Representing Data with Bits

bits, bytes, numbers, and notation

**binary = base 2**

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

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

When ambiguous, subscript with base:

- 101_10 Dalmatians (movie)
- 101_2 Second Rule (folk wisdom for food safety)

**positional number representation**

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

- 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}

\[= 2 \times 10^2 + 4 \times 10^1 + 0 \times 10^0\]

**Powers of 2:**

learn up to \(\geq 2^{10}\) (in base ten)
Show powers, strategies.

**conversion and arithmetic**

$19_{10} = ?_2$

$1001_{2} = ?_{10}$

$240_{10} = ?_2$

$11010011_{2} = ?_{10}$

$101_{2} + 1011_{2} = ?_2$

$1001011_{2} \times 2_{10} = ?_{2}$

---

**numbers and wires**

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

$1001$

$10001001$

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

---

**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>

Decoder #1

Decoder #2

Octal (base 8) also useful.

Why do 240 students often confuse Halloween and Christmas?
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>32</th>
<th>40</th>
<th>48</th>
<th>56</th>
<th>64</th>
<th>72</th>
<th>80</th>
<th>88</th>
<th>96</th>
<th>104</th>
<th>112</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;a&quot;</td>
<td>&quot;b&quot;</td>
<td>&quot;c&quot;</td>
<td>&quot;d&quot;</td>
<td>&quot;e&quot;</td>
<td>&quot;f&quot;</td>
<td>&quot;g&quot;</td>
<td>&quot;h&quot;</td>
<td>&quot;i&quot;</td>
<td>&quot;j&quot;</td>
<td>&quot;k&quot;</td>
</tr>
<tr>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
</tr>
<tr>
<td>&quot;l&quot;</td>
<td>&quot;m&quot;</td>
<td>&quot;n&quot;</td>
<td>&quot;o&quot;</td>
<td>&quot;p&quot;</td>
<td>&quot;q&quot;</td>
<td>&quot;r&quot;</td>
<td>&quot;s&quot;</td>
<td>&quot;t&quot;</td>
<td>&quot;u&quot;</td>
<td>&quot;v&quot;</td>
</tr>
<tr>
<td>108</td>
<td>109</td>
<td>110</td>
<td>111</td>
<td>112</td>
<td>113</td>
<td>114</td>
<td>115</td>
<td>116</td>
<td>117</td>
<td>118</td>
</tr>
<tr>
<td>&quot;w&quot;</td>
<td>&quot;x&quot;</td>
<td>&quot;y&quot;</td>
<td>&quot;z&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>119</td>
<td>120</td>
<td>121</td>
<td>122</td>
<td>123</td>
<td>124</td>
<td>125</td>
<td>126</td>
<td>127</td>
<td>128</td>
<td>129</td>
</tr>
<tr>
<td>^</td>
<td>^</td>
<td>^</td>
<td>^</td>
<td>^</td>
<td>^</td>
<td>^</td>
<td>^</td>
<td>^</td>
<td>^</td>
<td>^</td>
</tr>
</tbody>
</table>

word | word, 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

MSB: most significant bit
LSB: least significant bit

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>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>byte</td>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>short</td>
<td>short int</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>long double</td>
<td>8</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 ~

01101001 | 01101001 | 01101001 | 01101001
& 01010101 | 01010101 | 01010101 | 01010101
01000001 | 01010101 | 01010101 | 01010101

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 \( \{0, ..., n-1\} \).

\( a_i = 1 \iff i \in A \)

\[ \begin{align*}
01101001 & \quad \{0, 3, 5, 6\} \\
76543210 & \\
01010101 & \quad \{0, 2, 4, 6\} \\
76543210 &
\end{align*} \]

Bitwise Operations

\[ \begin{align*}
& \quad \& & 01000001 & \quad \{0, 6\} \\
| & 01111101 & \quad \{0, 2, 3, 4, 5, 6\} \\
^ & 00111100 & \quad \{2, 3, 4, 5\} \\
\sim & 10101010 & \quad \{1, 3, 5, 7\}
\end{align*} \]

Set Operations?

bitwise operations in C

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

Examples (char)

\[ \begin{align*}
\sim0x41 & = \\
\sim0x00 & = \\
0x69 & \& 0x55 = \\
0x69 & | 0x55 =
\end{align*} \]

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)

\[ \begin{align*}
!0x41 & = \\
!0x00 & = \\
!!0x41 & = \\
0x69 & \& 0x55 = \\
0x69 & | | 0x55 =
\end{align*} \]

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
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

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** (x << 2)
  - Lose bits on left
  - Fill with zeroes on right

- **Logical shift right** (x >> 2)
  - Lose bits on right

- **Arithmetic shift right** (x >> 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.
    - 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 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.