Representing Data with Bits

bits, bytes, numbers, and notation

position value = digit value \times base^{position}

base = 10 (decimal)

\[
\begin{array}{cccc}
100 & 10 & 1 & 0 \\
10^2 & 10^1 & 10^0 \\
2 & 1 & 0
\end{array}
\]

= 2 \times 10^2 + 4 \times 10^1 + 0 \times 10^0

Base determines:
- Maximum digit (base – 1). Minimum digit is 0.
- Weight of each position.
- Each position holds a digit.
- Represented value = sum of all position values

\[
\text{position value} = \text{digit value} \times \text{base}^{\text{position}}
\]

binary = base 2

Binary digits are called \textit{bits}: 0, 1

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
8 & 4 & 2 & 1 \\
2^3 & 2^2 & 2^1 & 2^0 \\
3 & 2 & 1 & 0
\end{array}
\]

= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0

When ambiguous, subscript with base:

101_{10} \text{ Dalmatians} \quad (\text{movie})

101_2 \text{- Second Rule} \quad (\text{folk wisdom for food safety})

Base = 2 (binary)

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
8 & 4 & 2 & 1 \\
2^3 & 2^2 & 2^1 & 2^0 \\
3 & 2 & 1 & 0
\end{array}
\]

Powers of 2:
memorize up to \geq 2^{10} (in base ten)

<table>
<thead>
<tr>
<th>Power: 2^x</th>
<th>Decimal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
<tr>
<td>11</td>
<td>2048</td>
</tr>
<tr>
<td>12</td>
<td>4096</td>
</tr>
<tr>
<td>13</td>
<td>8192</td>
</tr>
</tbody>
</table>
conversion from binary to decimal  

\[ 1 \ 0 \ 1 \ 1 \ 0 \ 1_2 = ?_{10} \]

Interpret the positional representation according to the base: sum the place weights where 1 appears (in either direction).

conversion from decimal to binary  

\[ 19_{10} = ?_2 \]

<table>
<thead>
<tr>
<th>Quotient</th>
<th>Remainder?</th>
<th>Value</th>
<th>Power that fits?</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>12</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Powers-of-2 Approach (Left to Right)

Divide-by-2 Approach (Right to Left)

binary arithmetic  

\[ 110_2 + 1011_2 = ?_2 \]
\[ 1101_2 - 1011_2 = ?_2 \]

\[ 1001011_2 \times 2_{10} = ?_2 \]

conversion and arithmetic  

\[ 19_{10} = ?_2 \]
\[ 1001_2 = ?_{10} \]

\[ 240_{10} = ?_2 \]
\[ 11010011_2 = ?_{10} \]

\[ 101_2 + 1011_2 = ?_2 \]
\[ 1001011_2 \times 2_{10} = ?_2 \]
**byte = 8 bits**

a.k.a. octet

Smallest unit of data

*used by a typical modern computer*

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>

Binary: 00000000₂ -- 11111111₂

Decimal: 000₁₀ -- 255₁₀

Hexadecimal: 00₁₆ -- FF₁₆

*Byte = 2 hex digits!*

Programmer’s hex notation (C, etc.):

0xB4 = B₄₁₆

Octal (base 8) also useful.

---

**char: representing characters**

A C-style string is represented by a series of bytes (`chars`).

- One-byte ASCII codes for each character.
- ASCII = American Standard Code for Information Interchange

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0x0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0x1</td>
<td>1</td>
</tr>
<tr>
<td>02</td>
<td>0x2</td>
<td>2</td>
</tr>
<tr>
<td>03</td>
<td>0x3</td>
<td>3</td>
</tr>
<tr>
<td>04</td>
<td>0x4</td>
<td>4</td>
</tr>
<tr>
<td>05</td>
<td>0x5</td>
<td>5</td>
</tr>
<tr>
<td>06</td>
<td>0x6</td>
<td>6</td>
</tr>
<tr>
<td>07</td>
<td>0x7</td>
<td>7</td>
</tr>
<tr>
<td>08</td>
<td>0x8</td>
<td>8</td>
</tr>
<tr>
<td>09</td>
<td>0x9</td>
<td>9</td>
</tr>
<tr>
<td>0A</td>
<td>0xA</td>
<td>10</td>
</tr>
<tr>
<td>0B</td>
<td>0xB</td>
<td>11</td>
</tr>
<tr>
<td>0C</td>
<td>0xC</td>
<td>12</td>
</tr>
<tr>
<td>0D</td>
<td>0xD</td>
<td>13</td>
</tr>
<tr>
<td>0E</td>
<td>0xE</td>
<td>14</td>
</tr>
<tr>
<td>0F</td>
<td>0xF</td>
<td>15</td>
</tr>
</tbody>
</table>

**word | word | n.**

Natural unit of data used by processor.

- **Fixed size** (e.g. 32 bits, 64 bits)
  - Defined by ISA: Instruction Set Architecture
  - Machine instruction operands
  - word size = register size = address size

```
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

Java/C int = 4 bytes: 11,501,584

MSB: most significant bit

LSB: least significant bit

---

fixed-size data representations

<table>
<thead>
<tr>
<th>Java Data Type</th>
<th>C Data Type</th>
<th>(size in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>byte</td>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>short int</td>
<td>2</td>
</tr>
<tr>
<td>short</td>
<td>short int</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>long int</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>long double</td>
<td>8</td>
</tr>
</tbody>
</table>

Depends on word size!
**bitwise operators**

**Bitwise operators on fixed-width bit vectors.**

<table>
<thead>
<tr>
<th>AND &amp;</th>
<th>OR</th>
<th>XOR ^</th>
<th>NOT ~</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101001 &amp; 01010101</td>
<td>01101001</td>
<td>01101001 ^ 01010101</td>
<td>~01010101</td>
</tr>
<tr>
<td>01000001</td>
<td>~01010101</td>
<td>01010101</td>
<td>^ 01010101</td>
</tr>
</tbody>
</table>

Laws of Boolean algebra apply bitwise.

*E.g.*, DeMorgan’s Law: \(\sim(A \lor B) = \sim A \land \sim B\)

---

**bitwise operators in C**

& | ^ | ~

apply to any *integral* data type

long, int, short, char, unsigned

Examples (*char*)

\(~0x41\) =

\(~0x00\) =

\(0x69 \& 0x55\) =

\(0x69 \mid 0x55\) =

Many bit-twiddling puzzles in upcoming assignment
### Representation Example 1: Sets as Bit Vectors

**Representation:** An *n*-bit vector gives a subset of \( \{0, \ldots, n-1\} \).

<table>
<thead>
<tr>
<th>Bitwise Operations</th>
<th>Set Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \land b )</td>
<td>Intersection</td>
</tr>
<tr>
<td>( a \lor b )</td>
<td>Union</td>
</tr>
<tr>
<td>( a \oplus b )</td>
<td>Symmetric difference</td>
</tr>
<tr>
<td>( \neg a )</td>
<td>Complement</td>
</tr>
</tbody>
</table>

**Examples:**
- \( a = 0b01101001 \quad A = \{0, 3, 5, 6\} \)
- \( b = 0b01010101 \quad B = \{0, 2, 4, 6\} \)

### logical operations in C

**& & | | !**

- apply to any "integral" data type
- long, int, short, char, unsigned

- 0 is false
- nonzero is true
- result always 0 or 1

- early termination
- a.k.a. short-circuit evaluation

**Examples (char):**
- \( !0x41 = 0x58 \)
- \( !0x00 = 0xff \)
- \( 0x69 \land 0x55 = 0x55 \)
- \( 0x69 \lor 0x55 = 0x6a \)
Representation Example 2: Playing Cards

52 cards in 4 suits
How do we encode suits, face cards?

What operations should be easy to implement?
Get and compare rank
Get and compare suit

Two possible representations

52 cards – 52 bits with bit corresponding to card set to 1

“One-hot” encoding
Hard to compare values and suits independently
Not space efficient

4 bits for suit, 13 bits for card value – 17 bits with two set to 1

Pair of one-hot encoded values
Easier to compare suits and values independently
Smaller, but still not space efficient

Two better representations

Binary encoding of all 52 cards – only 6 bits needed
Number cards uniquely from 0
Smaller than one-hot encodings.
Hard to compare value and suit

Binary encoding of suit (2 bits) and value (4 bits) separately
Number each suit uniquely
Number each value uniquely
Still small
Easy suit, value comparisons

Compare Card Suits

mask: a bit vector that, when bitwise ANDed with another bit vector v, turns all but the bits of interest in v to 0

#define SUIT_MASK 0x30

int sameSuit(char card1, char card2) {
    return !((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK));
    //same as (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}

char hand[5]; // represents a 5-card hand...
if (sameSuit(hand[0], hand[1])) { ... }
Compare Card Values

```c
#define VALUE_MASK

int greaterValue(char card1, char card2) {
    // represents a 5-card hand
    char hand[5];
    ...
    if (greaterValue(hand[0], hand[1])) { ... }
}
```

Bit shifting

```c
logical shift left 2
x << 2
```

```c
logical shift right 2
x >> 2
```

Shift gotchas

**Logical or arithmetic shift right: how do we tell?**

- C: compiler chooses
  - Usually based on type: rain check!
  - Java: >> is arithmetic, >>> is logical

**Shift an n-bit type by at least 0 and no more than (n-1).**

- C: other shift distances are undefined.
  - anything could happen
  - Java: shift distance is used modulo number of bits in shifted type
    
    ```c
    Given int x:    x << 34 == x << 2
    ```

Shift and mask: extract a bit field

```c
int get2ndMSB(int x) {
    // write a C function
    // that extracts the 2\textsuperscript{nd} most significant byte
    // from its 32-bit integer argument.
    // Example behavior:
    // argument: 0b 01100001 01100010 01100011 01100100
    // expected result: 0b 00000000 00000000 00000000 01100010
    // All other bits are zero. Desired bits in least significant byte.
```