• Previously:
  • Computer is machine that does what we tell it to do

• Next:
  • How do we tell computers what to do?
  • How do we represent data objects in binary?
  • How do we represent data locations in binary?
Representing High Level Things in Binary

- Computers represent **everything** in binary
- Instructions are specified in binary
- Instructions must be able to describe
  - Operation types (add, subtract, shift, etc.)
  - Data objects (integers, decimals, characters, etc.)
  - Memory locations
- Example:

```c
int x, y;       // Where are x and y? How to represent an int?
bool decision;  // How do we represent a bool? Where is it?
y = x + 7;       // How do we specify “add”? How to represent 7?
decision=(y>18); // Etc.
```
Representing Operation Types

• How do we tell computer to add? Shift? Read from memory? Etc.
• Arbitrarily! 😊
• Each Instruction Set Architecture (ISA) has its own binary encodings for each operation type
• E.g., in MIPS:
  • Integer add is: 00000 010000
  • Read from memory (load) is: 010011
  • Etc.
Representing Data Types

- Same as before: binary!
- Key Idea: the same 32 bits might mean one thing if interpreted as an integer but another thing if interpreted as a floating point number
Basic Data Types

**Bit** (bool): 0, 1

**Bit String**: sequence of bits of a particular length
- 4 bits is a nibble
- 8 bits is a byte
- 16 bits is a half-word (for MIPS32)
- 32 bits is a word (for MIPS32)
- 64 bits is a double-word (for MIPS32)
- 128 bits is a quad-word (for MIPS32)

**Integers** (int, long):
- “2's Complement” (32-bit or 64-bit representation)

**Floating Point** (float, double):
- Single Precision (32-bit representation)
- Double Precision (64-bit representation)
- Extended (Quad) Precision (128-bit representation)

**Character** (char):
- ASCII 7-bit code
Basic Binary

- Advice: memorize the following
  - $2^0 = 1$
  - $2^1 = 2$
  - $2^2 = 4$
  - $2^3 = 8$
  - $2^4 = 16$
  - $2^5 = 32$
  - $2^6 = 64$
  - $2^7 = 128$
  - $2^8 = 256$
  - $2^9 = 512$
  - $2^{10} = 1024$
### Decimal to binary using remainders

<table>
<thead>
<tr>
<th>?</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>457 ÷ 2 =</td>
<td>228</td>
<td>1</td>
</tr>
<tr>
<td>228 ÷ 2 =</td>
<td>114</td>
<td>0</td>
</tr>
<tr>
<td>114 ÷ 2 =</td>
<td>57</td>
<td>0</td>
</tr>
<tr>
<td>57 ÷ 2 =</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>28 ÷ 2 =</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>14 ÷ 2 =</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>7 ÷ 2 =</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3 ÷ 2 =</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 ÷ 2 =</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

111001001
### Decimal to binary using comparison

<table>
<thead>
<tr>
<th>Num</th>
<th>Compare $2^n$</th>
<th>$\geq$ ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>457</td>
<td>256</td>
<td>1</td>
</tr>
<tr>
<td>201</td>
<td>128</td>
<td>1</td>
</tr>
<tr>
<td>73</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

111001001
Binary to/from hexadecimal

- \(010110110010011_2 \rightarrow\)
- \(0101\ 1011\ 0010\ 0011_2 \rightarrow\)
- \(5\ B\ 2\ 3_{16}\)

\[
\begin{array}{cccc}
1 & F & 4 & B_{16} \rightarrow \\
0001\ 1111\ 0100\ 1011_2 \rightarrow \\
0001111101001011_2
\end{array}
\]
Issues for Binary Representation of Numbers

• There are many ways to represent numbers in binary
  • Binary representations are encodings \( \rightarrow \) many encodings possible
  • What are the issues that we must address?

• Issue #1: Complexity of arithmetic operations

• Issue #2: Negative numbers

• Issue #3: Maximum representable number

• Choose representation that makes these issues easy for machine, even if it’s not easy for humans (i.e., ECE/CS 250 students)
  • Why? Machine has to do all the work!
Sign Magnitude

- Use leftmost bit for + (0) or - (1):
- 6-bit example (1 sign bit + 5 magnitude bits):
  - +17 = 010001
  - -17 = 110001
- Pros:
  - Conceptually simple
  - Easy to convert
- Cons:
  - Harder to compute (add, subtract, etc) with
  - Positive and negative 0: 000000 and 100000

N O B O D Y  D O E S  T H I S
1’s Complement Representation for Integers

- Use largest positive binary numbers to represent negative numbers
  - To negate a number, invert (“not”) each bit:
    - 0 → 1
    - 1 → 0
- Cons:
  - Still two 0s (yuck)
  - Still hard to compute with

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-7</td>
</tr>
<tr>
<td>1001</td>
<td>-6</td>
</tr>
<tr>
<td>1010</td>
<td>-5</td>
</tr>
<tr>
<td>1011</td>
<td>-4</td>
</tr>
<tr>
<td>1100</td>
<td>-3</td>
</tr>
<tr>
<td>1101</td>
<td>-2</td>
</tr>
<tr>
<td>1110</td>
<td>-1</td>
</tr>
<tr>
<td>1111</td>
<td>-0</td>
</tr>
</tbody>
</table>

NOBODY DOES THIS
2’s Complement Integers

• Use large positives to represent negatives
  • \((-x) = 2^n - x\)
  • This is 1’s complement + 1
  • \((-x) = 2^n - 1 - x + 1\)
  • So, just invert bits and add 1

6-bit examples:
\[010110_2 = 22_{10}; 101010_2 = -22_{10}\]
\[1_{10} = 000001_2; -1_{10} = 111111_2\]
\[0_{10} = 000000_2; -0_{10} = 000000_2 \rightarrow \text{good!}\]
Pros and Cons of 2’s Complement

• Advantages:
  • Only one representation for 0 (unlike 1’s comp): \( 0 = 000000 \)
  • Addition algorithm is much easier than with sign and magnitude
    • Independent of sign bits

• Disadvantage:
  • One more negative number than positive
  • Example: 6-bit 2’s complement number
    \( 100000_2 = -32_{10} \); but \( 32_{10} \) could not be represented

All modern computers use 2’s complement for integers
Most computers today support 32-bit (int) or 64-bit integers
- Specify 64-bit using gcc C compiler with long long
- To extend precision, use sign bit extension
  - Integer precision is number of bits used to represent a number

Examples:
$14_{10} = 001110_2$ in 6-bit representation.
$14_{10} = 000000001110_2$ in 12-bit representation

$-14_{10} = 110010_2$ in 6-bit representation
$-14_{10} = 111111110010_2$ in 12-bit representation.
Binary Math : Addition

• Suppose we want to add two numbers:

\[
\begin{array}{c}
00011101 \\
+ 00101011 \\
\hline
00101011
\end{array}
\]

• How do we do this?
Binary Math : Addition

• Suppose we want to add two numbers:

\[
\begin{array}{c}
00011101 \\
+ 00101011 \\
\hline
00101011
\end{array}
\]

\[
\begin{array}{c}
695 \\
+ 232 \\
\hline
927
\end{array}
\]

• How do we do this?
  • Let’s revisit decimal addition
  • Think about the process as we do it
Binary Math : Addition

• Suppose we want to add two numbers:

\[
\begin{array}{c}
00011101 \\
+ 00101011 \\
\hline
00100100
\end{array}
\]

• First add one’s digit 5 + 2 = 7
Binary Math: Addition

- Suppose we want to add two numbers:
  \[ \begin{array}{c}
  \text{1} \\
  \text{00011101} \\
  + \text{00101011} \\
  \hline
  \text{00101011} \\
  \end{array} \]

- First add one’s digit 5+2 = 7
- Next add ten’s digit 9+3 = 12 (2 carry a 1)
Binary Math: Addition

• Suppose we want to add two numbers:

\[ \begin{array}{c}
00011101 \\
+ 00101011 \\
\hline
00101100
\end{array} \]

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>695</td>
<td>00011101</td>
</tr>
<tr>
<td>232</td>
<td>00101011</td>
</tr>
</tbody>
</table>

\[ 695 + 232 = 927 \]

• First add one’s digit 5+2 = 7
• Next add ten’s digit 9+3 = 12 (2 carry a 1)
• Last add hundred’s digit 1+6+2 = 9
Binary Math: Addition

• Suppose we want to add two numbers:

\[
\begin{array}{c}
00011101 \\
+ 00101011 \\
P00100111
\end{array}
\]

• Back to the binary:
• First add 1’s digit 1+1 = ...?
Binary Math: Addition

- Suppose we want to add two numbers:

\[
\begin{array}{c}
1 \\
00011101 \\
+ 00101011 \\
\hline
01010111
\end{array}
\]

- Back to the binary:
- First add 1’s digit 1+1 = 2 (0 carry a 1)
Binary Math : Addition

• Suppose we want to add two numbers:

\[
\begin{array}{c}
11 \\
00011101 \\
+ 00101011 \\
\hline
00 \\
\end{array}
\]

• Back to the binary:
  • First add 1’s digit 1+1 = 2 (0 carry a 1)
  • Then 2’s digit: 1+0+1 =2 (0 carry a 1)
  • You all finish it out....
Binary Math : Addition

• Suppose we want to add two numbers:

\[
\begin{align*}
111111 \\
00011101 & = 29 \\
+ 00101011 & = 43 \\
\hline
01001000 & = 72
\end{align*}
\]

• Can check our work in decimal
Binary Math: Addition

- What about this one:

\[
\begin{array}{c}
01011101 \\
+ 01101011 \\
\hline
01101011
\end{array}
\]
Binary Math: Addition

- What about this one:
  
  \[
  \begin{array}{c}
  11111111 \\
  01011101 \quad = \quad 93 \\
  + \quad 01101011 \quad = \quad 107 \\
  \hline
  11001000 \quad = \quad -56
  \end{array}
  \]

- But... that can’t be right?
  - What do you expect for the answer?
  - What is it in 8-bit signed 2’s complement?
Answer should be 200
  - Not representable in 8-bit signed representation
    - No right answer
This is called integer Overflow
Real problem in programs
Subtraction

• 2’s complement makes subtraction easy:
  • Remember: $A - B = A + (-B)$
  • And: $-B = \sim B + 1$
    \[ \uparrow \text{that means flip bits ("not")} \]
  • So we just flip the bits and start with carry-in (CI) = 1
  • Later: No new circuits to subtract (re-use adder hardware!)

\[
\begin{array}{c}
1 \\
0110101 \\
- 1010010 \\
\hline
- 0110101
\end{array} \quad \rightarrow \quad \begin{array}{c}
0110101 \\
+ 0101101 \\
\hline
0101101
\end{array}
\]
What About Non-integer Numbers?

- There are infinitely many real numbers between two integers
- Many important numbers are real
  - Speed of light $\sim= 3\times10^8$
  - $\pi = 3.1415...$
- Fixed number of bits limits range of integers
  - Can’t represent some important numbers
- Humans use Scientific Notation
  - $1.3\times10^4$
Option 1: Fixed point

- Use normal integers, but \((X \times 2^K)\) instead of \(X\)
  - Example: 32 bit int, but use \(X \times 65536\)
    - \(3.1415926 \times 65536 = 205887\)
    - \(0.5 \times 65536 = 32768\), etc..

- Pros:
  - Addition/subtraction just like integers ("free")

- Cons:
  - Mul/div require renormalizing (divide by 64K)
  - Range limited (no good rep for large + small)

- Can be good in specific situations
Can we do better?

- Think about scientific notation for a second:
- For example:
  \[6.02 \times 10^{23}\]
- Real number, but comprised of ints:
  - 6 generally only 1 digit here
  - 02 any number here
  - 10 always 10 (base we work in)
  - 23 can be positive or negative
- Can we do something like this in binary?
Option 2: Floating Point

- How about: 
  $ +/- X.YYYYYY \times 2^{+/-N}$

- Big numbers: large positive $N$
- Small numbers ($<1$): negative $N$
- Numbers near 0: small $N$

- This is “floating point” : most common way
IEEE single precision floating point

- Specific format called IEEE single precision: 
  
  \[ +/- \ 1.YYYYY \times 2^{(N-127)} \]

- “float” in Java, C, C++,...

- Assume first bit is always 1 (saves us a bit)
- 1 sign bit (+ = 0, 1 = -)
- 8 bit biased exponent (do N-127)
- Implicit 1 before *binary point*
- 23-bit *mantissa* (YYYYY)
Binary fractions

• 1.YYYY has a binary point
  • Like a decimal point but in binary
  • After a decimal point, you have
    • tenths
    • hundredths
    • thousandths
    • ...

• So after a binary point you have...
  • Halves
  • Quarters
  • Eighths
  • ...

Floating point example

- Binary fraction example:
  \[ 101.101 = 4 + 1 + \frac{1}{2} + \frac{1}{8} = 5.625 \]

- For floating point, needs normalization:
  \[ 1.01101 \times 2^2 \]

- Sign is +, which = 0

- Exponent = 127 + 2 = 129 = 1000 0001

- Mantissa = 1.011 0100 0000 0000 0000 0000
Example:
What floating-point number is:
0xC1580000?
What floating-point number is 0xC1580000?

\[ X = \begin{array}{cccccc}
1 & 1000 & 0010 & 101 & 1000 & 0000 0000 0000 0000 \\
\end{array} \]

\[ \begin{array}{cccccc}
s & E & F \\
1 & 128+2 & 3 \\
\end{array} \]

Sign = 1 which is negative

Exponent = \((128+2)-127 = 3\)

Mantissa = 1.1011

\[-1.1011 \times 2^3 = -1101.1 = -13.5\]
Trick question

- How do you represent 0.0?
  - Why is this a trick question?
    - 0.0 = 000000000
  - But need 1.XXXXX representation?

- Exponent of 0 is denormalized
  - Implicit 0. instead of 1. in mantissa
  - Allows 0000....0000 to be 0
  - Helps with very small numbers near 0

- Results in +/- 0 in FP (but they are “equal”)
Other Weird FP numbers

- Exponent = 1111 1111 also not standard
  - All 0 mantissa: +/- ∞
    
    \[
    1/0 = +\infty \\
    -1/0 = -\infty
    \]
  - Non zero mantissa: Not a Number (NaN)
    \[
    \sqrt{-42} = \text{NaN}
    \]
Floating Point Representation

• Double Precision Floating point:

  64-bit representation:
  • 1-bit sign
  • 11-bit (biased) exponent
  • 52-bit fraction (with implicit 1).

• “double” in Java, C, C++, ...

<table>
<thead>
<tr>
<th>S</th>
<th>Exp</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11-bit</td>
<td>52 - bit</td>
</tr>
</tbody>
</table>
What About Strings?

- Many important things stored as strings...
  - E.g., your name
- How should we store strings?
# Standardized ASCII (0-127)

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Html</th>
<th>Chr</th>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Html</th>
<th>Chr</th>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Html</th>
<th>Chr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>NUL</td>
<td>(null)</td>
<td>Space</td>
<td>32</td>
<td>20</td>
<td>040</td>
<td>&amp;#32;</td>
<td>&amp;#32;</td>
<td></td>
<td>64</td>
<td>40</td>
<td>100</td>
<td>&amp;#64;</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>SOH</td>
<td>(start of heading)</td>
<td>!</td>
<td>33</td>
<td>21</td>
<td>041</td>
<td>&amp;#33;</td>
<td>&amp;#33;</td>
<td></td>
<td>65</td>
<td>41</td>
<td>101</td>
<td>&amp;#65;</td>
</tr>
<tr>
<td>2</td>
<td>002</td>
<td>STX</td>
<td>(start of text)</td>
<td>&quot;</td>
<td>34</td>
<td>22</td>
<td>042</td>
<td>&amp;#34;</td>
<td>&amp;#34;</td>
<td></td>
<td>66</td>
<td>42</td>
<td>102</td>
<td>&amp;#66;</td>
</tr>
<tr>
<td>3</td>
<td>003</td>
<td>ETX</td>
<td>(end of text)</td>
<td>#</td>
<td>35</td>
<td>23</td>
<td>043</td>
<td>&amp;#35;</td>
<td>&amp;#35;</td>
<td></td>
<td>67</td>
<td>43</td>
<td>103</td>
<td>&amp;#67;</td>
</tr>
<tr>
<td>4</td>
<td>004</td>
<td>EOT</td>
<td>(end of transmission)</td>
<td>$</td>
<td>36</td>
<td>24</td>
<td>044</td>
<td>&amp;#36;</td>
<td>&amp;#36;</td>
<td></td>
<td>68</td>
<td>44</td>
<td>104</td>
<td>&amp;#68;</td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td>ENQ</td>
<td>(enquiry)</td>
<td>$</td>
<td>37</td>
<td>25</td>
<td>045</td>
<td>&amp;#37;</td>
<td>&amp;#37;</td>
<td></td>
<td>69</td>
<td>45</td>
<td>105</td>
<td>&amp;#69;</td>
</tr>
<tr>
<td>6</td>
<td>006</td>
<td>ACK</td>
<td>(acknowledge)</td>
<td>&amp;</td>
<td>38</td>
<td>26</td>
<td>046</td>
<td>&amp;#38;</td>
<td>&amp;#38;</td>
<td></td>
<td>70</td>
<td>46</td>
<td>106</td>
<td>&amp;#70;</td>
</tr>
<tr>
<td>7</td>
<td>007</td>
<td>BEL</td>
<td>(bell)</td>
<td>&amp; #39;</td>
<td>39</td>
<td>27</td>
<td>047</td>
<td>&amp;#39;</td>
<td>&amp;#39;</td>
<td></td>
<td>71</td>
<td>47</td>
<td>107</td>
<td>&amp;#71;</td>
</tr>
<tr>
<td>8</td>
<td>010</td>
<td>BS</td>
<td>(backspace)</td>
<td></td>
<td>40</td>
<td>28</td>
<td>050</td>
<td>&amp;#40;</td>
<td>&amp;#40;</td>
<td></td>
<td>72</td>
<td>48</td>
<td>110</td>
<td>&amp;#72;</td>
</tr>
<tr>
<td>9</td>
<td>011</td>
<td>TAB</td>
<td>(horizontal tab)</td>
<td></td>
<td>41</td>
<td>29</td>
<td>051</td>
<td>&amp;#41;</td>
<td>&amp;#41;</td>
<td></td>
<td>73</td>
<td>49</td>
<td>111</td>
<td>&amp;#73;</td>
</tr>
<tr>
<td>10</td>
<td>012</td>
<td>LF</td>
<td>(NL line feed, new line)</td>
<td>*</td>
<td>42</td>
<td>2A</td>
<td>052</td>
<td>&amp;#42;</td>
<td>&amp;#42;</td>
<td></td>
<td>74</td>
<td>4A</td>
<td>112</td>
<td>&amp;#74;</td>
</tr>
<tr>
<td>11</td>
<td>013</td>
<td>VT</td>
<td>(vertical tab)</td>
<td>+</td>
<td>43</td>
<td>2B</td>
<td>053</td>
<td>&amp;#43;</td>
<td>&amp;#43;</td>
<td></td>
<td>75</td>
<td>4B</td>
<td>113</td>
<td>&amp;#75;</td>
</tr>
<tr>
<td>12</td>
<td>014</td>
<td>FF</td>
<td>(NP form feed, new page)</td>
<td></td>
<td>44</td>
<td>2C</td>
<td>054</td>
<td>&amp;#44;</td>
<td>&amp;#44;</td>
<td></td>
<td>76</td>
<td>4C</td>
<td>114</td>
<td>&amp;#76;</td>
</tr>
<tr>
<td>13</td>
<td>015</td>
<td>CR</td>
<td>(carriage return)</td>
<td>-</td>
<td>45</td>
<td>2D</td>
<td>055</td>
<td>&amp;#45;</td>
<td>&amp;#45;</td>
<td></td>
<td>77</td>
<td>4D</td>
<td>115</td>
<td>&amp;#77;</td>
</tr>
<tr>
<td>14</td>
<td>016</td>
<td>SO</td>
<td>(shift out)</td>
<td>.</td>
<td>46</td>
<td>2E</td>
<td>056</td>
<td>&amp;#46;</td>
<td>&amp;#46;</td>
<td></td>
<td>78</td>
<td>4E</td>
<td>116</td>
<td>&amp;#78;</td>
</tr>
<tr>
<td>15</td>
<td>017</td>
<td>SI</td>
<td>(shift in)</td>
<td>/</td>
<td>47</td>
<td>2F</td>
<td>057</td>
<td>&amp;#47;</td>
<td>&amp;#47;</td>
<td></td>
<td>79</td>
<td>4F</td>
<td>117</td>
<td>&amp;#79;</td>
</tr>
<tr>
<td>16</td>
<td>020</td>
<td>DLE</td>
<td>(data link escape)</td>
<td>0</td>
<td>48</td>
<td>30</td>
<td>060</td>
<td>&amp;#48;</td>
<td>&amp;#48;</td>
<td></td>
<td>80</td>
<td>50</td>
<td>120</td>
<td>&amp;#80;</td>
</tr>
<tr>
<td>17</td>
<td>021</td>
<td>DC1</td>
<td>(device control 1)</td>
<td>1</td>
<td>49</td>
<td>31</td>
<td>061</td>
<td>&amp;#49;</td>
<td>&amp;#49;</td>
<td></td>
<td>81</td>
<td>51</td>
<td>121</td>
<td>&amp;#81;</td>
</tr>
<tr>
<td>18</td>
<td>022</td>
<td>DC2</td>
<td>(device control 2)</td>
<td>2</td>
<td>50</td>
<td>32</td>
<td>062</td>
<td>&amp;#50;</td>
<td>&amp;#50;</td>
<td></td>
<td>82</td>
<td>52</td>
<td>122</td>
<td>&amp;#82;</td>
</tr>
<tr>
<td>19</td>
<td>023</td>
<td>DC3</td>
<td>(device control 3)</td>
<td>3</td>
<td>51</td>
<td>33</td>
<td>063</td>
<td>&amp;#51;</td>
<td>&amp;#51;</td>
<td></td>
<td>83</td>
<td>53</td>
<td>123</td>
<td>&amp;#83;</td>
</tr>
<tr>
<td>20</td>
<td>024</td>
<td>DC4</td>
<td>(device control 4)</td>
<td>4</td>
<td>52</td>
<td>34</td>
<td>064</td>
<td>&amp;#52;</td>
<td>&amp;#52;</td>
<td></td>
<td>84</td>
<td>54</td>
<td>124</td>
<td>&amp;#84;</td>
</tr>
<tr>
<td>21</td>
<td>025</td>
<td>NAK</td>
<td>(negative acknowledge)</td>
<td></td>
<td>53</td>
<td>35</td>
<td>065</td>
<td>&amp;#53;</td>
<td>&amp;#53;</td>
<td></td>
<td>85</td>
<td>55</td>
<td>125</td>
<td>&amp;#85;</td>
</tr>
<tr>
<td>22</td>
<td>026</td>
<td>SYN</td>
<td>(synchronous idle)</td>
<td></td>
<td>54</td>
<td>36</td>
<td>066</td>
<td>&amp;#54;</td>
<td>&amp;#54;</td>
<td></td>
<td>86</td>
<td>56</td>
<td>126</td>
<td>&amp;#86;</td>
</tr>
<tr>
<td>23</td>
<td>027</td>
<td>ETB</td>
<td>(end of trans. block)</td>
<td>7</td>
<td>55</td>
<td>37</td>
<td>067</td>
<td>&amp;#55;</td>
<td>&amp;#55;</td>
<td></td>
<td>87</td>
<td>57</td>
<td>127</td>
<td>&amp;#87;</td>
</tr>
<tr>
<td>24</td>
<td>030</td>
<td>CAN</td>
<td>(cancel)</td>
<td>8</td>
<td>56</td>
<td>38</td>
<td>070</td>
<td>&amp;#56;</td>
<td>&amp;#56;</td>
<td></td>
<td>88</td>
<td>58</td>
<td>130</td>
<td>&amp;#88;</td>
</tr>
<tr>
<td>25</td>
<td>031</td>
<td>EM</td>
<td>(end of medium)</td>
<td>9</td>
<td>57</td>
<td>39</td>
<td>071</td>
<td>&amp;#57;</td>
<td>&amp;#57;</td>
<td></td>
<td>89</td>
<td>59</td>
<td>131</td>
<td>&amp;#89;</td>
</tr>
<tr>
<td>26</td>
<td>032</td>
<td>SUB</td>
<td>(substitute)</td>
<td>:</td>
<td>58</td>
<td>3A</td>
<td>072</td>
<td>&amp;#58;</td>
<td>&amp;#58;</td>
<td></td>
<td>90</td>
<td>5A</td>
<td>132</td>
<td>&amp;#90;</td>
</tr>
<tr>
<td>27</td>
<td>033</td>
<td>ESC</td>
<td>(escape)</td>
<td>;</td>
<td>59</td>
<td>3B</td>
<td>073</td>
<td>&amp;#59;</td>
<td>&amp;#59;</td>
<td></td>
<td>91</td>
<td>5B</td>
<td>133</td>
<td>&amp;#91;</td>
</tr>
<tr>
<td>28</td>
<td>034</td>
<td>FS</td>
<td>(file separator)</td>
<td>&lt;</td>
<td>60</td>
<td>3C</td>
<td>074</td>
<td>&amp;#60;</td>
<td>&amp;#60;</td>
<td></td>
<td>92</td>
<td>5C</td>
<td>134</td>
<td>&amp;#92;</td>
</tr>
<tr>
<td>29</td>
<td>035</td>
<td>GS</td>
<td>(group separator)</td>
<td>=</td>
<td>61</td>
<td>3D</td>
<td>075</td>
<td>&amp;#61;</td>
<td>&amp;#61;</td>
<td></td>
<td>93</td>
<td>5D</td>
<td>135</td>
<td>&amp;#93;</td>
</tr>
<tr>
<td>30</td>
<td>036</td>
<td>RS</td>
<td>(record separator)</td>
<td>&gt;</td>
<td>62</td>
<td>3E</td>
<td>076</td>
<td>&amp;#62;</td>
<td>&amp;#62;</td>
<td></td>
<td>94</td>
<td>5E</td>
<td>136</td>
<td>&amp;#94;</td>
</tr>
<tr>
<td>31</td>
<td>037</td>
<td>US</td>
<td>(unit separator)</td>
<td>?</td>
<td>63</td>
<td>3F</td>
<td>077</td>
<td>&amp;#63;</td>
<td>&amp;#63;</td>
<td></td>
<td>95</td>
<td>5F</td>
<td>137</td>
<td>&amp;#95;</td>
</tr>
</tbody>
</table>

Source: [www.LookupTables.com](http://www.LookupTables.com)
## One Interpretation of 128-255

<table>
<thead>
<tr>
<th>Unicode</th>
<th>Character</th>
<th>Unicode</th>
<th>Character</th>
<th>Unicode</th>
<th>Character</th>
<th>Unicode</th>
<th>Character</th>
<th>Unicode</th>
<th>Character</th>
<th>Unicode</th>
<th>Character</th>
<th>Unicode</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>ç</td>
<td>144</td>
<td>É</td>
<td>161</td>
<td>i</td>
<td>177</td>
<td>193</td>
<td>209</td>
<td>225</td>
<td>241</td>
<td>±</td>
<td></td>
<td></td>
</tr>
<tr>
<td>129</td>
<td>ü</td>
<td>145</td>
<td>æ</td>
<td>162</td>
<td>ó</td>
<td>178</td>
<td>194</td>
<td>210</td>
<td>226</td>
<td>242</td>
<td>&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>é</td>
<td>146</td>
<td>Æ</td>
<td>163</td>
<td>ú</td>
<td>179</td>
<td></td>
<td></td>
<td>211</td>
<td>227</td>
<td>π</td>
<td></td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>à</td>
<td>147</td>
<td>ô</td>
<td>164</td>
<td>ñ</td>
<td>180</td>
<td>196</td>
<td>212</td>
<td>228</td>
<td>244</td>
<td>&lt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>132</td>
<td>å</td>
<td>148</td>
<td>ö</td>
<td>165</td>
<td>ñ</td>
<td>181</td>
<td>197</td>
<td>213</td>
<td>229</td>
<td>245</td>
<td>&lt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>à</td>
<td>149</td>
<td>ö</td>
<td>166</td>
<td></td>
<td>182</td>
<td>198</td>
<td>214</td>
<td>230</td>
<td>246</td>
<td>‡</td>
<td></td>
<td></td>
</tr>
<tr>
<td>134</td>
<td>å</td>
<td>150</td>
<td>ú</td>
<td>167</td>
<td></td>
<td>183</td>
<td>199</td>
<td>215</td>
<td>231</td>
<td>247</td>
<td>‡</td>
<td></td>
<td></td>
</tr>
<tr>
<td>135</td>
<td>ç</td>
<td>151</td>
<td>û</td>
<td>168</td>
<td></td>
<td>184</td>
<td>200</td>
<td>216</td>
<td>232</td>
<td>248</td>
<td>°</td>
<td></td>
<td></td>
</tr>
<tr>
<td>136</td>
<td>è</td>
<td>152</td>
<td>_</td>
<td>169</td>
<td></td>
<td>185</td>
<td>201</td>
<td>217</td>
<td>233</td>
<td>249</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>137</td>
<td>é</td>
<td>153</td>
<td>Ö</td>
<td>170</td>
<td></td>
<td>186</td>
<td>202</td>
<td>218</td>
<td>234</td>
<td>250</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>138</td>
<td>è</td>
<td>154</td>
<td>Ü</td>
<td>171</td>
<td>£</td>
<td>187</td>
<td>203</td>
<td>219</td>
<td>235</td>
<td>251</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>139</td>
<td>i</td>
<td>156</td>
<td>£</td>
<td>172</td>
<td>£</td>
<td>188</td>
<td>204</td>
<td>220</td>
<td>236</td>
<td>252</td>
<td>_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>ï</td>
<td>157</td>
<td>¥</td>
<td>173</td>
<td>i</td>
<td>189</td>
<td>205</td>
<td>221</td>
<td>237</td>
<td>253</td>
<td>²</td>
<td></td>
<td></td>
</tr>
<tr>
<td>141</td>
<td>ï</td>
<td>158</td>
<td>_</td>
<td>174</td>
<td>«</td>
<td>190</td>
<td>206</td>
<td>222</td>
<td>238</td>
<td>254</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>142</td>
<td>Ä</td>
<td>159</td>
<td>j</td>
<td>175</td>
<td>»</td>
<td>191</td>
<td>207</td>
<td>223</td>
<td>239</td>
<td>255</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>143</td>
<td>Å</td>
<td>160</td>
<td>á</td>
<td>176</td>
<td></td>
<td>192</td>
<td>208</td>
<td>224</td>
<td>240</td>
<td>255</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: www.LookupTables.com
(This allowed totally sweet ASCII art in the 90s)

Sources:
Outline

• Previously:
  • Computer is machine that does what we tell it to do

• Next:
  • How do we tell computers what to do?
  • How do we represent data objects in binary?
  • How do we represent data locations in binary?
Computer Memory

- Where do we put these numbers?
  - Registers [more on these later]
    - In the processor core
    - Compute directly on them
    - Few of them (~16 or 32 registers, each 32-bit or 64-bit)

- Memory [Our focus now]
  - External to processor core
  - Load/store values to/from registers
  - Very large (multiple GB)
Memory Organization

- Memory: billions of locations...how to get the right one?
  - Each memory location has an address
  - Processor asks to read or write specific address
    - Memory, please load address 0x123400
    - Memory, please write 0xFE into address 0x8765000
  - Kind of like a giant array
    - Array of what?
      - Bytes?
      - 32-bit ints?
      - 64-bit ints?
Memory Organization

- Most systems: byte (8-bit) addressed
  - Memory is “array of bytes”
    - Each address specifies 1 byte
  - Support to load/store 8, 16, 32, 64 bit quantities
    - Byte ordering varies from system to system

- Some systems “word addressed”
  - Memory is “array of words”
    - Smaller operations “faked” in processor
  - Not very common
Word of the Day: Endianess

Byte Order

- **Big Endian:** byte 0 is 8 most significant bits IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- **Little Endian:** byte 0 is 8 least significant bits Intel 80x86, DEC Vax, DEC Alpha
• Memory is array of bytes, but there are conventions as to what goes where in this array
• Text: instructions (the program to execute)
• Data: global variables
• Stack: local variables and other per-function state; starts at top & grows down
• Heap: dynamically allocated variables; grows up
• What if stack and heap overlap????

Typical Address Space

Stack

Heap

Data

Text

Reserved

$2^{n-1}$
int anumber = 3;

int factorial (int x) {
    if (x == 0) {
        return 1;
    }
    else {
        return x * factorial (x - 1);
    }
}

int main (void) {
    int z = factorial (anumber);
    printf("%d\n", z);
    return 0;
}
public class Example {
    public static void swap (int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    }
    public static void main (String[] args) {
        int a = 42;
        int b = 100;
        swap (a, b);
        System.out.println("a =" + a + " b = " + b);
    }
}

• What does this print? Why?
public class Example {
    public static void swap (int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    }
    public static void main (String[] args) {
        int a = 42;
        int b = 100;
        swap (a, b);
        System.out.println("a =" + a + " b = " + b);
    }
}

• What does this print? Why?
public class Example {
    public static void swap (int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    }
    public static void main (String[] args) {
        int a = 42;
        int b = 100;
        swap (a, b);
        System.out.println("a =" + a + " b = " + b);
    }
}

• What does this print? Why?
public class Example {
    public static void swap (int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    }
    public static void main (String[] args) {
        int a = 42;
        int b = 100;
        swap (a, b);
        System.out.println("a =" + a + " b = " + b);
    }
}

• What does this print? Why?
public class Example {
    public static void swap (int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    }
    public static void main (String[] args) {
        int a = 42;
        int b = 100;
        swap (a, b);
        System.out.println("a =" + a + " b = " + b);
    }
}

- What does this print? Why?
public class Example {
    public static void swap (int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    }
    public static void main (String[] args) {
        int a = 42;
        int b = 100;
        swap (a, b);
        System.out.println("a =" + a + " b = " + b);
    }
}

• What does this print? Why?
public class Example {

    public static void swap (int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    public static void main (String[] args) {
        int a = 42;
        int b = 100;
        swap (a, b);

        System.out.println("a =" + a + " b = " + b);
    }
}

• What does this print? Why?
public class Ex2 {
    int data;
    public Ex2 (int d) { data = d; }
    public static void swap (Ex2 x, Ex2 y) {
        int temp = x.data;
        x.data = y.data;
        y.data = temp;
    }
    public static void main (String[] args) {
        Example a = new Example (42);
        Example b = new Example (100);
        swap (a, b);
        System.out.println("a =" + a.data +
            " b = " + b.data);
    }
}

• What does this print? Why?
public class Ex2 {
    int data;
    public Ex2 (int d) { data = d; }
    public static void swap (Ex2 x, Ex2 y) {
        int temp = x.data;
        x.data = y.data;
        y.data = temp;
    }
    public static void main (String[] args) {
        Example a = new Example (42);
        Example b = new Example (100);
        swap (a, b);
        System.out.println("a =" + a.data + " b = " + b.data);
    }
}

What does this print? Why?
Let’s do some different Java...

```java
public class Ex2 {
    int data;
    public Ex2 (int d) { data = d; }
    public static void swap (Ex2 x, Ex2 y) {
        int temp = x.data;
        x.data = y.data;
        y.data = temp;
    }
    public static void main (String[] args) {
        Example a = new Example (42);
        Example b = new Example (100);
        swap (a, b);
        System.out.println("a = " + a.data + " b = " + b.data);
    }
}
```

- What does this print? Why?
public class Ex2 {
    int data;
    public Ex2 (int d) { data = d; }
    public static void swap (Ex2 x, Ex2 y) {
        int temp = x.data;
        x.data = y.data;
        y.data = temp;
    }
    public static void main (String[] args) {
        Example a = new Example (42);
        Example b = new Example (100);
        swap (a, b);
        System.out.println("a =" + a.data +
        " b = " + b.data);
    }
}

What does this print? Why?
Let’s do some different Java…

```java
public class Ex2 {
    int data;
    public Ex2 (int d) { data = d; }
    public static void swap (Ex2 x, Ex2 y) {
        int temp = x.data;
        x.data = y.data;
        y.data = temp;
    }
    public static void main (String[] args) {
        Example a = new Example (42);
        Example b = new Example (100);
        swap (a, b);
        System.out.println("a = " + a.data + " b = " + b.data);
    }
}
```

What does this print? Why?
public class Ex2 {
    int data;
    public Ex2 (int d) { data = d; }
    public static void swap (Ex2 x, Ex2 y) {
        int temp = x.data;
        x.data = y.data;
        y.data = temp;
    }
    public static void main (String[] args) {
        Example a = new Example (42);
        Example b = new Example (100);
        swap (a, b);
        System.out.println("a =" + a.data + " b = " + b.data);
    }
}

• What does this print? Why?
public class Ex2 {
    int data;
    public Ex2 (int d) { data = d; }
    public static void swap (Ex2 x, Ex2 y) {
        int temp = x.data;
        x.data = y.data;
        y.data = temp;
    }
    public static void main (String[] args) {
        Example a = new Example (42);
        Example b = new Example (100);
        swap (a, b);
        System.out.println("a = " + a.data + " b = " + b.data);
    }
}

• What does this print? Why?
public class Ex2 {
    int data;
    public Ex2 (int d) { data = d; }
    public static void swap (Ex2 x, Ex2 y) {
        int temp = x.data;
        x.data = y.data;
        y.data = temp;
    }
    public static void main (String[] args) {
        Example a = new Example (42);
        Example b = new Example (100);
        swap (a, b);
        System.out.println("a =" + a.data +
                " b = " + b.data);
    }
}

- What does this print? Why?
References and Pointers (review)

• Java has **references**:
  - Any variable of object type is a reference
  - Point at objects (which are all in the heap)
    • Under the hood: is the memory address of the object
  - Cannot explicitly manipulate them (*e.g.*, add 4)

• Some languages (C, C++, assembly) have explicit **pointers**:
  - Hold the memory address of something
  - Can explicitly compute on them
  - Can **de-reference** the pointer (*ptr) to get thing-pointed-to
  - Can take the **address-of** (&x) to get something’s address
  - Can do very **unsafe** things, shoot yourself in the foot
Pointers

• “address of” operator &
  • don’t confuse with bitwise AND operator (&&)

Given
  int x; int* p;  // p points to an int
  p = &x;

Then
  *p = 2; and x = 2; produce the same result
  Note: p is a pointer, *p is an int

• What happens for p = 2?;

On 32-bit machine, p is 32-bits

\[
\begin{array}{c}
\text{x} & 0x26cf0 \\
\text{...} & \\
\text{p} & 0x26d00 \quad 0x26cbf0 \\
\end{array}
\]
Back to Arrays

- **Java:**
  
  ```java
  int [] x = new int [nElems];
  ```

- **C:**
  
  ```c
  int data[42]; //if size is known constant
  int* data = (int*)malloc (nElem * sizeof(int));
  ```

  - **malloc** takes number of bytes
  - **sizeof** tells how many bytes something takes
• x is a pointer, what is x+33?
• A pointer, but where?
  • what does calculation depend on?
• Result of adding an int to a pointer depends on size of object pointed to
  • One reason why we tell compiler what type of pointer we have, even though all pointers are really the same thing (and same size)
More Pointer Arithmetic

• address one past the end of an array is ok for pointer comparison only

• what’s at *(begin+44)?

• what does begin++ mean?

• how are pointers compared using < and using == ?

• what is value of end - begin?

char* a = new char[44];
char* begin = a;
char* end = a + 44;

while (begin < end)
{
    *begin = ‘z’;
    begin++;
}
More Pointers & Arrays

```cpp
int* a = new int[100];
```

- `a` is a pointer
- `*a` is an int
- `a[0]` is an int (same as `*a`)
- `a[1]` is an int
- `a+1` is a pointer
- `a+32` is a pointer
- `*(a+1)` is an int (same as `a[1]`)
- `*(a+99)` is an int
- `*(a+100)` is trouble
#include <stdio.h>

main()
{
    int* a = (int*)malloc (100 * sizeof(int));
    int* p = a;
    int k;

    for (k = 0; k < 100; k++)
    {
        *p = k;
        p++;
    }

    printf("entry 3 = %d\n", a[3])
}
Memory Manager (Heap Manager)

- `malloc()` and `free()`
- Library routines that handle memory management for heap (allocation / deallocation)
- Java has garbage collection (reclaim memory of unreferenced objects)
- C must use `free`, else memory leak
Strings as Arrays (review)

- A string is an array of characters with '\0' at the end
- Each element is one byte, ASCII code
- '\0' is null (ASCII code 0)
`strlen()` again

- `strlen()` returns the number of characters in a string
  - same as number elements in char array?

```c
int strlen(char * s)
// pre: ‘\0’ terminated
// post: returns # chars
{
    int count=0;
    while (*s++)
        count++;
    return count;
}
```
Vector Class vs. Arrays

- Vector Class
  - insulates programmers
  - array bounds checking
  - automagically growing/shrinking when more items are added/deleted

- How are Vectors implemented?
  - Arrays, re-allocated as needed

- Arrays can be more efficient
Summary: From C to Binary

- Everything must be represented in binary!
- Computer memory is linear array of bytes
- Pointer is memory location that contains address of another memory location
- We’ll visit these topics again throughout semester