Contact Information

Christopher Conway
conway@cs.columbia.edu
(212) 939-7069
521 Computer Science Building
Office Hours: 3:30 - 5:00 Tuesday and Thursday

TA:
Bogdan Caprita
bc2008@columbia.edu
TA Room (1st Floor Mudd)
Office Hours: 3:00 - 4:30 Monday and Wednesday

http://www.columbia.edu/~cs1007

Textbooks

Required:
Available at Papyrus Books on Broadway at 114th.

Recommended:

Grading

- Homework 40%
- Final 30%
- Midterm 20%
- Quiz 10%

- There will be five homeworks. The first homework is due next Tuesday.
- Homeworks must be submitted by 6:00 AM on the due date.

Test

- Quiz: June 3 (next Tuesday), 15 minutes (beginning of class).
- Midterm: June 13, 90 minutes (first half of class).
- Final: July 3, 190 minutes (whole class), covers the entire semester.

Software

Everything you need is on CUNIX. There are fancy Java IDEs available, but we won’t do anything that requires more than a text editor and the Java SDK.

If you want to work on your own computer:

- Use Java version 1.3.1
- Test all of your work on CUNIX.

What is Computer Science?

Computer science is the study of computation (i.e., processing and manipulating data). It is an intersection of mathematics, science and engineering.

Primary concerns:
1. Is a problem computable? (Algorithms)
2. Is finding a solution to a problem feasible? (Complexity Theory)
3. Is solving a problem practical? (Hardware/Software)

What is a Computer?

An algorithm is a set of rules for transforming an input into an output.

A computer is a device that can execute algorithms. It takes:

a) A set of rules (a program)

b) Input data (e.g., keyboard input, mouse clicks, a database file)

And produces:

- Output data (e.g., display output, a web page, a result set)

No, Really, What is a Computer?

By “computer”, we typically mean a binary stored program digital computer. (Von Neumann model)

- “binary” - the data is stored as ones and zeroes.
- “stored program” - the computer has memory, and both the data and the program instructions are stored in that memory.
- “digital” - operates on discrete quantities (i.e., ones and zeroes). Not continuous.
- “computer” - it can execute algorithms.
**Binary**

Binary is a base-2 number system. In general, a base-$n$ number system encodes numbers as:

\[(x_1x_{i-1} \ldots x_1x_0)_n = x_0n^0 + x_1n^1 + \ldots + x_{i-1}n^{i-1} + x_in^i\]

Thus:

\[101011_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 43\]

Computers like binary because $0/1 \leftrightarrow$ off/on.

**Powers of Two**

The importance of binary means we have to get familiar with the powers of 2.

- $2^0 = 1$
- $2^1 = 2$
- $2^2 = 4$
- $2^3 = 8$
- $2^4 = 16$
- $2^5 = 32$
- $2^6 = 64$

We call one binary digit (one or zero) a **bit**. 8 bits is a **byte**.

\[2^{10} \text{ bytes} = 1 \text{ kilobyte} \]

\[2^{20} \text{ bytes} = 1 \text{ megabyte} \]

\[2^{30} \text{ bytes} = 1 \text{ gigabyte} \]

\[2^{40} \text{ bytes} = 1 \text{ terabyte} \]

Notice that the exponent goes up by increments of ten between each order of magnitude. Thus there are $2^{10}$ kilobytes in a megabyte and $2^{20}$ megabytes in a gigabyte.

**Bits and Bytes**

We call one binary digit (one or zero) a *bit*. 8 bits is a *byte*. $n$ bits can represent $2^n$ different numbers.

**Hexadecimal**

Another base that turns up in computer science is base-16, or *hexadecimal*. Base-16 requires inventing a few new digits.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hex</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>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

**Binary and Hex**

16 is $2^4$, so every hex digit can be converted to exactly four binary digits.

\[\begin{align*}
C_{16} &= 1100_2 \\
7_{16} &= 0111_2 \\
C7_{16} &= 1100 0111_2
\end{align*}\]

Converting from binary to hex is just as easy:

\[1101_2 0100_2 1011_2 0110_2 = \text{D4B6}_{16}\]

**Converting from Decimal**

Going to binary or hex from decimal is a little trickier. In general, the base-$n$ representation $(x_1x_{i-1} \ldots x_1x_0)_n$ of a number $X$ is:

\[x_j = \left\lfloor \frac{(X \mod n^{j+1})}{n^j} \right\rfloor\]

Here’s an algorithm:

1. $X' := X$
2. Find the largest $i$ such that $n^i \leq X$.
3. $x_i := X' \div n^i$, $X' := X' \mod n^i$, $i := i - 1$
4. If $i \geq 0$, return to Step 3.

**Decimal to Binary: Example**

That’s not as hard as it looks. Let’s convert 43 to binary.

\[\begin{align*}
\text{Step 1:} & \quad X' := 43 \\
\text{Step 2:} & \quad i := 43 \\
\text{Step 3:} & \quad i := 5 \quad (2^5 = 32, \quad 2^6 = 64)
\end{align*}\]

**Decimal to Binary: Example**

That’s not as hard as it looks. Let’s convert 43 to binary.

\[\begin{align*}
\text{Step 1:} & \quad X' := 43 \\
\text{Step 2:} & \quad i := 5 \quad (2^5 = 32, \quad 2^6 = 64) \\
\text{Step 3:} & \quad x_5 := 43 \div 2^5 = 43 \div 32 = 1 \\
& \quad X' := 43 \mod 32 = 11 \\
& \quad i := 5 - 1 = 4
\end{align*}\]
Decimal to Binary: Example

That's not as hard as it looks. Let's convert 43 to binary.

Step 1: \( X' := 43 \)

Step 2: \( i := 5 \; \quad (2^5 = 32, \quad 2^6 = 64) \)

Step 3: \( x_5 := 43 \div 2^5 = 43 \div 32 = 1 \)
\( X' := 43 \ mod \ 32 = 11 \)
\( i := 5 - 1 = 4 \)

Step 4: \( 4 \geq 0? \) Return to Step 3

Step 5: \( 3 \geq 0? \) Return to Step 3

Step 6: \( x_3 := 11 \div 2^3 = 11 \div 8 = 1 \)
\( X' := 11 \mod 8 = 3 \)
\( i := 3 - 1 = 2 \)

Step 7: \( 2 \geq 0? \) Return to Step 3

Step 8: \( 1 \geq 0? \) Return to Step 3

Step 9: \( x_1 := 3 \div 2^1 = 3 \div 2 = 1 \)
\( X' := 3 \mod 2 = 1 \)
\( i := 1 - 1 = 0 \)

Step 10: \( 0 \geq 0? \) Return to Step 3

Done.

Step 1: \( X' := 43 \)
Step 2: \( i := 5 \; \quad (2^5 = 32, \quad 2^6 = 64) \)
Step 3: \( x_5 := 43 \div 2^5 = 43 \div 32 = 1 \)
Step 4: \( X' := 43 \mod 32 = 11 \)
\( i := 5 - 1 = 4 \)
Step 5: \( 4 \geq 0? \) Return to Step 3
Step 6: \( x_4 := 11 \div 2^4 = 11 \div 16 = 0 \)
\( X' := 11 \mod 16 = 11 \)
\( i := 4 - 1 = 3 \)
Step 7: \( 3 \geq 0? \) Return to Step 3
Step 8: \( x_3 := 11 \div 2^3 = 11 \div 8 = 1 \)
\( X' := 11 \mod 8 = 3 \)
\( i := 3 - 1 = 2 \)
Step 9: \( 2 \geq 0? \) Return to Step 3
Step 10: \( x_2 := 3 \div 2^2 = 3 \div 4 = 0 \)
\( X' := 3 \mod 4 = 3 \)
\( i := 2 - 1 = 1 \)
Step 11: \( 1 \geq 0? \) Return to Step 3
Step 12: \( x_1 := 3 \div 2^1 = 3 \div 2 = 1 \)
\( X' := 3 \mod 2 = 1 \)
\( i := 1 - 1 = 0 \)
Step 13: \( 0 \geq 0? \) Return to Step 3

Done.

Step 1: \( X' := 43 \)
Step 2: \( i := 5 \; \quad (2^5 = 32, \quad 2^6 = 64) \)
Step 3: \( x_5 := 43 \div 2^5 = 43 \div 32 = 1 \)
Step 4: \( X' := 43 \mod 32 = 11 \)
\( i := 5 - 1 = 4 \)
Step 5: \( 4 \geq 0? \) Return to Step 3
Step 6: \( x_4 := 11 \div 2^4 = 11 \div 16 = 0 \)
\( X' := 11 \mod 16 = 11 \)
\( i := 4 - 1 = 3 \)
Step 7: \( 3 \geq 0? \) Return to Step 3
Step 8: \( x_3 := 11 \div 2^3 = 11 \div 8 = 1 \)
\( X' := 11 \mod 8 = 3 \)
\( i := 3 - 1 = 2 \)
Step 9: \( 2 \geq 0? \) Return to Step 3
Step 10: \( x_2 := 3 \div 2^2 = 3 \div 4 = 0 \)
\( X' := 3 \mod 4 = 3 \)
\( i := 2 - 1 = 1 \)
Step 11: \( 1 \geq 0? \) Return to Step 3
Step 12: \( x_1 := 3 \div 2^1 = 3 \div 2 = 1 \)
\( X' := 3 \mod 2 = 1 \)
\( i := 1 - 1 = 0 \)
Step 13: \( 0 \geq 0? \) Return to Step 3

Done.
Step 4: 1 ≥ 0? Return to Step 3
Step 3: \( x_1 := \frac{3}{2^1} = \frac{3}{2} = 1 \)
\( X' := 3 \mod 2 = 1 \)
i := 1 – 1 = 0
Step 4: 0 ≥ 0? Return to Step 3
Step 3: \( x_0 := \frac{1}{2^0} = \frac{1}{1} = 1 \)
\( X' := 1 \mod 1 = 0 \)
i := 0 – 1 = -1
Step 4: -1 ≥ 0? Done.

Decimal to Hex: Example

Decimal to Hex: Example

Decimal to Hex: Example

Step 1: \( X' := 43 \)

Step 1: \( X' := 43 \)

Step 1: \( X' := 43 \)

\( i := 1 \) (16\(^1\) = 16, 16\(^2\) = 256)

\( i := 1 \) (16\(^1\) = 16, 16\(^2\) = 256)

\( i := 1 \) (16\(^1\) = 16, 16\(^2\) = 256)

Step 2: \( 43 \div 16 = 2 \) ...

Step 2: \( 43 \div 16 = 2 \) ...

Step 2: \( 43 \div 16 = 2 \) ...

\( X' := 43 \mod 16 = 11 \)
i := 1 – 1 = 0

\( X' := 43 \mod 16 = 11 \)
i := 1 – 1 = 0

\( X' := 43 \mod 16 = 11 \)
i := 1 – 1 = 0

Step 3: \( 0 ≥ 0? \) Return to Step 2

Step 3: \( 0 ≥ 0? \) Return to Step 2

Step 3: \( 0 ≥ 0? \) Return to Step 2

\( x_0 := 11 \div 16^0 = 11 \div 1 = 11 = [B] \)

\( x_0 := 11 \div 16^0 = 11 \div 1 = 11 = [B] \)

\( x_0 := 11 \div 16^0 = 11 \div 1 = 11 = [B] \)

\( X' := 11 \mod 1 = 0 \)
i := 0 – 1 = -1

\( X' := 11 \mod 1 = 0 \)
i := 0 – 1 = -1

\( X' := 11 \mod 1 = 0 \)
i := 0 – 1 = -1

Step 4: -1 ≥ 0? Done.

43\(_{10}\) = 101011\(_2\)
Decimal to Hex: Example

Step 1: \( X' := 43 \)

Step 2: \( i := 1 \) \((16^1 = 16, \quad 16^2 = 256)\)

\[ x_1 := 43 \div 16^1 = 43 \div 16 = 2 \]

\( X' := 43 \mod 16 = 11 \)

\( i := 1 - 1 = 0 \)

Step 3: \( 0 \geq 0? \) Return to Step 3

Step 4: \( x_0 := 11 \)

\( X' := 11 \mod 1 = 0 \)

\( i := 0 - 1 = -1 \)

\( -1 \geq 0? \) Done.

\[ 43_{10} = 2B_{16} = 0010 1011_2 \]

Computer Architecture

Your average desktop computer has:

- A CPU (central processing unit) - executes program instructions
- Main memory (RAM) - stores program instructions and data during execution. Volatile (i.e., goes away when the power goes off).
- Secondary storage (disks) - permanently stores program instructions and data.
- Input devices - keyboard, mouse, etc.
- Output devices - monitor, printer, etc.

Machine Code

As you might expect, the instructions that the CPU executes are encoded in binary. We call the CPU's binary instruction format **machine code**.

E.g., \( a := b + c \) in MIPS machine code could be:

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

Each instruction is composed of fields which are relatively easy to decode. You can look up the fields in the processor's manual.

\[ 000000 \quad 00010 \quad 00011 \quad 00001 \quad 00000 \quad 100000 \]

Assembly Code

The first layer of abstraction for the programmer is **assembly language**. Assembly language is essentially a mnemonic form of machine language.

\[ a := b + c \rightarrow \text{add} \$1, \$2, \$3 \]

**Idealization**: we can simply tell the processor what we want it to do.

**What we ignore**: that the instructions must be encoded in a particular binary format.

The program that turns our assembly instructions into machine code is called an assembler.

Layers of Abstraction

A layer of abstraction is an idealization of reality, for the purpose of simplifying a task where the details aren’t important.

E.g., Newtonian mechanics is a layer of abstraction on general relativity where \( v/c \approx 0 \).

In computer science, layers of abstraction let us concentrate on solving the problem and let the computer take care of the unimportant details.

Programming languages with many layers of abstraction are **high-level languages**. Programming languages with few or no layers of abstraction are **low-level languages**.

Assembly Code

Assembly is still very low-level. There is a more-or-less one-to-one correspondence between assembly instructions and machine instructions. This can be tedious.

\[ c := \sqrt{a^2 + b^2} \rightarrow \text{\textbf{lw}} \$1, a \]

\[ \text{\textbf{lw}} \$2, b \]

\[ \text{\textbf{mult}} \$1, \$1, \$1 \]

\[ \text{\textbf{mult}} \$2, \$2, \$2 \]

\[ \text{\textbf{add}} \$4, \$1, \$2 \]

\[ \text{\textbf{sqrt}} \$3, \$4 \]

\[ \text{\textbf{sw}} c, \$3 \]

Another Layer: C

C is the most enduringly popular of the higher-level languages invented in the 60's and 70's. C allows us to communicate our desires to the CPU without explicitly describing every step.

\[ c := \sqrt{a^2 + b^2} \quad \rightarrow \quad c = \text{sqrt}(a*a + b*b) ; \]

**Idealization**: we can write expressions in a syntax that looks familiar to us.

**What we ignore**: these expressions must be translated into many separate machine instructions (maybe dozens).

Compilers

The program that turns our C **source code** into machine code is called a **compiler**. Every high-level language needs a compiler to turn code that humans can read and write into code that the processor can execute.

\[ \text{C source} \quad \rightarrow \quad \text{C compiler} \quad \rightarrow \quad \text{Machine code} \]
Compilers, 2

There isn’t usually only one way to translate a source program into executable code. The compiler has to analyze the source code and try to choose the best translation into machine code.

What’s “best” depends on what you want out of the program:

- Speed
- Small code (fewer instructions)
- Retention of source structure (for debugging)

C: Portability

C is designed to be highly portable. That means that it is possible to compile and run the same program on the 8-bit Intel 8088 and a 64-bit Sparc.

C achieves portability partly by hiding details about the size of numbers. C programs should be written so they run equally well no matter how many bits are used in the arithmetic.

Unfortunately, this is hard to do well. Most C programs are portable only to a certain class of processors (e.g., 32-bit programs that run on Linux and Windows).

Java: The Pros

- Byte code is platform-independent. Any platform with a JVM can run any Java program without recompiling.
- Java is secure. The JVM implements security policies to prevent unsafe code from running.
- Java is easier to read and write than C and its variants.

Summary: Java makes writing Internet applications easy and safe.

Java: The Cons

- Java is slow (~10x slower than C).
- Java cannot directly access hardware.
- Java is big. It is hard to cram onto handhelds and other embedded devices.

Summary: Java is not a good choice when you need raw speed and access to the underlying hardware. (In other words, Java is not a good choice when you really should be using C.)

Why do we use Java?

We use Java because:

- It’s relatively easy to learn.
- It protects you from a lot of beginner’s mistakes.

In other words, it offers a high level of abstraction, and we don’t want to distract you with a lot of the machine-level details (yet). Also:

- It’s more likely to get you a job than any of the alternatives.

Another Layer: Java

Java was invented in the mid-1990’s for writing Internet applications. Its design goals were portability and security. Java guarantees that numbers will be the same size on every platform, no matter what the “native” size of the machine.

Idealization: Every computer that our program runs on looks exactly the same from the programmer’s point of view.

What We Ignore: There are major differences in the way different processors achieve the same operations.

The Java Virtual Machine

The Java compiler doesn’t produce machine code. It produces byte code for the Java Virtual Machine (JVM). The JVM for a platform reads byte code and translates it to machine code at run time.

Java: The Pros

- Byte code is platform-independent. Any platform with a JVM can run any Java program without recompiling.
- Java is secure. The JVM implements security policies to prevent unsafe code from running.
- Java is easier to read and write than C and its variants.

Summary: Java makes writing Internet applications easy and safe.

Java: The Cons

- Java is slow (~10x slower than C).
- Java cannot directly access hardware.
- Java is big. It is hard to cram onto handhelds and other embedded devices.

Summary: Java is not a good choice when you need raw speed and access to the underlying hardware. (In other words, Java is not a good choice when you really should be using C.)