1. (§2.6) Binary Number Representation

- Modern computers are "binary digital computers" meaning that they compute using binary numbers.
- What are binary numbers?
- Definition: a binary number is a number composed of only the digits 0 and 1 using a positional number representation.
- Examples:
  0, 111, 011011001

- Number systems are characterized by the number of digits used to represent values, called the base of the number system.
- Binary number are base-2
- While ordinary numbers using 10-digits ("0", "1", ..., "9") are base-10.
- Examples:
  0, 128, 99999

- Note the difference between a number and its representation in some base.
- Every number can be represented in any base
- Examples:
  \[0_{10} = 0_2\]
  \[129_{10} = 10000001_2\]
  \[99999_{10} = 11000011010011111_2\]

where we use the subscripts following the numbers to indicate in which base they are represented.

- Other popular bases include:
  - base-16 (called "hexadecimal" and pioneered by IBM)
    \[\text{digits} = 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\]
  - base-8 (called "octal" and pioneered by Digital Equipment Corp - nay Oracle)
    \[\text{digits} = 0,1,2,3,4,5,6,7\]
• Note that these bases are really binary "under the hood"
• Definition: a **position**al number representation represents arbitrarily large numbers using a fixed alphabet of digits organized such that digits (read right-to-left) represent successively higher orders of magnitude (of the base).
• Example: base-10
  \[ 129 = 1 \times 10^2 + 2 \times 10^1 + 9 \times 10^0 \]
  \[ = 100 + 20 + 9 \]
  \[ = 129 \]
• Example: base-2
  \[ 10000001 = 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \]
  \[ = 128 + 0 + 0 + 0 + 0 + 0 + 0 + 1 \]
  \[ = 129 \]
• Why do computers use binary arithmetic?
• Early computers used decimal arithmetic instead
• More natural for people (why?)
• How many digits do you have?
• Decimal computers used a number representation called "binary coded decimal" (BCD)
• BCD representation (using 4 binary digits)
  
<table>
<thead>
<tr>
<th>Value</th>
<th>BCD Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>Error</td>
<td>1010</td>
</tr>
<tr>
<td>Error</td>
<td>1011</td>
</tr>
<tr>
<td>Error</td>
<td>1100</td>
</tr>
<tr>
<td>Error</td>
<td>1101</td>
</tr>
<tr>
<td>Error</td>
<td>1110</td>
</tr>
<tr>
<td>Error</td>
<td>1111</td>
</tr>
</tbody>
</table>

• So the BCD representation wasted a lot of memory for illegal values
• Modern computers ALL use binary number representation
• Conversion from decimal to binary and back to decimal implemented as I/O functions
• See today's laboratory assignment
• BCD still popular in business/financial software
• Bad ideas never die!
• See: http://en.wikipedia.org/wiki/Binary-coded_decimal

2. Bits and Bytes

• Each piece of binary data is called a bit (smallest possible piece)
• In modern machines, bits are grouped into 8-bit pieces called bytes (for convenience)
• Computer arithmetic performed in 16-bit, 32-bit or now 64-bit chunks called words.
• First personal computer (Altair homebrew kit) used an 8-bit Intel 8080 chip.
• Memory sizes also expressed in bytes
• All sizes as powers of 2
• Examples:
  ‣ kilobyte = \(2^{10}\) bytes = 1024
  ‣ megabyte = \(2^{20}\) = 1048576
  ‣ gigabyte = \(2^{30}\) = 1073741824

• Why represent information in computers using binary data? Why not natural base-10?
  • (1) Because binary data is robust in memory
    ‣ magnetic polarity in hard disks
    ‣ electrical charge in flash memory
    ‣ electrical voltage in RAM memory
    ‣ electrical voltage in CPU chips
    ‣ pulses of light in optical fibers
  • (2) Boolean logic is simple for implementation in hardware
    ‣ arithmetic for boolean number is much simpler than base-10 arithmetic
    ‣ can be implemented using simple boolean logic gates
    ‣ examples: AND, OR, XOR, NOT, ...
    ‣ logic gates are easy to make on integrated circuit chips (and cheap!)
    ‣ we shall see how to perform binary arithmetic later . . .

3. Binary Arithmetic

• Lets practice with binary addition
• Consider the following example:

\[
\begin{array}{c}
1010 \\
+0100 \\
\hline
1110
\end{array}
\]
• Now an example with carries
  1101
+0101
_______
10010
• Exactly the same as decimal arithmetic (grade 3?)
• How does the computer CPU chip implement binary addition?

Using boolean logic circuits (note the transition from arithmetic to logic)
• Here is the logic circuit for a 1-bit adder
• see: http://www.circuitstoday.com/half-adder-and-full-adder

• Logic:
  Carry = AND(A, B)
  Sum = ExclusiveOR(A, B)
• Example: show sum and carry logic for binary addition
• Above is actually called a "half-adder" because it has no input for the carry bit. Need to use two half-adders to make a "full-adder" for each bit in the binary number.
• Here is a full-adder circuit (for curiosity sake only):
• These details are not important for our purposes

**CPUs**

• We could "glue" 8 or 16 or more of these things (full adders) together to add larger numbers.

• This is how the CPU implements arithmetic
• *Observation*: building a modern computer CPU using boolean logic is straightforward

**Simulation**

• Lets simulate the logic of the full-adder in software.
• Full 8bit adder coded in Python

```python
# full adder for 8bit binary arithmetic
# arguments are all 8-bit lists with high-order bit leftmost

def adder8bit(x, y, sum):
    carry = 0
    for i in range(7,-1,-1):
        sum[i] = (x[i] + y[i] + carry) % 2      # XOR gate
        carry =  (x[i] + y[i] + carry) / 2      # AND gate
    return carry                                # overflow
```
• Simulates XOR and NAND gates using modulus arithmetic and integer division
• Examples: show addition of lists of binary numbers as registers
  
x = [00000011]  # 3 (base10)
y = [00000010]  # 2 (base10)
z = [00000000]  # 0 (base10)
adder8bit(x, y, z)

4. Signed vs Unsigned Numbers
• Above we considered unsigned arithmetic only (all positive)
• Java, C, C++ all provide operations for unsigned arithmetic
• How can we represent positive AND negative numbers?
• Called signed arithmetic
• How can we do subtraction, multiplication and division?
• Similar boolean logic circuits do it all!
• Handling negative numbers is particularly interesting
• Two approaches:
  • (1) attach a sign bit to each number to indicate whether positive or negative
    ‣ Example
      +0100
      -0010
      ______
      +0010
    ‣ Sign bit is just the left-most (high-order) bit in the binary number
    ‣ But this approach is awkward
    ‣ Logic is complicated to implement
  • (2) use a clever scheme called 2's-complement arithmetic
• note that for n-bits there are $2^n$ possible patterns (permutations)
• for 8-bits there are $2^8 = 256$ possible numbers to represent
• in unsigned arithmetic the numbers are 0, 1, ..., 255
• BUT we could allocate half for positive and half for negative as follows:
• possibles values are:
  
  \[-2^{n-1}, \ldots, -1, 0, +1, \ldots, +2^{n-1}-1\]

• for 8bits this scale is:
  
  \[-128, \ldots, -1, 0, +1, \ldots, +127\]
• note that zero is considered positive
• But those hardware engineers are even more clever!
• Negative values above are encoded different than positive values (called 2's complement)
• Negative values are coded as follows:
  ‣ all the bits of the number are negated (flipped) from their positive version
  ‣ 1 is added to the result
• Example:
  
  \[
  \begin{align*}
  0101 & = +5 \\
  1010 & = \text{all bits flipped} \\
  1011 & = -5 \text{ by adding one}
  \end{align*}
  \]

• Advantages
  ‣ testing for negative value is easy (high order bit = 1)
  ‣ for positive numbers, the signed and unsigned versions are the same
  ‣ only one representation for zero
  ‣ addition and subtraction work the same (without testing the sign bit)
• Example: signed addition
  
  \[
  \begin{align*}
  1011 & = -5 \\
  0100 & = +4 \\
  \_ \_ \_ \_ & \_ \_ \_ \_ \\
  1111 & = -1 \\
  \end{align*}
  \]
  
which can be seen by converting back to unsigned

  \[
  \begin{align*}
  1111 & \quad \# \text{ subtract 1} \\
  1110 & \quad \# \text{ now flip bits} \\
  0001 & \quad = 1
  \end{align*}
  \]

5. Conversions to/from Binary
• Already seen how to convert binary to a decimal above
• But as an algorithm for any base its similar
  
say given a binary number \(x\)
  
  \[
  \begin{align*}
  \text{let sum} & = 0 \\
  \text{for the binary digit d scanning x from left to right:} \\
  \quad \text{sum} & = \text{sum} \times 2 + d \\
  \text{return sum}
  \end{align*}
  \]
• Example: convert 0101 to decimal
  \[
  \text{sum} = 0 \\
  \text{sum} = \text{sum} \times 2 + 0 = 0 \\
  \text{sum} = \text{sum} \times 2 + 1 = 1 \\
  \text{sum} = \text{sum} \times 2 + 0 = 2 \\
  \text{sum} = \text{sum} \times 2 + 1 = 5 \\
  \text{sum} = 5
  \]

• Converting from a decimal value to binary
• Need to repeatedly find the lowest order bit and shift the remainder
• Similar to the 8-bit adder above
• Algorithm:

  given a decimal number \(x\)
  repeat:
    \[
    \begin{align*}
    \text{next digit} &= \text{x} \% 2 \quad \# \text{find low order bit} \\
    \text{x} &= \text{x} / 2 \quad \# \text{shift number to the right}
    \end{align*}
    \]
  until \(x = 0\)

• Example: convert decimal 3 to binary
  \[
  x = 3 \\
  \text{next digit} = 3 \% 2 = 1 \quad \# \text{next binary digit} = 1 \\
  x = x / 2 = 3 / 2 = 1 \\
  \text{next digit} = 1 \% 2 = 1 \quad \# \text{next binary digit} = 1 \\
  x = x / 2 = 1/2 = 0
  \]
  halt

6. Floating-Point Numbers

• Floating point numbers are a different "kettle of fish"
• What are they for?
• Compare: counting versus distance
• Counting uses integers
• Distance measure is a real value (arbitrarily small or large)
• Approximated in computer arithmetic using floating point representation
• Basic Idea: Use scientific notation to extend dynamic range
• Examples in decimal:
  \[
  \begin{align*}
  0.67788 \times 10^{28} & \quad \# \text{large number} \\
  0.1 \times 10^{-200} & \quad \# \text{very small number}
  \end{align*}
  \]
• Scientific notation represents real numbers as:
  \((<\text{exp}>, <\text{fraction}>)\)
  where \(<\text{exp}>\) is a signed integer
  and \(<\text{fraction}>\) is a signed integer fraction with the radix
  point at the far left

7. **Characters and Strings**

• Characters are stored in memory as binary patterns.
• Each character has a unique (assigned) bit pattern.
• Called a **character code**
• Patterns are organized to facilitate sorting
• Various character codes are defined.
• Examples: EBCDIC, ASCII, Unicode

- **EBCDIC**
  ‣ defined by IBM for their IBM-360 series computers
  ‣ circa 1963
  ‣ very complicated extension of BCD numbering

- **ASCII** = American Standard Code for Information Interchange
  ‣ circa 1963
  ‣ popular standard for many years
  ‣ 7-bit code

• ASCII table:
<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>Nul</td>
<td>32</td>
<td>20</td>
<td>Space</td>
<td>64</td>
<td>40</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>Start of heading</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>Start of text</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>End of text</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>End of transit</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>Enquiry</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>Acknowledge</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>Audit trail</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>Backspace</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>Horizontal tab</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>Vertical tab</td>
<td>42</td>
<td>2A</td>
<td>*</td>
<td>74</td>
<td>4A</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>Form feed</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>Form feed</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>Carriage return</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>Shift out</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>Shift in</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>Data link escape</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>Device control 1</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>Device control 2</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>Device control 3</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>Device control 4</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>Hang up</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>Synchronous Bell</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>End of line</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>Cancel</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>End of medium</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>Substitution</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>Escape</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>]</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>File separator</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>Group separator</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>]</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>Record separator</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>Unit separator</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>_</td>
</tr>
</tbody>
</table>

**Unicode**

- Successor to ASCII
- contains as a subset
- used by Java exclusively
- 16-bit code
- supports many languages / character sets