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?
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:
  ```
  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

- 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

What is a *word*?
The standard unit of manipulation for a particular system. E.g.:

- **MIPS32:** 32 bits
- Original Nintendo: 8 bit
- Super Nintendo: 16 bit
- Intel x86 (classic): 32 bit
- Nintendo 64: 64 bit
- Intel x86_64 (modern): 64 bit
Basic Binary

- Advice: memorize the following
  - $2^0 = 1$
  - $2^1 = 2$
  - $2^2 = 4$
  - $2^3 = 8$
  - $2^4 = 16$
  - $2^5 = 32$
  - $2^6 = 64$
  - $2^7 = 128$
  - $2^8 = 256$
  - $2^9 = 512$
  - $2^{10} = 1024$
Useful bit facts

- **If you have N bits, you can represent** \(2^N\) **things.**

- **The binary metric system:**
  - \(2^{10} = 1024\).
  - This is *basically* 1000, so we can have an alternative form of metric units based on base 2.
  - \(2^{10}\) bytes = 1024 bytes = 1kB.
    - Sometimes written as 1kib (pronounced “kibibyte” where the ‘bi’ means ‘binary’) (but nobody says “kibibyte” out loud because it sounds stupid)
  - \(2^{20}\) bytes = 1MB, \(2^{30}\) bytes = 1GB, \(2^{40}\) bytes = 1TB, etc.
  - **Easy rule to convert between exponent and binary metric number:**

\[
2^{XY} \text{ bytes} = 2^Y \text{ <X_suffix> B}
\]

- \(2^{13}\) bytes = \(2^3\) kB = 8 kB
- \(2^{39}\) bytes = \(2^9\) GB = 512 GB
- \(2^{05}\) bytes = \(2^5\) B = 32 B
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
# Hexadecimal

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

- 0101101100100011₂ -->
- 0101 1011 0010 0011₂ -->
- 5  B  2  3₁₆

1  F  4  B₁₆ -->

0001 1111 0100 1011₂ -->

0001111101001011₂
BitOps: Unary

- Bit-wise complement (~)
  - Flips every bit.

\[
\begin{align*}
\sim 0x0d & \quad // \ (binary \ 00001101) \\
== 0xf2 & \quad // \ (binary \ 11110010)
\end{align*}
\]

Not the same as Logical NOT(!) or sign change (−)

```c
char i, j1, j2, j3;
i = 0x0d; \quad // binary \ 00001101
j1 = \sim i; \quad // binary \ 11110010
j2 = -i; \quad // binary \ 11110011
j3 = !i; \quad // binary \ 00000000
```
BitOps: Two Operands

- Operate **bit-by-bit** on operands to produce a result operand of the same length
- And (**&**): result 1 if both inputs 1, 0 otherwise
- Or (**|**): result 1 if either input 1, 0 otherwise
- Xor (**^**): result 1 if one input 1, but not both, 0 otherwise
- Operands **must** be of type integer
Two Operands... (cont’d)

Examples

\[
\begin{array}{c}
0011 1000 \\
\& 1101 1110 \\
\hline
0001 1000
\end{array}
\]

\[
\begin{array}{c}
0011 1000 \\
| 1101 1110 \\
\hline
1111 1110
\end{array}
\]

\[
\begin{array}{c}
0011 1000 \\
\^ 1101 1110 \\
\hline
1110 0110
\end{array}
\]
Shift Operations

- $x << y$ is left (logical) shift of $x$ by $y$ positions
  - $x$ and $y$ must both be integers
  - $x$ should be unsigned or positive
  - $y$ leftmost bits of $x$ are discarded
  - zero fill $y$ bits on the right

$$01111001 << 3 \quad \text{--------------------} \quad 11001000$$

these 3 bits are zero filled

these 3 bits are discarded
ShiftOps... (cont’d)

- \( x \gg y \) is right (logical) shift of \( x \) by \( y \) positions
  - \( y \) rightmost bits of \( x \) are discarded
  - zero fill \( y \) bits on the left

\[
\begin{array}{c}
01111001 \\
\downarrow \downarrow \downarrow \\
00001111
\end{array}
\]

these 3 bits are discarded

these 3 bits are zero filled
Bitwise Recipes

• Set a certain bit to 1?
  • Make a MASK with a one at every position you want to set:
    \[ m = 0x02; \quad // \quad 00000010_2 \]
  • OR the mask with the input:
    \[ v = 0x41; \quad // \quad 01000001_2 \]
    \[ v |= m; \quad // \quad 01000011_2 \]

• Clear a certain bit to 0?
  • Make a MASK with a one at every position you want to clear:
    \[ m = 0xFD; \quad // \quad 11111101_2 \] (could also write \(~0x02\) )
  • AND the mask with the input:
    \[ v = 0x27; \quad // \quad 00100111_2 \]
    \[ v &= m; \quad // \quad 00100101_2 \]

• Get a substring of bits (such as bits 2 through 5)?
  Note: bits are numbered right-to-left starting with zero.
  • Shift the bits you want all the way to the right then AND them with an appropriate mask:
    \[ v = 0x67; \quad // \quad 01 \underline{10011}1_2 \]
    \[ v >>= 2; \quad // \quad 0001 \underline{1001}2 \]
    \[ v &= 0x0F; \quad // \quad 0000 \underline{1001}2 \]
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
00100110 \\
\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:

\[
\begin{align*}
00011101 & \quad 695 \\
+ \quad 00101011 & \quad + \quad 232 \\
\hline
00100110 & \quad 7
\end{align*}
\]

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

• Suppose we want to add two numbers:

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

\[
\begin{array}{c}
695 \\
+ 232 \\
\hline
27
\end{array}
\]

• 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:

\[
\begin{array}{c}
00011101 \\
+ 00101011 \\
\hline
001001101
\end{array}
\]

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

• 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
• Suppose we want to add two numbers:

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

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

• Suppose we want to add two numbers:

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

• 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:

\[
\begin{array}{c}
111111 \\
00011101 = 29 \\
+ 00101011 = 43 \\
\hline
01001000 = 72
\end{array}
\]

• Can check our work in decimal
Issues for Binary Representation of Numbers

- **How to represent negative numbers?**

- There are many ways to represent numbers in binary
  - Binary representations are encodings → 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

NOBODY DOES THIS
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

```plaintext
  +---------+---+  +---------+---+
  0000 0   0001 1   0010 2   0011 3
  0100 4   0101 5   0110 6   0111 7
  1000 -7  1001 -6  1010 -5  1011 -4
  1100 -3  1101 -2  1110 -1  1111 -0
```

NOBODY DOES THIS EITHER
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:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>010110</td>
<td>22 (_{10})</td>
</tr>
<tr>
<td>101010</td>
<td>-22 (_{10})</td>
</tr>
<tr>
<td>000001</td>
<td>1 (_{10})</td>
</tr>
<tr>
<td>111111</td>
<td>-1 (_{10})</td>
</tr>
<tr>
<td>000000</td>
<td>0 (_{10})</td>
</tr>
<tr>
<td>111111</td>
<td>-1 (_{10})</td>
</tr>
</tbody>
</table>

EVERYBODY DOES THIS
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
2’s Complement Precision Extension

- 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

\[ 14_{10} = 001110_2 \text{ in 6-bit representation.} \]
\[ 14_{10} = 000000001110_2 \text{ in 12-bit representation} \]

\[ -14_{10} = 110010_2 \text{ in 6-bit representation} \]
\[ -14_{10} = 11111110010_2 \text{ in 12-bit representation.} \]
Binary Math : Addition

- Let’s look at another binary addition:

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

• What about this one:

\[
\begin{align*}
1111111 & \\
01011101 & = 93 \\
+ 01101011 & = 107 \\
\hline
11001000 & = -56
\end{align*}
\]

• But... that can’t be right?
  • What do you expect for the answer?
  • What is it in 8-bit signed 2’s complement?
Integer Overflow

- 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: \( -B = \sim B + 1 \)
    - \( \uparrow \) 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!)

\[
\begin{array}{c}
0110101 \\
\downarrow \\
1010010 \\
\uparrow \\
0101101
\end{array}
\]

\[
\begin{array}{c}
0110101 \\
\rightarrow \\
0110101 \\
\downarrow \\
0101101
\end{array}
\]
What About Non-integer Numbers?

- There are infinitely many real numbers between two integers.
- Many important numbers are real:
  - Speed of light $\sim= 3 \times 10^8$
  - $\pi = 3.1415...$
- Fixed number of bits limits range of integers:
  - Can’t represent some important numbers.
- Humans use Scientific Notation:
  - $1.3 \times 10^4$
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:
  $\pm 1.YYYYY \times 2^{(N-127)}$

• “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

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 23 & 22 & \cdots & 0 \\
1000 & 0001 & 011 & 0100 & 0000 & 0000 & 0000 & 0000 \\
\end{array}
\]
Example:
What floating-point number is: 
0xC1580000?
What floating-point number is 0xC1580000?

1100 0001 0101 1000 0000 0000 0000 0000

\[ X = \begin{array}{cccccc}
31 & 30 & 23 & 22 & \\
1 & 1000 & 0010 & 101 & 1000 & 0000 0000 0000 0000
\end{array} \]

\[ s \quad E \quad F \]

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.XXXX 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: +/- $\infty$
    - $1/0 = +\infty$
    - $-1/0 = -\infty$
  - 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>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td><code>null</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>001</td>
<td><code>SOH</code> (start of heading)</td>
<td><code>!</code></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>002</td>
<td><code>STX</code> (start of text)</td>
<td><code>&quot;</code></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>003</td>
<td><code>ETX</code> (end of text)</td>
<td><code>#</code></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>004</td>
<td><code>EOT</code> (end of transmission)</td>
<td>``</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>005</td>
<td><code>ENQ</code> (enquiry)</td>
<td><code>$</code></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>006</td>
<td><code>ACK</code> (acknowledge)</td>
<td><code>&amp;</code></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>007</td>
<td><code>BEL</code> (bell)</td>
<td><code>'</code></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>010</td>
<td><code>BS</code> (backspace)</td>
<td><code>(space)</code></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>011</td>
<td><code>TAB</code> (horizontal tab)</td>
<td><code>(tab)</code></td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>012</td>
<td><code>LF</code> (NL line feed, new line)</td>
<td><code>*</code></td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>013</td>
<td><code>VT</code> (vertical tab)</td>
<td><code>+</code></td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>014</td>
<td><code>FF</code> (NP form feed, new page)</td>
<td><code>,</code></td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>015</td>
<td><code>CR</code> (carriage return)</td>
<td><code>-</code></td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>016</td>
<td><code>SO</code> (shift out)</td>
<td><code>.</code></td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>017</td>
<td><code>SI</code> (shift in)</td>
<td><code>/</code></td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>020</td>
<td><code>DLE</code> (data link escape)</td>
<td><code>:</code></td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>021</td>
<td><code>DC1</code> (device control 1)</td>
<td><code>&lt;</code></td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>022</td>
<td><code>DC2</code> (device control 2)</td>
<td><code>=</code></td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>023</td>
<td><code>DC3</code> (device control 3)</td>
<td><code>&gt;</code></td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>024</td>
<td><code>DC4</code> (device control 4)</td>
<td><code>?</code></td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>025</td>
<td><code>NAK</code> (negative acknowledge)</td>
<td><code>@</code></td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>026</td>
<td><code>SYN</code> (synchronous idle)</td>
<td><code>^</code></td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>027</td>
<td><code>ETB</code> (end of trans. block)</td>
<td><code>_</code></td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>030</td>
<td><code>CAN</code> (cancel)</td>
<td><code>DEL</code></td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>031</td>
<td><code>EM</code> (end of medium)</td>
<td><code>DEL</code></td>
</tr>
</tbody>
</table>
One Interpretation of 128-255

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</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></td>
<td>193</td>
<td></td>
<td>209</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></td>
<td>194</td>
<td></td>
<td>210</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>195</td>
<td></td>
<td>211</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>à</td>
<td>147</td>
<td>ò</td>
<td>164</td>
<td>ŋ</td>
<td>180</td>
<td></td>
<td>196</td>
<td></td>
<td>212</td>
<td></td>
</tr>
<tr>
<td>132</td>
<td>ä</td>
<td>148</td>
<td>ö</td>
<td>165</td>
<td>Ñ</td>
<td>181</td>
<td></td>
<td>197</td>
<td></td>
<td>213</td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>à</td>
<td>149</td>
<td>ò</td>
<td>166</td>
<td></td>
<td>182</td>
<td></td>
<td>198</td>
<td></td>
<td>214</td>
<td></td>
</tr>
<tr>
<td>134</td>
<td>å</td>
<td>150</td>
<td>ü</td>
<td>167</td>
<td></td>
<td>183</td>
<td></td>
<td>199</td>
<td></td>
<td>215</td>
<td></td>
</tr>
<tr>
<td>135</td>
<td>ş</td>
<td>151</td>
<td>ü</td>
<td>168</td>
<td>ç</td>
<td>184</td>
<td></td>
<td>200</td>
<td></td>
<td>216</td>
<td></td>
</tr>
<tr>
<td>136</td>
<td>ö</td>
<td>152</td>
<td>–</td>
<td>169</td>
<td>–</td>
<td>185</td>
<td></td>
<td>201</td>
<td></td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>137</td>
<td>î</td>
<td>153</td>
<td>Ö</td>
<td>170</td>
<td>–</td>
<td>186</td>
<td></td>
<td>202</td>
<td></td>
<td>218</td>
<td></td>
</tr>
<tr>
<td>138</td>
<td>è</td>
<td>154</td>
<td>Ü</td>
<td>171</td>
<td>½</td>
<td>187</td>
<td></td>
<td>203</td>
<td></td>
<td>219</td>
<td></td>
</tr>
<tr>
<td>139</td>
<td>i</td>
<td>156</td>
<td>£</td>
<td>172</td>
<td>¼</td>
<td>188</td>
<td></td>
<td>204</td>
<td></td>
<td>220</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>ñ</td>
<td>157</td>
<td>¥</td>
<td>173</td>
<td>i</td>
<td>189</td>
<td></td>
<td>205</td>
<td></td>
<td>221</td>
<td></td>
</tr>
<tr>
<td>141</td>
<td>i</td>
<td>158</td>
<td>–</td>
<td>174</td>
<td>«</td>
<td>190</td>
<td></td>
<td>206</td>
<td></td>
<td>222</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></td>
<td>207</td>
<td></td>
<td>223</td>
<td></td>
</tr>
<tr>
<td>143</td>
<td>Á</td>
<td>160</td>
<td>á</td>
<td>176</td>
<td></td>
<td>192</td>
<td></td>
<td>208</td>
<td></td>
<td>224</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 Layout

- Memory is an 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????
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;
}
Summary: From C to Binary

- Everything must be represented in binary!
- Pointer is memory location that contains address of another memory location
- Computer memory is linear array of bytes
  - **Integers:**
    - `unsigned` \(0..2^n-1\) vs `signed` \(-2^{n-1} .. 2^{n-1}-1\) (“2’s complement”)
    - `char` (8-bit), `short` (16-bit), `int/long` (32-bit), `long long` (64-bit)
  - **Floats:** IEEE representation,
    - `float` (32-bit: 1 sign, 8 exponent, 23 mantissa)
    - `double` (64-bit: 1 sign, 11 exponent, 52 mantissa)
  - **Strings:** char array, ASCII representation
- Memory layout
  - **Stack** for local, **static** for globals, **heap** for malloc’d stuff (must free!)
The following slides re-state a lot of what we’ve covered but in a different way. We’ll likely skip it for time, but you can use the slides as an additional reference.
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?

Stack

<table>
<thead>
<tr>
<th>main</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>42</td>
</tr>
<tr>
<td>b</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>swap</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>42</td>
</tr>
<tr>
<td>y</td>
<td>100</td>
</tr>
<tr>
<td>temp</td>
<td>???</td>
</tr>
<tr>
<td>RA</td>
<td>c0</td>
</tr>
</tbody>
</table>
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?
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
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);
    }
}
```

**Stack**
- main
  - a
  - b

**Heap**
- Ex2
  - data 42
  - data 100
  - temp 42
  - RA c0

• 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
Points

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

Given

```c
int x; int* p;  // p points to an int
p = &x;
```

Then

```c
*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

```
x 0x26cf0

... ...

p 0x26d00 0x26cbf0
```
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
Arrays, Pointers, and Address Calculation

- 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)

```c
int* a = malloc(100*sizeof(int));

*(a+33) is the same as *(a+33)
if a is 0x00a0, then a+1 is 0x00a4, a+2 is 0x00a8
(decimal 160, 164, 168)
```

```c
doUBLE* d = malloc(200*sizeof(double));

*(d+33) is the same as d[33]
if d is 0x00b0, then d+1 is 0x00b8, d+2 is 0x00c0
(decimal 176, 184, 192)
```
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?

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

while (begin < end)
{
    *begin = 'z';
    begin++;
}
```
```c
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
```
```c
#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