Representing Data with Bits

bits, bytes, numbers, and notation

conversion and arithmetic

19₁₀ = ?₂
1001₂ = ?₁₀

240₁₀ = ?₂
11010011₂ = ?₁₀

101₂ + 1011₂ = ?₂
1001011₂ x 2₁₀ = ?₂

binary = base 2

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

= 1 x 2³ + 0 x 2² + 1 x 2¹ + 1 x 2⁰

When ambiguous, subscript with base:

101₁₀ Dalmatians (movie)

101₂-Second Rule (folk wisdom for food safety)

Irony

Show powers, strategies.

conversion and arithmetic

19₁₀ = ?₂
1001₂ = ?₁₀

240₁₀ = ?₂
11010011₂ = ?₁₀

101₂ + 1011₂ = ?₂
1001011₂ x 2₁₀ = ?₂

byte = 8 bits

A.k.a. octet

Smallest unit of data

*used by a typical modern computer*

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000₀₂</td>
<td>00₁₀</td>
<td>0₀₁₆</td>
</tr>
<tr>
<td>1111111₁₂</td>
<td>25₅₁₀</td>
<td>FF₁₆</td>
</tr>
</tbody>
</table>

Byte = 2 hex digits!

Programmer’s hex notation (C, etc.):

0xB₄ = B₄₁₆

Octal (base 8) also useful.

Why do 240 students often confuse Halloween and Christmas?
**word** /wərd/, 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 |
| 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0                |
+--------------------------------------------------------------------------+
```

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

**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>32-bit 64-bit</td>
</tr>
<tr>
<td>byte</td>
<td>char</td>
<td>1 1</td>
</tr>
<tr>
<td>char</td>
<td>short int</td>
<td>2 2</td>
</tr>
<tr>
<td>short</td>
<td>int</td>
<td>2 2</td>
</tr>
<tr>
<td>int</td>
<td>float</td>
<td>4 4</td>
</tr>
<tr>
<td>float</td>
<td>long int</td>
<td>4 8</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>8 8</td>
</tr>
<tr>
<td>long</td>
<td>long long</td>
<td>8 8</td>
</tr>
<tr>
<td></td>
<td>long double</td>
<td>8 16</td>
</tr>
</tbody>
</table>

**bitwise operators**

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

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

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

01010101  ^ 01010101

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

**bitwise operators in C**

& | ^ ~ apply to any **integral** data type

Examples (**char**)  
~0x41 =  
~0x00 =  
0x69 & 0x55 =  
0x69 | 0x55 =  

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**

mask: 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);
}
```

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

```c
#define VALUE_MASK
```

```c
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**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td>10011001</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>x &lt;&lt; 2</code></td>
<td>01100100</td>
<td>logical shift left 2</td>
<td>lose bits on left, fill with zeroes on right</td>
</tr>
<tr>
<td><code>x &gt;&gt; 2</code></td>
<td>00110010</td>
<td>logical shift right 2</td>
<td>fill with zeroes on left</td>
</tr>
<tr>
<td><code>x &gt;&gt; 2</code></td>
<td>11100110</td>
<td>arithmetic shift right 2</td>
<td>fill with copies of MSB on left</td>
</tr>
</tbody>
</table>

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

Write C code:

extract 2\(^{nd}\) most significant byte from a 32-bit integer.

given \( x = \) 01100001 01100010 01100011 01100100

should return:

\( 00000000 00000000 00000000 01100010 \)

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