Number Systems
Number systems

- As humans, we prefer base 10 a.k.a. decimal.
- For reasons we will discuss, computers prefer different number systems…
  - Binary (base 2)
  - Hexadecimal (base 16)
- It's easy to understand these other number systems if we analyze how base 10 works.
Base 10

- For every base, the value of each digit depends on its position in the number.
- Decimal uses 10 digits, 0-9

7423
- is -
(7 * 10^3) + (4 * 10^2) + (2 * 10^1) + (3 * 10^0)
- or -
7000 + 400 + 20 + 3
Base 10

- For every base, the value of each digit depends on its position in the number.

- Decimal uses 10 digits, 0-9

\[
7423 \\
\text{is-} \\
(7 \times 10^3) + (4 \times 10^2) + (2 \times 10^1) + (3 \times 10^0) \\
\text{or-} \\
7000 + 400 + 20 + 3
\]

- Assuming ‘w’ represents the ‘width’ of the number and ‘x’ represents the number itself, we can generalize and convert any base to decimal as follows:

\[
\sum_{i=0}^{w-1} x_i \cdot base^i
\]
Base 2

- Binary uses 2 digits, 0-1

\[ 1101 \]

-is-

\[ (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \]

-or-

\[ 8 + 4 + 0 + 1 = 13_{10} \]
Base 2

- Binary uses 2 digits, 0-1

  1101
  -is-
  \[(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)\]
  -or-
  \[8 + 4 + 0 + 1 = 13_{10}\]

- You should be able to do this by hand! I have provided a way to check.

- See `decimal_to_binary.c`
Base 16

- Hexadecimal uses 16 digits 0-9, A-F \((A=10, B=11, \text{etc.})\)

742A

-is-

\((7 \times 16^3) + (4 \times 16^2) + (2 \times 16^1) + (10 \times 16^0)\)

-or-

\(28672 + 1024 + 32 + 10 = 29738\)

- You can see that the higher the base, the fewer digits it takes to express some value.

- Hexadecimal is used often to note memory addresses (among other things)
Relative expressiveness of systems

- Hexadecimal is useful because it's often more convenient to write one digit as opposed to than four.
  - In other words, since a single digit in hexadecimal can represent 16 values, it can hold as much information as 4 bits.
  - This chart here you should attempt to commit to memory, or at least be able to work out relatively quickly.
Representing Information as Bits
Everything is bits

- Each bit is 0 or 1
- Everything on a computer is encoded as sets of binary digits, or bits
  - All programs running on disk and running in memory are represented as sets of bits
  - … and represent and manipulate numbers, sets, strings, etc…
- Why bits? Electronic implementation
  - Easy to store on bistable elements (an electronic circuit that has two stable states)
  - Reliably transmitted on noisy and inaccurate wires.
Everything is bits *con’t*

- Again, the basic unit of information in computing is the bit.
- A single bit denotes two states “on or off”
- Note that values with more than two states require multiple bits.
  - A collection of two bits has four possible states
    - Ex. 00, 01, 10, 11
  - A collection of three bits has eight possible states
    - Ex. 000, 001, 010, 011, 100, 101, 110, 111
  - A collection of \( w \) bits has \( 2^w \) possible states.
- We call a collection of bits a ‘**bit vector**’
Bytes

- Byte = 8 bits
  - So how many different values can it represent?

- It is the smallest addressable unit of memory in most computer architectures.

- Range of representation (non-negative integers)
  - Binary: 000000002 to 111111112
  - Decimal: 010 to 25510
  - Hexadecimal: 0016 to FF16

  - By the way, we can write hexadecimal numbers in C as
    - 0xFF or 0xff

  - See hex.c
## Data types in C in bytes

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Typical 64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
In some contexts, what is meant by a KB, MB, or GB differ.

- Depends on number base, 2 or 10

The true names for these things are mebibyte, kebibyte, gibibyte, etc..

- You can read in detail here https://en.wikipedia.org/wiki/Kibibyte

In computers we love base 2 so we will be using the binary semantics of the ‘mega’, ‘kilo’, etc. prefixes.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 KB (1KiB)</td>
<td>10^3 bytes = 1,000 bytes</td>
</tr>
<tr>
<td>1 MB (1MiB)</td>
<td>10^6 bytes = 1,000,000 bytes</td>
</tr>
<tr>
<td>1 GB (1GiB)</td>
<td>10^9 bytes = 1,000,000,000 bytes</td>
</tr>
<tr>
<td>1 TB (1TiB)</td>
<td>10^12 bytes = 1,000,000,000,000 bytes</td>
</tr>
</tbody>
</table>
Bit-level manipulations
Boolean Algebra

- Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

**And**
A&B = 1 when both A=1 and B=1

<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>

**Or**
A|B = 1 when either A=1 or B=1

<table>
<thead>
<tr>
<th>1</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>

**Not**
~A = 1 when A = 0

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

**Exclusive-Or (Xor)**
A^B = 1 when either A = 1 or B = 1, but not both

<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>0</td>
</tr>
</tbody>
</table>
Boolean Algebra *con’t*

- We can use this algebra to operate on bit vectors
  - Operations applied bitwise
    
    $$
    \begin{array}{cccc}
    01101001 & 01101001 & 01101001 \\
    \& 01010101 & | 01010101 & ^ 01010101 & \sim 01010101 \\
    \hline
    01000001 & 01111101 & 00111100 & 10101010
    \end{array}
    $$

- All the properties of Boolean Algebra apply.

- We have these operators in C, they are called *bitwise* operators.
Bit-level operations in C

- Operations &, |, ~, ^ available in C
  - Apply to any “integral” data type (long, int, short, char, unsigned)
  - View arguments as bit vectors, arguments applied bit-wise

- Examples:
  - \( \sim 1100 = 0011 \)
  - \( 0110 \& 1010 = 0010 \)
  - \( 0110 \mid 1010 = 1110 \)
  - \( 0110 \^ 1010 = 1100 \)

- See `bit_flipping.c`
Example: representing & manipulating sets

- “Bit sets”, very useful in practice
  - Width w bit vector represents subsets of \( \{0, \ldots, w-1\} \)
  - \( a_j = 1 \) if \( j \in A \)

\[
\begin{align*}
01101001 & \quad \text{represents set} \quad \{0, 3, 5, 6\} \quad \text{Set A} \\
76543210 & \\
01010101 & \quad \text{represents set} \quad \{0, 2, 4, 6\} \quad \text{Set B} \\
76543210 & \\
\end{align*}
\]

- Operations
  - \& \quad \text{Intersection } (A \& B) \quad 01000001 \quad \{0, 6\}
  - | \quad \text{Union } (A | B) \quad 01111101 \quad \{0, 2, 3, 4, 5, 6\}
  - ^ \quad \text{Symmetric difference } (A ^ B) \quad 00111100 \quad \{2, 3, 4, 5\}
  - ~ \quad \text{Complement } (~B) \quad 10101010 \quad \{1, 3, 5, 7\}
Contrast: logical operators

- These operators in some cases look the same but have very different effects.
- `&&`, `||`, `!
  - View 0 as “False”
  - Anything nonzero as “True”
  - All expressions with these operators always return 0 or 1
  - Early termination a.k.a. “short circuiting”

- Example mistake:
  - `1010 & 0101 ➔ 0000` (false)
  - `1010 && 0101 ➔ 0001` (true)

- See `bitwise_vs_boolean.c`
Shift operations

- **Left Shift:** \( x << y \)
  - Shift bit-vector \( x \) left \( y \) positions
  - Throw away extra bits on left
  - Fill with 0’s on right

- **Right Shift:** \( x >> y \)
  - Shift bit-vector \( x \) right \( y \) positions
  - Throw away extra bits on right
  - Logical shift
    - Fill with 0’s on left
  - Arithmetic shift
    - Replicate most significant bit on left

- **Undefined Behavior**
  - Shift amount < 0 or \( \geq \) width of type

- See `bit_shifting.c`

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; \ 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td><strong>Log. ( &gt;&gt; \ 2 )</strong></td>
<td>00011000</td>
</tr>
<tr>
<td><strong>Arith. ( &gt;&gt; \ 2 )</strong></td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; \ 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td><strong>Log. ( &gt;&gt; \ 2 )</strong></td>
<td>00101000</td>
</tr>
<tr>
<td><strong>Arith. ( &gt;&gt; \ 2 )</strong></td>
<td>11101000</td>
</tr>
</tbody>
</table>
Swapping values using XOR

- Swapping values of two variables normally requires a temporary storage

- Using the bitwise exclusive or operator we can actually do this using only the storage of the two bit-vectors

- See xor_swap.c
Integer Encoding
Two Types of Integers

- **Unsigned**
  - positive numbers and 0
  - *unsigned char* has a range of 0-255

- **Signed**
  - negative numbers as well as positive numbers and 0
  - *signed char* has a range of -128-127

- Signed and unsigned have the same cardinality, but different ranges of values!

- If **unsigned** keyword used type is unsigned, if not defaults to signed.

```c
int signed_int = -1;    /* positive & negative allowed */
unsigned int unsigned_int = 1;  /* only non-negative allowed */
```
Unsigned Integers

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

- B2U stands for binary-to-unsigned
- X is a binary number, a bit pattern
- w is the ‘width’ of the binary number (i.e. number of bits)
- Take the sum of every i’th position of X multiplied by \(2^i\)
Unsigned Integers con’t

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

\[ \begin{array}{ccccccc}
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
\downarrow & & & & & & & \\
128 & 0 & 32 & 16 & 8 & 0 & 2 & 1 \\
\end{array} \]

= 187_{10}
Signed integers

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

- **B2T** stands for binary-to-twos-complement
- Same as equation for binary-to-unsigned, with one modification.
- For 2’s complement *most significant bit* indicates *sign*, gets special treatment
  - 0 indicates a nonnegative number
  - 1 indicates a negative number
Signed integers \textit{con’t}

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

\[ \text{sign bit} \]

\[ \begin{array}{cccccccc}
1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
-128 & 0 & 32 & 16 & 8 & 0 & 2 & 1 & \end{array} = -69_{10} \]
Signed integers \textit{con't}

- Again,….

- With \( n \) bits, we have \( 2^n \) distinct values.
  - Assign about half to positive integers and about half to negative non-negative integers
  - If 0 in most significant bit, behave like unsigned:
    \[ 0101 = 5 \]

- Negative integers
  - If 1 in most significant bit, use two’s complement form:
    \[ 1101 = -3 \]
Numeric ranges

- **Unsigned**
  - Umin = 0
  - Umax = $2^{w-1}$

- Example
  - Assume $w = 5$
    - Smallest unsigned
      - $00000_2 = 0_{10}$
    - Largest unsigned
      - $11111_2 = 31_{10}$

- **Signed (Two’s Complement)**
  - Tmin = $-2^{w-1}$
  - Tmax = $2^{w-1} - 1$

- Example
  - Assume $w = 5$
    - Smallest signed
      - $10000_2 = -16_{10}$
    - Largest signed
      - $01111_2 = 15_{10}$
Umax, Tmin, Tmax for standard widths

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>UMax</td>
<td>255</td>
<td>65,535</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
<td>32,767</td>
<td>2,147,483,647</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
<td>-32,768</td>
<td>-2,147,483,648</td>
<td>-9,223,372,036,854,775,808</td>
</tr>
</tbody>
</table>

- Observations:
  - For a given value of w
    \[ \text{Umax} = 2 \times \text{Tmax} + 1 \]
  - Range of two’s complement not symmetric
    \[ |\text{Tmin}| = |\text{Tmax}| + 1 \]

- In C…
  - These ranges are system specific. Therefore, to reference them we must `#include <limits.h>`
  - Declares constants, e.g.,
    - UINT_MAX
    - INT_MAX
    - INT_MIN
  - See `limits.c`
Comparison of unsigned & signed

- **Equivalence**
  - Same encodings for non-negative values.
  - +/- 16 for negative two’s complement and positive unsigned 4-bit values.

- **Uniqueness**
  - Every bit pattern represents a unique integer value.
  - Every integer has a unique bit pattern.

<table>
<thead>
<tr>
<th>$X$</th>
<th>B2U($X$)</th>
<th>B2T($X$)</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>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
</tr>
</tbody>
</table>