About the Course

- its all on the web page ...
  - [http://www.cs.ubc.ca/~feeley/cs213](http://www.cs.ubc.ca/~feeley/cs213)
  - Lecture Notes Companion
  - Piazza
- marks
  - in-class clicker questions *(you will need a clicker)*
  - labs
  - quizzes
  - midterm
  - final
- work together! *but don’t cheat!*
  - never present anyone else’s work as your own
    - it is your responsibility to provide proper attribution
  - anything you hand in in this course should follow this rule *anything*
  - but, don’t let this stop you from helping each other learn ...
Overview of the course

› Hardware context of a single executing program
  • hardware context is CPU and Main Memory
  • develop CPU architecture to implement C and Java
  • differentiate compiler (static) and runtime (dynamic) computation

› System context of multiple executing programs with IO
  • extend context to add IO, concurrency and system software
  • thread abstraction to hide IO asynchrony and to express concurrency
  • synchronization to manage concurrency
  • virtual memory to provide multi-program, single-system model
  • hardware protection to encapsulate operating system
  • message-passing to communicate between processes and machines

GOAL: To develop a model of computation that is rooted in what really happens when programs execute.

What you will get out of this ...

› Become a better programmer by
  • deepening your understand of how programs execute
  • learning to build concurrent and distributed programs

› Learn to design real systems by
  • evaluating design trade-offs through examples
  • distinguish static and dynamic system components and techniques

› Impress your friends and family by
  • telling them what a program really is
What do you know now?

What happens when a program runs

Here’s a program

class SortedList {
    static SortedList aList;
    int size;
    int list[];

    void insert (int aValue) {
        int i = 0;
        while (list[i] <= aValue)
            i++;
        for (int j=size-1; j>=i; j--)
            list[j+1] = list[j];
        list[i] = aValue;
        size++;
    }
}

What do you understand about the execution of \texttt{insert}?
Example

- list stores \{1, 3, 5, 7, 9\}
- `SortedList.aList.insert(6)` is called

Data structures

- draw a diagram of the data structures
- as they exist just before `insert` is called

```java
class SortedList {
    static SortedList aList;
    int size;
    int list[];

    void insert (int aValue) {
        int i = 0;
        while (list[i] <= aValue) i++;
        for (int j=size-1; j>=i; j--) list[j+1] = list[j];
        list[i] = aValue;
        size++;
    }
}
```

Data structures

- let's dig a little deeper
- which of these existed before program started?
  - these are the *static* features of the program
- which were created by execution of program?
  - these are the *dynamic* features of the program

```
Static:
* class and aList variable
  (sort of - clearer in C)

Dynamic:
* SortedList object
* size and list variables
* value of aList, size and list
* list of 10 integers
```
Execution of insert
• how would you describe this execution?
  • carefully, step by step?

Sequence of Instructions
* program order
* changed by control-flow structures

```java
class SortedList {
    static SortedList aList;
    int size;
    int list[];

    void insert (int aValue) {
        int i = 0;
        while (list[i] <= aValue) {
            i++;
            for (int j = size-1; j >= i; j--)
                list[j+1] = list[j];
            list[i] = aValue;
            size++;
        }
    }
}
```

Instruction Types?
* read/write variable
* arithmetic
* conditional goto

Data structures
• variables have a storage location and a value
• some variables are created before the program starts
• some variables are created by the program while it runs
• variable values can be set before program runs or by the execution

Execution of program statements
• execution is a sequence of steps
• sequence-order can be changed by certain program statements
• each step executes an instruction
• instructions access variables, do arithmetic, or change control flow
An Overview of Computation

Phases of Computation

- **Human creation**
  - design program and describe it in high-level language

- **Compilation**
  - convert high-level, human description into machine-executable text

- **Execution**
  - a physical machine executes the text
  - parameterized by input values that are unknown at compilation
  - producing output values that are unknowable at compilation

Two important initial definitions

- anything that can be determined **before execution** is called *static*
- anything that can only be determined **during execution** is called *dynamic*
Examples of Static vs Dynamic State

- Static state in Java
- Dynamic state in Java

A Simple Machine that can Compute

- Memory
  - stores programs and data
  - everything in memory has a unique name: its memory location (address)
  - two operations: read or write value at location X
- CPU
  - machine that executes programs to transform memory state
  - loads program from memory on demand one step at a time
  - each step may also read or write memory
- Not in the Simple Machine
  - I/O Devices such as mouse, keyboard, graphics, disk and network
  - we will deal with these other things in the second half of the course
The Simple Machine Model
A Closer Look

How do we start?

- One thing we need to do is add integers
  - you already know how to do this from 121 (hopefully :))
- A 32-bit Adder
  - implemented using logic gates implemented by transistors
  - it adds bits one at a time, with carry-out, just like in grade 2.

```
+-------------------+             +-------------------+
| INPUT register    |             | INPUT register    |
|                   |             |                   |
|                   |             |                   |
|                   |             |                   |
|                   |             |                   |
| OUTPUT register   |             | OUTPUT register   |
```
Generalizing the Adder

‣ What other things do we want to do with Integers

‣ What do we do with the value in the output register

Register File and ALU

‣ Arithmetic and Logic Unit (ALU)
  • generalizes ADDER to perform many *operations* on integers
  • three inputs: two source *operands* (valA, valB) and a *operation code* (opCode)
  • output value (valE) = operation-code (operand0, operand1)

‣ Register File
  • generalizes input and output registers of ADDER
  • a single bank of registers that can be used for input or output
  • registers *named* by *numbers*: two source (srcA, srcB) and one destination (dst)
Functional View
• input for one step: opCode, srcA, srcB, and dst
• a program is a sequence of these steps (and others)

Current model is too restrictive
• to add two numbers the numbers must be in registers
• programs must specify values explicitly

Extend model to include *immediates*
• an *immediate value* is a constant specified by a program instruction
• extend model to allow some instructions to specify an immediate (valC)
Functional View

- We now have an additional input, the immediate value, valC

Memory Access

- Memory is an array of bytes, indexed by byte address

- Memory access is restricted to a transfer between registers and memory
  - The ALU is thus unchanged, it still takes operands from registers
  - This is approach taken by Reduced Instruction Set Computers (RISC)

- Extending model to include RISC-like memory access:
  - Opcode selects from set of memory-access and ALU operations
  - Memory address and value are in registers
Central Processing Unit or Core (CPU)
- a register file
- logic for ALU, memory access and control flow
- a clock to sequence instructions
- memory cache of some active parts of memory (e.g., instructions)

Memory
- is too big to fit on the CPU chip, so it's stored off chip
- much slower than registers or cache (200 x slower than registers)

A Program
- sequence of instructions stored in memory

An Instruction
- does one thing: math, memory-register transfer, or flow control
- specifies a value for each of the functional inputs
Instruction Set Architecture (ISA)

- The ISA is the “interface” to a processor implementation
  - defines the instructions the processor implements
  - defines the format of each instruction

- Instruction format
  - is a set of bits (a number)
  - an opcode and set of operand values

- Types of instruction
  - math
  - memory access
  - control transfer (gotos and conditional gotos)

- Design alternatives
  - simplify compiler design (CISC such as Intel Architecture 32)
  - simplify processor implementation (RISC)

Assembly language
- symbolic representation of machine code

Example Instruction: ADD

- Description
  - opCode = 61
  - two source operands in registers: srcA = rA, srcB = rB
  - put destination in register: dst = rB

- Assembly language
  - general form: add rA, rB
  - e.g., add r0, r1

- Instruction format
  - 16 bit number, divided into 4-bit chunks: 61sd
  - high-order 8 bits are opCode (61)
  - next 4 bits are srcA (s)
  - next 4 bits are srcB/dst (d)

- Assembly code
  - add rA, rB
  - add r0, r1
Simulating a Processor Implementation

- Java simulator
  - edit/execute assembly-language
  - see register file, memory, etc.

- You will implement
  - the **fetch** + **execute** logic
  - for every instruction in SM213 ISA

- SM213 ISA
  - developed as we progress through key language features
  - patterned after MIPS ISA, one of the 2 first RISC architectures