$</th>
<th>$u + v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Sum: $w+1$ bits</td>
<td>$u + v$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Discard Carry: $w$ bits</td>
<td>TAdd, $(u, v)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Characterizing TAdd

- True sum requires $w+1$ bits
- Drop MSB

\[
TAdd(u, v) = \begin{cases} 
    u + v + 2^{w-1} & \text{if } u + v < TMin_w \\
    u + v & \text{if } TMin_w \leq u + v \leq TMax_w \\
    u + v - 2^{w-1} & \text{if } TMax_w < u + v
\end{cases}
\]

Detecting Two’s Complement Overflow

- Task:
  - Given $s = TAdd_w(u, v)$
  - Determine if $s = Add_w(u, v)$
- Claim:
  - Overflow iff either:
    - $u, v < 0$, $s \geq 0$ (NegOver)
    - $u, v \geq 0$, $s < 0$ (PosOver)
  - $ovf = (u < 0 == v < 0) \&\& (u < 0 \neq s < 0)$;
- Proof:
  - Obviously, if $u \geq 0$ and $v < 0$, then $TMin_w \leq u + v \leq TMax_w$
  - Symmetrically if $u < 0$ and $v \geq 0$
  - Other cases from analysis of TAdd

Negation vs. Inversion

Inversion:
- A bit-wise operation
- Flip all 0’s to 1’s and vice versa: 0011 $\Rightarrow$ 1100
- What does this do to the two’s complement value?

Negation:
- Two’s complement: invert all bits and add 1
- Example:
  - $3_{10} = 0011$
  - $\text{invert}(0011) + 1 \rightarrow 1100 + 1 \rightarrow 1101$
  - $1101 = -3_{10}$
Two’s Complement Negation

- Mostly like Integer Negation
  - $T_{\text{Comp}}(u) = -u$

- $T_{\text{Min}}$ is Special Case
  - $T_{\text{Comp}}(T_{\text{Min}}) = T_{\text{Min}}$
  - Note Also: $T_{\text{Comp}}(0) = 0$

- Negation in C ($x = -x;$) is Actually $T_{\text{Comp}}$

Comparing Two’s Complements

- Given signed numbers $u$, $v$
- Determine whether or not $u > v$
- Return true for shaded region:

- Bad Approach:
  - Test $(u - v) > 0$
  - Problem: Thrown off by Overflow

Representation: A Collection of Bits

- Treat unsigned int as a collection 32 independent bits
- Good for tracking 32 individual binary conditions
  - True/False
  - Yes/No
  - Black/White

- Can also treat unsigned in as:
  - 16 2-bit values
  - 8 4-bit values
  - 4 8-bit values
  - 8 1-bit value, 4 2-bit values, 2 4-bit values, and 1 8-bit value
Bitwise Operators: AND and OR

- Bitwise AND (&)
  
<table>
<thead>
<tr>
<th>&amp;</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 1</td>
</tr>
</tbody>
</table>

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

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

- Bitwise OR (|)
  
<table>
<thead>
<tr>
<th></th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0</td>
</tr>
<tr>
<td>1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

  - 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 1 0

Bitwise Operators: Not and XOR

- One's complement (~)
  - Turns 0 to 1, and 1 to 0
  - E.g., set last three bits to 0
    - x = x & ~7;

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

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

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)`
  - Input: unsigned integer
  - Output: number of bits set to 1 in the binary representation of x

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

```c
int bitcount(unsigned int 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
  - Input: original text in stdin
  - Output: encrypted text in stdout

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

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

```
0100 0101
^ 0110 0111
---
0010 0010
```

XOR Encryption, Continued

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

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

- C function `iscntrl()`
  - Returns true if the character is a control character
XOR Encryption, C Code

```c
#define KEY '&'
int main(void) {
    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;
}
```

Stupid Programmer Tricks

- **Where do I use bitwise & most?**
  - Bit vectors

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

- **Your compiler can do this?**
  ```c
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
```
Conclusions

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

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

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