Representing Data with Bits

bits, bytes, numbers, and notation
bit = binary digit = 0 or 1

Electronically: high voltage vs. low voltage

Basis of all digital representations
ints, floats, chars, arrays, objects, strings, booleans...
machine instructions
position 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}

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

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

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

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

When ambiguous, subscript with base:

- \(101_{10}\) Dalmatians (movie)
- \(101_2\)-Second Rule (folk wisdom for food safety)
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**

a.k.a. octet

Smallest unit of data
used by a typical modern computer

<table>
<thead>
<tr>
<th></th>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
<td>1111</td>
</tr>
</tbody>
</table>

Binary 00000000₂ -- 11111111₂
Decimal 000₁₀ -- 255₁₀
Hexadecimal 00₁₆ -- FF₁₆

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?
Hex encoding practice
**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>space</th>
<th>48</th>
<th>0</th>
<th>64</th>
<th>@</th>
<th>80</th>
<th>P</th>
<th>96</th>
<th>`</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>!</td>
<td>49</td>
<td>1</td>
<td>65</td>
<td>A</td>
<td>81</td>
<td>Q</td>
<td>97</td>
<td>a</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
<td>50</td>
<td>2</td>
<td>66</td>
<td>B</td>
<td>82</td>
<td>R</td>
<td>98</td>
<td>b</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
<td>51</td>
<td>3</td>
<td>67</td>
<td>C</td>
<td>83</td>
<td>S</td>
<td>99</td>
<td>c</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
<td>52</td>
<td>4</td>
<td>68</td>
<td>D</td>
<td>84</td>
<td>T</td>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
<td>53</td>
<td>5</td>
<td>69</td>
<td>E</td>
<td>85</td>
<td>U</td>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
<td>54</td>
<td>6</td>
<td>70</td>
<td>F</td>
<td>86</td>
<td>V</td>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>39</td>
<td>'</td>
<td>55</td>
<td>7</td>
<td>71</td>
<td>G</td>
<td>87</td>
<td>W</td>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>56</td>
<td>8</td>
<td>72</td>
<td>H</td>
<td>88</td>
<td>X</td>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
<td>57</td>
<td>9</td>
<td>73</td>
<td>I</td>
<td>89</td>
<td>Y</td>
<td>105</td>
<td>l</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
<td>58</td>
<td>:</td>
<td>74</td>
<td>J</td>
<td>90</td>
<td>Z</td>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
<td>59</td>
<td>;</td>
<td>75</td>
<td>K</td>
<td>91</td>
<td>[</td>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
<td>60</td>
<td>&lt;</td>
<td>76</td>
<td>L</td>
<td>92</td>
<td>\</td>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
<td>61</td>
<td>=</td>
<td>77</td>
<td>M</td>
<td>93</td>
<td>]</td>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
<td>62</td>
<td>&gt;</td>
<td>78</td>
<td>N</td>
<td>94</td>
<td>^</td>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
<td>63</td>
<td>?</td>
<td>79</td>
<td>O</td>
<td>95</td>
<td>_</td>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>80</td>
<td>P</td>
<td>96</td>
<td>`</td>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>81</td>
<td>Q</td>
<td>97</td>
<td>a</td>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>82</td>
<td>R</td>
<td>98</td>
<td>b</td>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>83</td>
<td>S</td>
<td>99</td>
<td>c</td>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>84</td>
<td>T</td>
<td>100</td>
<td>d</td>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>85</td>
<td>U</td>
<td>101</td>
<td>e</td>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>86</td>
<td>V</td>
<td>102</td>
<td>f</td>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>87</td>
<td>W</td>
<td>103</td>
<td>g</td>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>88</td>
<td>X</td>
<td>104</td>
<td>h</td>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89</td>
<td>Y</td>
<td>105</td>
<td>l</td>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>90</td>
<td>Z</td>
<td>106</td>
<td>j</td>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>91</td>
<td>[</td>
<td>107</td>
<td>k</td>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92</td>
<td>\</td>
<td>108</td>
<td>l</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>93</td>
<td>]</td>
<td>109</td>
<td>m</td>
<td>125</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>94</td>
<td>^</td>
<td>110</td>
<td>n</td>
<td>126</td>
<td>~</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>95</td>
<td>_</td>
<td>111</td>
<td>o</td>
<td>127</td>
<td>del</td>
</tr>
</tbody>
</table>
**word** |ward|, 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

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

**MSB:** most significant bit

**LSB:** least significant bit
# 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 1</td>
</tr>
<tr>
<td>byte</td>
<td>short int</td>
<td>2 2</td>
</tr>
<tr>
<td>char</td>
<td>int</td>
<td>4 4</td>
</tr>
<tr>
<td>short</td>
<td>long int</td>
<td>4 8</td>
</tr>
<tr>
<td>int</td>
<td>double</td>
<td>8 8</td>
</tr>
<tr>
<td>float</td>
<td>long long</td>
<td>8 8</td>
</tr>
<tr>
<td>long</td>
<td>long double</td>
<td>8 16</td>
</tr>
</tbody>
</table>

*Depends on word size!*
**bitwise operators**

Bitwise operators on fixed-width bit vectors.

AND & OR | XOR ^ NOT ~

\[
\begin{array}{c}
01101001 \\
& 01010101 \\
\hline
01000001
\end{array}
\begin{array}{c}
01101001 \\
| 01010101 \\
\hline
01101001
\end{array}
\begin{array}{c}
01101001 \\
^ 01010101 \\
\hline
~ 01010101
\end{array}
\]

Laws of Boolean algebra apply.

\textit{e.g.}, DeMorgan’s Law: \( \sim(A \mid B) = \sim A \& \sim B \)
Aside: sets as bit vectors

Representation: $n$-bit vector gives subset of $\{0, \ldots, n-1\}$. 
\[ a_i = 1 \equiv i \in A \]

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

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

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

**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
```

```c
int sameSuit(char card1, char card2) {
    return 0 == ((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) {
}
```

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

\[ x \]

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

\[ \text{logical shift left 2} \]

\[ x \ll 2 \]

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

fill with zeroes on right

lose bits on left

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

x

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

arithmetic shift right 2

fill with copies of MSB on left

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

x \gg 2

lose bits on right

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

fill with zeroes on left

logical shift right 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 C code:

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

given \(\times = \)

| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

should return:

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

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