Representing Data with Bits

bits, bytes, numbers, and notation

https://cs.wellesley.edu/~cs240/s20/

positional number representation

\[
\begin{array}{ccc}
2 & 4 & 0 \\
100 & 10 & 1 \\
10^2 & 10^1 & 10^0 \\
2 & 1 & 0 \\
\end{array}
\]

weight

position

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

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

101_{10} Dalmatians (movie)

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

binary = base 2

When ambiguous, subscript with base:

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

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

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

19\(_{10}\) = ?\(_2\)  
1001\(_2\) = ?\(_{10}\)

240\(_{10}\) = ?\(_2\)  
11010011\(_2\) = ?\(_{10}\)

101\(_2\) + 1011\(_2\) = ?\(_2\)  
1001011\(_2\) \times 2\(_{10}\) = ?\(_2\)

### numbers and wires

One wire carries one bit.  
How many wires to represent a given number?

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

What if I want to build a computer (and not change the hardware later)?

### byte = 8 bits

**Smallest unit of data**  
*a.k.a. octet*

- **Binary** 00000000\(_2\) -- 11111111\(_2\)
- **Decimal** 000\(_{10}\) -- 255\(_{10}\)
- **Hexadecimal** 00\(_{16}\) -- FF\(_{16}\)

*Byte = 2 hex digits!*

Programmer’s hex notation (C, etc.):  
0xB4 = B4\(_{16}\)

Octal (base 8) also useful.

### Hex encoding practice

**What do you call 4 bits?**

**a.k.a. octet**

<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>
**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>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>00000001</td>
</tr>
<tr>
<td>2</td>
<td>00000010</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>127</td>
<td>01111111</td>
</tr>
</tbody>
</table>

**word** /ˈwɜːrd/, 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

![Data as Bits](image)

**fixed-size data representations**

<table>
<thead>
<tr>
<th>Java Data Type</th>
<th>C Data Type</th>
<th>32-bit</th>
<th>64-bit</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>short</td>
<td>short 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>long</td>
<td>long int</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>double</td>
<td>double</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 ~

![Data as Bits](image)
Aside: sets as bit vectors

Representation: \( n \)-bit vector gives subset of \( \{0, ..., n-1\} \).
\[
a_i = 1 \iff i \in A
\]

\[
\begin{align*}
01101001 &= \{0, 3, 5, 6\} \\
76543210 &
\end{align*}
\]

\[
\begin{align*}
01010101 &= \{0, 2, 4, 6\} \\
76543210 &
\end{align*}
\]

Bitwise Operations
\[
\begin{array}{c|c}
\& & \{0, 6\} \\
| & \{0, 2, 3, 4, 5, 6\} \\
^ & \{2, 3, 4, 5\} \\
\sim & \{1, 3, 5, 7\}
\end{array}
\]

Set Operations?

Bitwise operators in C

\& | ^ ~ apply to any integral data type
\[
\begin{align*}
\text{long, int, short, char, unsigned}
\end{align*}
\]

Examples (char)
\[
\begin{align*}
\sim0x41 &= \\
\sim0x00 &=
\end{align*}
\]

\[
\begin{align*}
0x69 &\ & 0x55 = \\
0x69 &|| 0x55 =
\end{align*}
\]

Many bit-twiddling puzzles in upcoming assignment

logical operations in C

\&\& | | | ! apply to any "integral" data type
\[
\begin{align*}
\text{long, int, short, char, unsigned}
\end{align*}
\]

0 is false nonzero is true result always 0 or 1

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

Examples (char)
\[
\begin{align*}
!0x41 &= \\
!0x00 &= \\
!!0x41 &=
\end{align*}
\]

\[
\begin{align*}
0x69 &\ & 0x55 = \\
0x69 | | 0x55 =
\end{align*}
\]

Encode 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

52-card deck
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
char card1, card2;  // two cards to compare
...
if (sameSuit(hand[0], hand[1])) { ... }

Compare Card Values

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 VALUE_MASK

int greaterValue(char card1, char card2) {
}

char hand[5];       // represents a 5-card hand
char card1, card2;  // two cards to compare
...
if (greaterValue(hand[0], hand[1])) { ... }
Bit shifting

\[ x = \begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
\end{array} \]

\[ x << 2 = \begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]

logical shift left 2
lose bits on left
fill with zeroes on right

\[ x \]

\[ \begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
\end{array} \]

\[ x >> 2 = \begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]

logical shift right 2
fill with zeroes on left
lose bits on right

\[ x \]

\[ \begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
\end{array} \]

arithmetic shift right 2
fill with copies of MSB on left


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.

\[ \text{anything could happen} \]
Java: shift distance is used modulo number of bits in shifted type

Given \( \text{int} \ x: \quad x << 34 = x << 2 \)

Shift and mask: extract a bit field

Write a C function that extracts the 2\( \text{nd} \) most significant byte from its 32-bit integer argument.

Example behavior:

argument: \( 0b \ 01100001 \ 01100010 \ 01100011 \ 01100100 \)

equilned result: \( 0b \ 00000000 \ 00000000 \ 00000000 \ 01100010 \)

\[ \text{All other bits are zero.} \quad \text{Desired bits in least significant byte.} \]

\[ \text{int } \text{get2ndMSB}(\text{int} \ x) \{ \]