Introduction

Week 1
CS 212 – Spring 2008

Computer Science 212
Programming Practicum

- One credit course
  - Grade is based entirely on programming assignments
  - No exams
- Meetings
  - One lecture per week
  - One section per week
- Course objectives
  - Improve programming skill
  - Learn something about software engineering
  - Develop project management skills
  - Learn about computer science

Mundane Details

- Staff
  - Instructor: Paul Chew
  - Course Administrator: Kelly Patwell
  - TAs: Etan Bukiet, Zoe Chiang, Jimmy Hartzell, Ken Kruger, Cangming (Geoff) Liu, Dan Perelman, Chuck Sakoda, Ozzie Smith
  - Consultants: none (but the 211 consultants can help with general Java questions)
- Text
  - None required, but some Java texts that might be helpful are listed on the 211 website
- Lecture
  - W 3:35 – 4:25, Phillips 203
- Sections (beginning Jan 28)
  - M 12:20 - 1:10 in Olin Hall 245
  - M 7:30 - 8:20 in Upson 205
  - W 7:30 - 8:20 in Upson 205
- Website:
  - cs.cornell.edu/courses/212/

Announcements

- Sections start this next week (beginning Jan 28)
- We use CMS (Course Management System) for maintaining grade information
  - Make sure you're on CMS
  - Notify the course administrator (see website) if you're not
- The first assignment (Part 1) will appear on the website next week

Lecture Topics

- Programming in a group
- Software engineering
  - Abstraction
  - Specification
  - Models for software development
- Software testing
  - Unit testing vs. integration testing
- Software tools
  - Scripting languages
  - Regular expressions
  - Use of standard data structures
  - Version control systems
  - Profilers
- Programming languages
- Computer architecture and the JVM
- Compilers, syntax, context free grammars
- Recursive descent parsing, abstract syntax trees
- Runtime stack, implementing functions, recursion
- Pointers, the heap
- Implementing objects

- No exams, but…

There is a Project

- A single large project over the semester
  - Typically, split into 4 parts
  - Students are encouraged to work in groups of 2 or 3
- This semester, we offer a choice of two projects:
  - Compiler Project
  - GBA (Game Boy Advance) Project

Section Topics

- Help for the project
- Also
  - Assembly language
    - Looping & branching
    - Calling functions
  - Recursive descent parsing
  - Using a debugger
  - Implementing recursive functions
  - Understanding the heap

The Compiler Project

- Build a compiler for a Java-like language called *Bali*

An island of southern Indonesia in the Lesser Sundas just east of Java

- Compiled code: sam-code
  - Resembles (sort of) Java Byte Code (JBC)
  - Runs on SaM (Stack Machine)
    - A simplified substitute for the JVM (Java Virtual Machine)
- Part 1
  - Introduction to SaM, simple expressions
- Part 2
  - Compiling expressions, control structures
- Part 3
  - Compiling functions
- Part 4
  - Compiling (simple) classes

The GBA Project

- Build a set of game-design tools and a game for the Nintendo Game Boy Advance

- Part 1
  - Practice with C++ and the GBA; create a Pong game using a simplified interface
- Part 2
  - Given specifications, write C++ code to manipulate sprites and background on the GBA
- Part 3
  - Design and implement a sprite manager for the GBA
- Part 4
  - Use the sprite manager to build a game (e.g., Space Invaders, Pacman)

Software

- For the Compiler Project
  - JDK (Java Development Kit) 6
  - An IDE (Interactive Development Environment): *Eclipse* is recommended
  - See the CS 211 website for additional details
- For the GBA Project
  - Uses a C++ to ARM-code compiler (the GBA has an ARM processor)
  - Uses a GBA emulator for Windows
    - Additional hardware is needed to transfer a program to the GBA

Both Projects

- Involve substantial programming broken into coherent parts over the semester
- Include interesting/challenging design choices
- Use object-oriented programming
  - Java for the Compiler Project
  - C++ for the GBA Project
- Provide a useful model of how computers work
- Allow students to gain experience working in groups

Picking a Project

- Things to keep in mind
  - The GBA Project is probably more work (need to learn a good-sized chunk of C++)
- We will post an "assignment" on CMS asking for
  - Project preference
  - Section availability
  - Programming experience
  - This assignment should appear later this week

Working in Groups

- Work individually on first assignment (Part 1)
- After that, partners are allowed/encouraged
  - Good practice for group-projects in later courses
  - Groups of 2 or 3
- Partnership rules
  - You choose group
  - For a given assignment, once you start with a group, you must continue
  - You may not work with different partners for different parts of the same assignment
  - Can change groups for each assignment
  - More details on course website

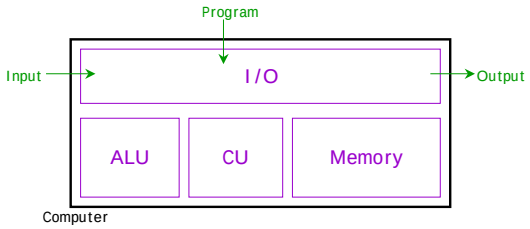
When to Take CS212

- At same time as CS211
  - Some coordination of topics
  - Coordination of assignment due dates
- After CS211
  - You'll have more experience
  - But possibly less connection with *your* CS211
- Before CS211
  - No!

Computer Architecture: Memory

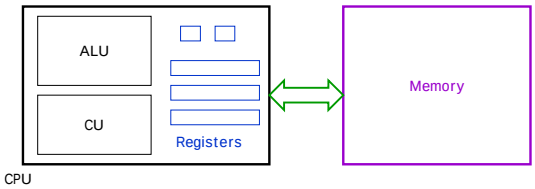
- A computer contains a large collection of circuits that can be used to store *bits* (a bit is a 0 or a 1)
  - Bits are grouped into *bytes* (8 bits)
  - Bytes are grouped into words or *cells*
- *Memory* consists of a large collection of cells
  - Each memory cell has an *address* (usually from 0 to numCells–1)
  - Cells can be accessed in any order
  - Computer memory is called
    - *Main memory* or
    - *RAM* (Random Access Memory) or
    - (obsolete) *core memory*

Von Neumann Model



- Memory: holds both data and program
- Arithmetic Logic Unit: handles arithmetic and logic calculations
- Control Unit: interprets instructions; controls ALU, Memory, I/O
- I/O: storage, input, output

Central Processing Unit (CPU)



- Registers hold small amounts of data
  - PC: program counter
  - IR: instruction register (current instruction)
  - SP: stack pointer
  - more...

Machine Language

- Used with the earliest electronic computers (1940s)
  - Machines use vacuum tubes (instead of transistors)
- Programs are entered by setting switches or reading punch cards
- All instructions are numbers
- Example code

```
0110 0001 0000 0110
Add  Reg1        6
```

- Idea for improvement
  - Let's use *words* instead of numbers
  - Result: Assembly Language

Assembly Language

- Idea: Use a program (an *assembler*) to convert assembly language into machine code
- Early assemblers were some of the most complicated code of the time (1950s)

- Example code

```
ADD R1 6
MOV R1 COST
SET R1 0
JMP TOP
```

  - Typically, an assembler used *2 passes*
- Idea for improvement
  - Let's make it easier for humans by designing a more powerful computer language
  - Result: high-level languages

High-Level Language

- Idea: Use a program (a *compiler* or an *interpreter*) to convert high-level code into machine code
- Pro
  - Easier for humans to write, read, and maintain code
- Con
  - The resulting program will never be as efficient as good assembly-code
    - Waste of memory
    - Waste of time
- The whole concept was initially controversial
  - Thus, FORTRAN (mathematical FORmula TRANslating system) was designed with efficiency very-much in mind

FORTRAN

- Initial version developed in 1957 by IBM

- Example code

```
C       SUM OF SQUARES
        ISUM = 0
        DO 100 I=1,10
        ISUM = ISUM + I*I
  100 CONTINUE
```

- FORTRAN introduced many of the ideas typical of programming languages
  - Assignment
  - Loops
  - Conditionals
  - Subroutines

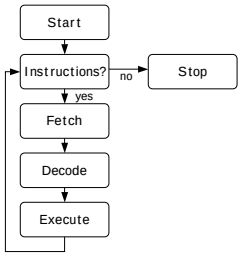
Machine Language vs. Assembly Language

- Machine Language
  - Instructions and coding scheme used internally by computer
  - Humans do not usually write machine language
  - Typical machine language instructions have two parts
    - Op-code (operation code)
    - Operand
- Assembly Language
  - Symbolic representation of machine language
  - Use mnemonic word for op-code
    - Example: PUSHIMM 5
  - Typically provide additional features to help make code readable for humans
    - Example: names as labels instead of numbers

Machine Instruction Categories

- Data transfer
  - Copy data from one memory location to another
    - LOAD: copy data from a memory cell to a register
    - STORE: copy data from a register to a memory cell
    - I/O instructions
- Arithmetic / Logic
  - Request activity in ALU
    - Arithmetic (ADD, SUB, TIMES, ...)
    - Logic (AND, OR, NOT, XOR)
    - SHIFT, ROTATE
- Control
  - Direct execution of program
    - JUMP, JUMPC (conditional jump)

Fetch and Decode Cycle



- Control Unit (CU) fetches next instruction from memory at address specified by Program Counter (PC)
- CU places instruction into the instruction register (IR)
- CU increments PC to prepare for next cycle
- CU decodes instruction to see what to do
- CU activates correct circuits to execute the instruction (e.g., ALU performs an addition)

Java Byte Code (JBC)

- A Java compiler creates Java Byte Code (JBC)
  - A sequence of bytes
  - Not easily readable by humans
  - JBC is machine code for a *virtual* (pretend) computer called the *Java Virtual Machine* (JVM)
  - A *byte code interpreter* reads and executes each instruction
- javap –c classfile
  - Can use this to see JBC

Java Virtual Machine (JVM)

- JBC is code for the JVM
  - No such machine really exists
  - A *JVM interpreter* must be created for each machine architecture on which JBC is to run
- The JVM is designed as an "average" computer
  - Uses features that are widely available (e.g., a stack)
- Design goals
  - Should be easy to convert Java code into JBC
  - Should be reasonably easy to create a JVM interpreter for most computer architectures

SaM (Stack Machine)

- Goals
  - Approximate the JVM
  - But simpler
- We produce sam-code, assembly language for SaM, our own virtual machine
- We have a SaM Simulator (thanks David Levitan) that we can use to execute sam-code
- In place of
  JBC for the JVM
- We will produce
  sam-code for SaM

Some Sam-Code Instructions

- SaM's main memory is maintained as a Stack
- The SP (stack pointer) register points at the next empty position on the stack
  - The first position has address 0
  - Addresses increase as more items are pushed onto the Stack
- PUSHIMM c
  - (push immediate)
  - Push integer c onto Stack
- ADD
  - Add top two Stack items, removing those items, and pushing result onto Stack
- SUB
  - Subtract top two Stack items, removing those items, and pushing result onto Stack
  - Order is important
    - stack[top-1] – stack[top]

More Sam-Code Instructions

- ALU Instructions
  - ADD, SUB, TIMES, DIV
  - NOT, OR, AND
  - GREATER, LESS, EQUAL
- Stack Manipulation Instructions
  - PUSHIMM c
  - DUP, SWAP
  - PUSHIND
    - (push indirect)
    - Push stack[stack[top]] onto Stack
  - STOREIND
    - (store indirect)
    - Store stack[top] into stack[stack[top-1]]