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

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

**Binary = Base 2**

When ambiguous, subscript with base:
- 101_{10} Dalmatians (movie)
- 101_2 Second Rule (folk wisdom for food safety)

Powers of 2:
memorize up to \geq 2^{10} (in base ten)
Show powers, strategies.

conversion and arithmetic

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

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

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

Data as Bits

Hex encoding practice

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
**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

---

**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>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>8</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>long</td>
<td>long long</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

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

MSB: most significant bit  
LSB: least significant bit

**bitwise operators**

**Bitwise operators** on fixed-width **bit vectors**.
- AND &  
- OR |  
- XOR ^  
- NOT ~

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

$$ a_i = 1 \equiv i \in A $$

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

Laws of Boolean algebra apply bitwise.  
- e.g., DeMorgan’s Law: $\neg(A \lor B) = \neg A \land \neg B$  
- ex  

**Aside: sets as bit vectors**

<table>
<thead>
<tr>
<th>Bitwise Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000001 &amp; 01010101</td>
</tr>
<tr>
<td>01111101</td>
</tr>
<tr>
<td>^ 00111100 ^ 10101010</td>
</tr>
<tr>
<td>~ 01010101 ~ 10101010</td>
</tr>
</tbody>
</table>

**Set Operations?**
**bitwise operators in C**

& | ^ ~ apply to any integral data type long, int, short, char, unsigned

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

```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

```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

```
x 1 0 0 1 1 0 0 1

x << 2 1 0 1 1 0 0 1 0 logical shift left 2
lose bits on left fill with zeroes on right

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

fill with zeroes on left logical shift right 2 0 0 1 1 0 1 0 0
lose bits on right

fill with copies of MSB on left arithmetic shift right 2 1 1 0 0 1 1 0 0 1 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
    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 01100011 01100100</td>
<td>0b 00000000 00000000 00000000 01100010</td>
</tr>
</tbody>
</table>

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

int get2ndMSB(int x) {