Representing Data with Bits

bits, bytes, numbers, and notation

Show powers, strategies.

conversion and arithmetic

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

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

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

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

\[ \text{byte} = 8 \text{ bits} \]
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>

Programmer’s hex notation (C, etc.):
\[ 0xB4 = B4_{16} \]

Octal (base 8) also useful.

Why do 240 students often confuse Halloween and Christmas?
Hex encoding practice

**bitwise operators**

Bitwise operators on fixed-width bit vectors.

<table>
<thead>
<tr>
<th>&amp;</th>
<th></th>
<th>^</th>
<th>~</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101001 &amp; 01010101</td>
<td>01010101 ^ 01010101</td>
<td>01101001 ^ 01010101</td>
<td>01010101 &amp; 01010101</td>
</tr>
</tbody>
</table>

| 01000011 | 01101001 \^ 01010101 | 01101001 \& 01010101 | 01010101 |

Laws of Boolean algebra apply bitwise.
e.g., DeMorgan’s Law: ~(A | B) = ~A & ~B

**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 & \\
01010101 & \{0, 2, 4, 6\} \\
76543210 & \\
\end{align*}

**Bitwise Operations**

<table>
<thead>
<tr>
<th>&amp;</th>
<th></th>
<th>^</th>
<th>~</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000001 &amp; 01111101</td>
<td>00111100 &amp; 10101010</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**Set Operations?**

01111101 \& 00000000 =

00111100 \^ 00000000 =

Many bit-twiddling puzzles in upcoming assignment
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 =
!0x00 =
!!0x41 =
0x69 && 0x55 =
0x69 || 0x55 =

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

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**

A mask is a bit vector that, when bitwise ANDed with another bit vector \( v \), turns all but the bits of interest in \( v \) to 0.

```c
#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);
}
```

**Compare Card Values**

A mask is a bit vector that, when bitwise ANDed with another bit vector \( v \), turns all but the bits of interest in \( v \) to 0.

```c
#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**

- Logical shift left 2:
  - Lose bits on left
  - Fill with zeroes on right

- Logical shift right 2:
  - Fill with zeroes on left

- Arithmetic shift right 2:
  - Fill with copies of MSB on left

**Shift and Mask: extract a bit field**

Write C code:

- Extract 2\textsuperscript{nd} most significant byte from a 32-bit integer.

Given \( x = 01100001 \) 01100010 01100100, should return:

\[ 00000000 00000000 00000000 01100010 \]

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