Data as Bits
Number Systems

- Why do humans use decimal?
  - Another Question: Why do some humans have unique words for specific numbers, and 10 unique digits (0-9)?

- Why do computers use binary?
**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\(^{position}\)

```
   2  4  0  = 2 \times 10^2 + 4 \times 10^1 + 0 \times 10^0
   100 10 1
   10^2 10^1 10^0
   2 1 0
```
binary = base 2

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

= $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

When ambiguous, subscript with base:

$101_{10}$ Dalmatians (movie)

$101_2$-Second Rule (folk wisdom for food safety)
## Powers of 2:

memorize up to $\geq 2^{10}$ (in base ten)

<table>
<thead>
<tr>
<th>$2^0$</th>
<th>$2^1$</th>
<th>$2^2$</th>
<th>$2^3$</th>
<th>$2^4$</th>
<th>$2^5$</th>
<th>$2^6$</th>
<th>$2^7$</th>
<th>$2^8$</th>
<th>$2^9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>$2^{11}$</td>
<td>$2^{12}$</td>
<td>$2^{13}$</td>
<td>$2^{14}$</td>
<td>$2^{15}$</td>
<td>$2^{16}$</td>
<td>$2^{17}$</td>
<td>$2^{18}$</td>
<td>$2^{19}$</td>
</tr>
</tbody>
</table>
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?

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

Smallest unit of data
used by a typical modern computer

Binary  \( \begin{align*}
00000000_2 & \quad \rightarrow \quad 11111111_2 \\
000_10 & \quad \rightarrow \quad 255_{10}
\end{align*} \)

Hexadecimal \( \begin{align*}
00_{16} & \quad \rightarrow \quad FF_{16}
\end{align*} \)

[Byte = 2 hex digits!]

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
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>ASCII</th>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>48</td>
<td>00010010</td>
<td>space</td>
</tr>
<tr>
<td>33</td>
<td>49</td>
<td>00010011</td>
<td>!</td>
</tr>
<tr>
<td>34</td>
<td>50</td>
<td>00010100</td>
<td>&quot;</td>
</tr>
<tr>
<td>35</td>
<td>51</td>
<td>00010101</td>
<td>#</td>
</tr>
<tr>
<td>36</td>
<td>52</td>
<td>00010110</td>
<td>$</td>
</tr>
<tr>
<td>37</td>
<td>53</td>
<td>00010111</td>
<td>%</td>
</tr>
<tr>
<td>38</td>
<td>54</td>
<td>00010110</td>
<td>&amp;</td>
</tr>
<tr>
<td>39</td>
<td>55</td>
<td>00010111</td>
<td>'</td>
</tr>
<tr>
<td>40</td>
<td>56</td>
<td>00010100</td>
<td>(</td>
</tr>
<tr>
<td>41</td>
<td>57</td>
<td>00010101</td>
<td>)</td>
</tr>
<tr>
<td>42</td>
<td>58</td>
<td>00010110</td>
<td>*</td>
</tr>
<tr>
<td>43</td>
<td>59</td>
<td>00010111</td>
<td>+</td>
</tr>
<tr>
<td>44</td>
<td>60</td>
<td>00011000</td>
<td>,</td>
</tr>
<tr>
<td>45</td>
<td>61</td>
<td>00011001</td>
<td>-</td>
</tr>
<tr>
<td>46</td>
<td>62</td>
<td>00011010</td>
<td>.</td>
</tr>
<tr>
<td>47</td>
<td>63</td>
<td>00011011</td>
<td>/</td>
</tr>
<tr>
<td>64</td>
<td>91</td>
<td>01010001</td>
<td>@</td>
</tr>
<tr>
<td>65</td>
<td>92</td>
<td>01010010</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>93</td>
<td>01010011</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>94</td>
<td>01010100</td>
<td>C</td>
</tr>
<tr>
<td>68</td>
<td>95</td>
<td>01010101</td>
<td>D</td>
</tr>
<tr>
<td>69</td>
<td>96</td>
<td>01010110</td>
<td>E</td>
</tr>
<tr>
<td>70</td>
<td>97</td>
<td>01010111</td>
<td>F</td>
</tr>
<tr>
<td>71</td>
<td>98</td>
<td>01011000</td>
<td>G</td>
</tr>
<tr>
<td>72</td>
<td>99</td>
<td>01011001</td>
<td>H</td>
</tr>
<tr>
<td>73</td>
<td>100</td>
<td>01011010</td>
<td>I</td>
</tr>
<tr>
<td>74</td>
<td>101</td>
<td>01011011</td>
<td>J</td>
</tr>
<tr>
<td>75</td>
<td>102</td>
<td>01011100</td>
<td>K</td>
</tr>
<tr>
<td>76</td>
<td>103</td>
<td>01011101</td>
<td>L</td>
</tr>
<tr>
<td>77</td>
<td>104</td>
<td>01011110</td>
<td>M</td>
</tr>
<tr>
<td>78</td>
<td>105</td>
<td>01011111</td>
<td>N</td>
</tr>
<tr>
<td>79</td>
<td>106</td>
<td>01100000</td>
<td>O</td>
</tr>
<tr>
<td>80</td>
<td>107</td>
<td>01100001</td>
<td>P</td>
</tr>
<tr>
<td>81</td>
<td>108</td>
<td>01100010</td>
<td>Q</td>
</tr>
<tr>
<td>82</td>
<td>109</td>
<td>01100011</td>
<td>R</td>
</tr>
<tr>
<td>83</td>
<td>110</td>
<td>01100100</td>
<td>S</td>
</tr>
<tr>
<td>84</td>
<td>111</td>
<td>01100101</td>
<td>T</td>
</tr>
<tr>
<td>85</td>
<td>112</td>
<td>01100110</td>
<td>U</td>
</tr>
<tr>
<td>86</td>
<td>113</td>
<td>01100111</td>
<td>V</td>
</tr>
<tr>
<td>87</td>
<td>114</td>
<td>01101000</td>
<td>W</td>
</tr>
<tr>
<td>88</td>
<td>115</td>
<td>01101001</td>
<td>X</td>
</tr>
<tr>
<td>89</td>
<td>116</td>
<td>01101010</td>
<td>Y</td>
</tr>
<tr>
<td>90</td>
<td>117</td>
<td>01101011</td>
<td>Z</td>
</tr>
<tr>
<td>91</td>
<td>118</td>
<td>01101100</td>
<td>[</td>
</tr>
<tr>
<td>92</td>
<td>119</td>
<td>01101101</td>
<td>\</td>
</tr>
<tr>
<td>93</td>
<td>120</td>
<td>01101110</td>
<td>]</td>
</tr>
<tr>
<td>94</td>
<td>121</td>
<td>01101111</td>
<td>^</td>
</tr>
<tr>
<td>95</td>
<td>122</td>
<td>01110000</td>
<td>_</td>
</tr>
<tr>
<td>96</td>
<td>123</td>
<td>01110001</td>
<td>`</td>
</tr>
<tr>
<td>97</td>
<td>124</td>
<td>01110010</td>
<td>a</td>
</tr>
<tr>
<td>98</td>
<td>125</td>
<td>01110011</td>
<td>b</td>
</tr>
<tr>
<td>99</td>
<td>126</td>
<td>01110100</td>
<td>c</td>
</tr>
<tr>
<td>100</td>
<td>127</td>
<td>01110101</td>
<td>d</td>
</tr>
<tr>
<td>101</td>
<td>128</td>
<td>01110110</td>
<td>e</td>
</tr>
<tr>
<td>102</td>
<td>129</td>
<td>01110111</td>
<td>f</td>
</tr>
<tr>
<td>103</td>
<td>130</td>
<td>01111000</td>
<td>g</td>
</tr>
<tr>
<td>104</td>
<td>131</td>
<td>01111001</td>
<td>h</td>
</tr>
<tr>
<td>105</td>
<td>132</td>
<td>01111010</td>
<td>i</td>
</tr>
<tr>
<td>106</td>
<td>133</td>
<td>01111011</td>
<td>j</td>
</tr>
<tr>
<td>107</td>
<td>134</td>
<td>01111100</td>
<td>k</td>
</tr>
<tr>
<td>108</td>
<td>135</td>
<td>01111101</td>
<td>l</td>
</tr>
<tr>
<td>109</td>
<td>136</td>
<td>01111110</td>
<td>m</td>
</tr>
<tr>
<td>110</td>
<td>137</td>
<td>01111111</td>
<td>n</td>
</tr>
<tr>
<td>111</td>
<td>138</td>
<td>01000000</td>
<td>o</td>
</tr>
<tr>
<td>112</td>
<td>139</td>
<td>01000001</td>
<td>p</td>
</tr>
</tbody>
</table>
**Word** |ˈwɜrd|, 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></td>
<td>32-bit</td>
</tr>
<tr>
<td>byte</td>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>short</td>
<td>short 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></td>
<td>long int</td>
<td>4</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></td>
<td>long double</td>
<td>8</td>
</tr>
</tbody>
</table>

Depends on word size!
Bitwise Operations

Shifty bits and counting cards
**bitwise operators**

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

\[
\begin{align*}
\text{AND} & \quad \text{OR} \quad | \\
\text{XOR} & \quad \text{NOT} \quad ~
\end{align*}
\]

\[
\begin{align*}
01101001 & \quad 01101001 & \quad 01101001 \\
\& 01010101 & \quad | 01010101 & \quad ^{01010101} & \quad ~01010101
\end{align*}
\]

Laws of Boolean algebra apply bitwise.

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

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

\[ a_i = 1 \equiv i \in A \]

01101001 \quad \{0, 3, 5, 6\}

76543210

01010101 \quad \{0, 2, 4, 6\}

76543210

Bitwise Operations

&   01000001 \quad \{0, 6\}

|   01111101 \quad \{0, 2, 3, 4, 5, 6\}

^   00111100 \quad \{2, 3, 4, 5\}

~   10101010 \quad \{1, 3, 5, 7\}

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

  - 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

[low-order 6 bits of a byte]

[suit value]
#define SUIT_MASK 0x30

int sameSuit(char card1, char card2) {
    return !((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 \( \nu \), turns all *but* the bits of interest in \( \nu \) to 0

\[
\text{#define VALUE\_MASK}
\]

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

\[ x \]

**x \ll 2**

\[
10011001 \\
\downarrow \downarrow \downarrow \downarrow \downarrow \\
01100100
\]

*logical shift left 2*

lose bits on left

*fill with zeroes on right*

**fill with zeroes on left**

**logical shift right 2**

**fill with zeroes on left**

**arithmetic shift right 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\textsuperscript{nd} most significant byte from a 32-bit integer.

given \quad x = \begin{array}{c}
\textcolor{red}{01100001} \quad \textcolor{red}{01100010} \quad \textcolor{red}{01100011} \quad \textcolor{red}{01100100}
\end{array}

should return: \quad \begin{array}{c}
\textcolor{red}{01100010}
\textcolor{red}{01100010}
\textcolor{red}{01100010}
\textcolor{red}{01100010}
\textcolor{red}{01100010}
\textcolor{red}{01100010}
\textcolor{red}{01100010}
\end{array}

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