Representing Data with Bits

bits, bytes, numbers, and notation

positional number representation

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 bits: 0, 1

When ambiguous, subscript with base:
- 101_{10} Dalmatians (movie)
- 101_{2} Second Rule (folk wisdom for food safety)

Powers of 2:

memorize up to \(2^{10}\) (in base ten)
conversion from binary to decimal

101101₂ = ?₁₀

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

conversion from decimal to binary

19₁₀ = ?₂

Divide-by-2 Approach (Right to Left)

Quotient    Remainder?

Powers-of-2 Approach (Left to Right)

Value    Power that fits?

binary arithmetic

110₂ + 1011₂ = ?₂
110₁₂ − 1011₂ = ?₂

1001011₂ × 2₁₀ = ?₂

conversion and arithmetic

19₁₀ = ?₂
100₁₂ = ?₁₀

240₁₀ = ?₂
11010011₂ = ?₁₀

101₂ + 1011₂ = ?₂
1001011₂ × 2₁₀ = ?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 00000002 -- 11111112
Decimal 00010 -- 25510
Hexadecimal 0016 -- FF16

**What do you call 4 bits?**

**a.k.a. octet**

Programmer’s hex notation (C, etc.):

\[ \text{0xB4} = B4_{16} \]

Octal (base 8) also useful.

---

**char: representing characters**

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

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

**fixed-size data representations**

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

<table>
<thead>
<tr>
<th>Java Data Type</th>
<th>C Data Type</th>
<th>[word = 32 bits]</th>
<th>[word = 64 bits]</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>byte</td>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>short int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>short</td>
<td>int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>long double</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

Depends on word size!
**bitwise operators**

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

- AND &
- OR |
- XOR ^
- NOT ~

Laws of Boolean algebra apply bitwise.

- e.g., DeMorgan’s Law: ~(A | B) = ~A & ~B

**Example**

```
01101001 & 01010101 = 01000001
01101001 | 01010101 = 01101001
01101001 ^ 01010101 = 01010100
~ 01010101 = 01010100
```

**Representation Example 1:**
Sets as Bit Vectors

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

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

**Bitwise Operations**

- \( a \& b = 0b01000001 \) \( \{0, 6\} \)
- \( a | b = 0b01111101 \) \( \{0, 2, 3, 4, 5, 6\} \)
- \( a ^ b = 0b00111100 \) \( \{2, 3, 4, 5\} \)
- \( ~a = 0b10101010 \) \( \{1, 3, 5, 7\} \)

**Set Operations**

- \( a \cap b = 0b00000001 \) \( \{0\} \)
- \( a \cup b = 0b10111101 \) \( \{0, 2, 3, 4, 5, 6\} \)
- \( a \Delta b = 0b00111100 \) \( \{2, 3, 4, 5\} \)
- \( ~b = 0b10010101 \) \( \{1, 3, 5, 7\} \)
**Logical operations in C**

```c
&&  ||  !
```

apply to any "integral" data type

```c
long, int, short, char, unsigned
```

0 is false

```c
nonzero is true
```

result always 0 or 1

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

**Examples (char)**

```c
!0x41 =
!0x00 =
!!0x41 =
```

```c
0x69 & 0x55 =
0x69 || 0x55 =
```

---

**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

```
```

52 bits in 2 x 32-bit words

“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

```c
#define SUIT_MASK 0x30

int sameSuit(char card1, char card2) {
    return !(card1 & SUIT_MASK) ^ (card2 & SUIT_MASK);
}
```

```c
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) {
    // same as (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

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

Bit shifting

**Logical shift left 2**

```
1 0 0 1 1 0 0 1
x << 2

0 1 1 0 0 1 0 0
```

**Logical shift right 2**

```
1 0 0 1 1 0 0 1
x >> 2

0 0 1 1 0 1 1 0
```

**Arithmetic shift right 2**

```
1 1 0 0 1 1 0 0 1
x >> 2

1 0 0 1 1 0 0 1 0 1
```

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

Given `int x; x << 34 == x << 2`
Shift and mask: extract a bit field

Write a C function that extracts the 2nd most significant byte from its 32-bit integer argument.

Example behavior:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b 01100001 01100010 01100111 01100100</td>
<td>0b 00000000 00000000 00000000 01100010</td>
</tr>
</tbody>
</table>

All other bits are zero. Desired bits in least significant byte.

```c
int get2ndMSB(int x) {
    // Implementation
}
```