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
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

<table>
<thead>
<tr>
<th>ASCII value</th>
<th>Character</th>
<th>Control character</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>(null)</td>
<td>NULL</td>
</tr>
<tr>
<td>001</td>
<td>@</td>
<td>SOH</td>
</tr>
<tr>
<td>002</td>
<td>#</td>
<td>STX</td>
</tr>
<tr>
<td>003</td>
<td>$</td>
<td>ETX</td>
</tr>
<tr>
<td>004</td>
<td>%</td>
<td>TID</td>
</tr>
<tr>
<td>005</td>
<td>^</td>
<td>SNAP</td>
</tr>
<tr>
<td>006</td>
<td>&amp;</td>
<td>ACK</td>
</tr>
<tr>
<td>007</td>
<td>(back)</td>
<td>BEL</td>
</tr>
<tr>
<td>008</td>
<td>(bell)</td>
<td>BS</td>
</tr>
<tr>
<td>009</td>
<td>(till)</td>
<td>HT</td>
</tr>
<tr>
<td>010</td>
<td>(line feed)</td>
<td>LF</td>
</tr>
<tr>
<td>011</td>
<td>(home)</td>
<td>VT</td>
</tr>
<tr>
<td>012</td>
<td>(form feed)</td>
<td>FF</td>
</tr>
<tr>
<td>013</td>
<td>(carriage return)</td>
<td>CR</td>
</tr>
<tr>
<td>014</td>
<td>$</td>
<td>SO</td>
</tr>
<tr>
<td>015</td>
<td>(left)</td>
<td>SI</td>
</tr>
<tr>
<td>016</td>
<td>0</td>
<td>DLE</td>
</tr>
<tr>
<td>017</td>
<td>1</td>
<td>DC1</td>
</tr>
<tr>
<td>018</td>
<td>2</td>
<td>DC2</td>
</tr>
<tr>
<td>019</td>
<td>3</td>
<td>DC3</td>
</tr>
<tr>
<td>020</td>
<td>4</td>
<td>DC4</td>
</tr>
<tr>
<td>021</td>
<td>5</td>
<td>NAK</td>
</tr>
<tr>
<td>022</td>
<td>6</td>
<td>SYN</td>
</tr>
<tr>
<td>023</td>
<td>7</td>
<td>ETB</td>
</tr>
<tr>
<td>024</td>
<td>8</td>
<td>CAN</td>
</tr>
<tr>
<td>025</td>
<td>9</td>
<td>LMI</td>
</tr>
<tr>
<td>026</td>
<td>(corser right)</td>
<td>ESC</td>
</tr>
<tr>
<td>027</td>
<td>(corser left)</td>
<td>GS</td>
</tr>
<tr>
<td>028</td>
<td>(corser up)</td>
<td>RS</td>
</tr>
<tr>
<td>029</td>
<td>(corser down)</td>
<td>RS</td>
</tr>
</tbody>
</table>

Computer Science Building West Wall
**Interpretation:**

**Code and Data (Hello World!)**

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

```
00000000: 7475 4e44 0200 0200 0000 0000 0000 0000 ... .ELF...............
...
0000260: 5002 0000 0000 0000 0000 0000 0000 0000 50 68 65 73 74 69 66 69 0a
0000270: 2e31 6e79 6573 7665 616e 646c 744a 6f6e
0000280: 696e 6765 6e63 6961 6e5f 6973 2061 7374
0000290: 6976 6572 6571 7565 7374 2068 6f76 6572
```  

**IA-64 Binary (objdump)**

```
00000860: 4865 6c6c 6f20 7374 7269 6e67 656e 7465
00000861: 2064 6172 7479 7374 696f 6e69 7669 6572
```

**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, …, 0xff

```
0000 = 0 1000 = 8
0001 = 1 1001 = 9
0010 = 2 1010 = A
0011 = 3 1011 = B
0100 = 4 1100 = C
0101 = 5 1101 = D
0110 = 6 1110 = E
0111 = 7 1111 = F
```

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

**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:

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

Result = 1100

**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"><i>Symbol Table Assignment Due</i></font>`
  - Blue: `<font color="#0000FF"><i>Fall Recess</i></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>
<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

Number of bits determines unsigned integer range

Overflow:
- 8-bit integer \(\rightarrow 11111111_2 (255_{10}) \)
- Add 1
- What happens?

<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>

<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>...</td>
<td>...</td>
</tr>
<tr>
<td>1 (n)</td>
<td>(2^{n-1})</td>
</tr>
</tbody>
</table>

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

```
+ 1 9 8
+ 2 6 4
Sum 4 6 2
Carry 0 1 1
```

```
+ 0 1 1
+ 0 0 1
Sum 1 0 0
Carry 0 1 1
```

```
XOR
```

```
AND
```

```
0100 0101 \(\rightarrow\) 69
+ 0110 0111 \(\rightarrow\) 103
1010 1100 \(\rightarrow\) 172
```
Overflow in Unsigned Addition

Operands: \( w \) bits

\[
\begin{array}{c}
\text{True Sum: } w + 1 \text{ bits} \\
\text{Discard Carry: } w \text{ bits}
\end{array}
\]

\[
\begin{array}{c}
\text{UAdd}_w(u, v) = u + v \mod 2^w
\end{array}
\]

\[UAdd_w(u, v) = \begin{cases} 
  u + v & u + v < 2^w \\
  u + v - 2^w & u + v \geq 2^w
\end{cases}\]

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 \( \implies s = u + v \quad \geq u + 0 = u \)
  - Overflow \( \implies s = u + v - 2^w < u + 0 = u \)

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 \( \text{UComp}_w(u) = 2^w - u \)
  - \( \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \)
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

- 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>-0</td>
</tr>
<tr>
<td>101</td>
<td>-1</td>
</tr>
<tr>
<td>110</td>
<td>-2</td>
</tr>
<tr>
<td>111</td>
<td>-3</td>
</tr>
</tbody>
</table>

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

• **Unsigned Values**

\[
B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i
\]

• **Two’s Complement**

\[
B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i
\]

• **UMin = 0**

• **UMax = 2^w – 1**

• **TMin = -2^{w-1}**

• **TMax = 2^{w-1} – 1**

Values for \( W = 16 \)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF 1111 1111 1111</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>7F FF 0111 1111 1111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80 00 1000 0000 0000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF 1111 1111 1111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 0000 0000 0000</td>
</tr>
</tbody>
</table>

Sign Extension

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

Given w bit signed integer, return equivalent w+k bit signed integer

Sign Extend:

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**

**Proof of Correctness Outline**

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

**Characterizing TAdd**

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

**Detecting Two’s Complement Overflow**

- Task:
  - Given $s = \text{TAdd}_w(u, v)$
  - Determine if $s = \text{Add}_w(u, v)$
- Claim:
  - Overflow iff either:
    - $u, v < 0$, $s \geq 0$ (NegOver)
    - $u, v \geq 0$, $s < 0$ (PosOver)
  - $\text{ovf} = (u < 0 == v < 0) && (u < 0 != s < 0)$;
- 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

**Two’s Complement Addition**

- TAdd and UAdd have identical Bit-Level Behavior!

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

Inversion:
• A bit-wise operation
• Flip all 0’s to 1’s and vice versa: 0011 => 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 (&)**
  - Mod on the cheap!
  - E.g., \( h = 53 \& 15; \)

- **Bitwise OR (|)**

\[
\begin{array}{c|c|c|c|c|c|c}
& 0 & 1 \\
--&--&--&--&--
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
& 0 & 1 \\
--&--&--
0 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

- Bitwise Operators: Not and XOR

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

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

\[
\begin{array}{c|c|c|c|c|c|c}
& 0 & 1 \\
--&--&--&--&--
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

Bitwise Operators: Shift Left/Right

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

- **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!

\[
\begin{array}{c|c|c|c|c|c|c}
\text{53} & 0 & 0 & 1 & 1 & 0 & 1 \\
\hline
\text{53<<2} & 1 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
\text{53} & 0 & 0 & 1 & 1 & 0 & 1 \\
\hline
\text{53>>2} & 0 & 0 & 0 & 1 & 1 & 0 \\
\end{array}
\]

Count Number of 1s in an Integer

- Function \texttt{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

\[
\begin{array}{c|c|c|c|c|c|c}
& 0 & 1 \\
--&--&--&--&--
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{c}
\text{int bitcount(unsigned int x) } \\
\{ \\
\text{ int b; } \\
\text{ for (b = 0; x != 0; x >>>= 1) } \\
\text{ if (x & 1) } \\
\text{ b++; } \\
\text{ return b; } \\
\}
\]
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

\[
\begin{array}{c|c}
0100 0101 & 0010 0010 \\
\hline
0110 0111 & 0110 0111 \\
\hline
0010 0010 & 0100 0101 \\
\end{array}
\]

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?
  - 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_NEWERCOPYEXISTS 0x020000
- #define DONTCACHE_BADVARYFIELDS 0x040000
- #define DONTCACHE_HTTPSTATUSCODE 0x200000
- #define DONTCACHE_OBJECTINCOMPLETE 0x400000

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