CPSC 213

Introduction to Computer Systems

Unit 0

Introduction

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

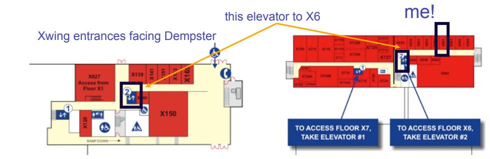
GOAL: To develop a model of computation that is rooted in what really happens when programs execute.

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

About the Course - Logistics

- it's all on the web page ...
  - http://www.ugrad.cs.ubc.ca/~cs213/winter12t1
    - news, admin details, schedule and readings
    - lecture slides (always posted before class)
    - *213 Companion* (free PDF)
    - Piazza for discussion
    - marks (coming soon) secure download
  - updated often, don't forget to reload page!
- me
  - instructor: Tamara Munzner
    - call me Tamara or Dr. Munzner, as you like
    - office hours in X661 Mon/Fri 9-10am or by appointment



Reading

- see web page for exact schedule
- textbook: Bryant and O'Hallaron
  - also used in CPSC 313 followon course
  - ok to use either 1st or 2nd edition (very little difference for us)
- 213 Companion
  - additional reading; PDF posted on web page

Course Policies

- read http://www.ugrad.cs.ubc.ca/~cs213/policies.html
- marking
  - assignments: 20%
    - 9 labs/assignments (same thing, no separate lab material)
    - usually one week for each, out Monday morning and due next Monday 6pm
      - exceptions for exam weeks, to give you time for studying
  - quizzes: 30%
    - Oct 15, Nov 5
  - final: 50%
    - date TBD. do not book tickets out of town until announced!
  - must pass labs and quizzes and final (50% or better) to pass course
- assignments
  - critical for learning material
  - they build on each other; don't fall behind
  - come get help if you get stuck - labs, office hours...

Scaling and Regrading

- I often scale exams
  - so don't panic if it seems hard while you're taking it!
- regrading
  - detailed argument in writing required (email or paper)
  - read through solutions first; no requests accepted until 24 hours after work is returned
  - email TA first for assignments, then instructor if not resolved
  - bring paper to instructor for quizzes/midterms

Late/Missed Work, Illness

- late work penalty
  - 25% first day (or fraction of day)
  - 50% second day (or fraction thereof)
  - no late work accepted after 48 hrs
    - no exceptions
    - handin drafts early, handin often: do not wait until last minute!
    - check what you have handed in!
- email me immediately if you'll miss lab/exam from illness
- written documentation due within 7 days after you return to school
  - copy of doctor's note or other proof (ICBC accident report, etc)
  - written cover sheet with dates of absence and list of work missed
- I'll decide on how to handle
  - might give extension if solutions not out yet
  - might grade you only on completed work

Cheating: Things I Never Want To Hear

- read http://www.ugrad.cs.ubc.ca/~cs213/cheat.html
- Cheating: The List Of Things I Never Want To Hear Again
  - read this page, ask if you have any questions!
  - you must sign statement that you have read and completely understood this page before turning in assignments
  - http://www.cs.ubc.ca/~tmm/courses/cheat.html
- the bottom line
  - the fundamental reason not to cheat is you don't learn the material
  - you need to work through the labs yourself to learn this stuff!
  - if you cheat on the labs, you will fail the exams

Course-Specific Guidelines

- work together and help each other - ***but don't cheat!***
  - never present anyone else's work as your own
  - but, don't let this stop you from helping each other learn...
    - general discussion always fine
    - one-hour context switch rule for specific discussions (Gilligan's Island rule)
      - don't take written notes
      - do something else for an hour
      - then sit down to do the work on your own
    - proper attribution
      - include list of names if you had significant discussions with others
  - not allowed
    - working as a team and handing in joint work as your own
    - looking at somebody else's paper or smuggling notes into exam
    - getting or giving code, electronically or hardcopy
    - typing in code from somebody else's screen
    - using code from previous terms
    - paying somebody to write your code
  - it's a bad idea: you don't learn the stuff, and we'll probably catch you
    - I do prosecute, so that it's a level playing field for everybody else
    - possible penalties: 0 for the work, 0 for the course, permanent notation in transcript, suspended...

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++;
  }
}
```

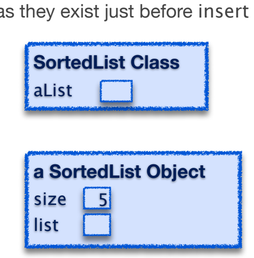
- What do you understand about the execution of **insert**?

- 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

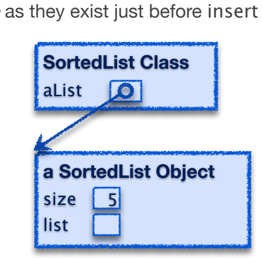

```
class SortedList {
  static SortedList aList;
  int   size;
  int   list[];

  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++;
  }
}
```


- 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

- 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



- 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



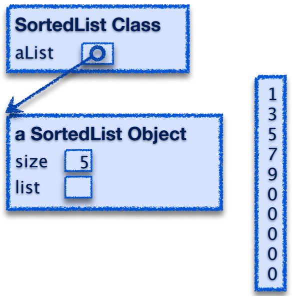
Example

- 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

```
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++;
  }
}
```



assuming list[] was initialized to store 10 elements:

list = new Integer[10];

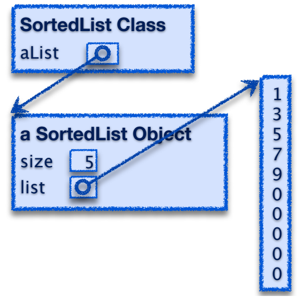
Example

- 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

```
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++;
  }
}
```



assuming list[] was initialized to store 10 elements:

list = new Integer[10];

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

```
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++;
  }
}
```

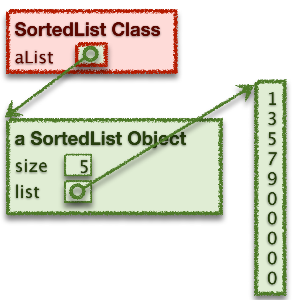

Data structures

- 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)

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?

Execution of insert

- how would you describe this execution?
- carefully, step by step?

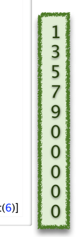
Sequence of Instructions
* program order
* changed by control-flow structures

Execution of insert

- how would you describe this execution?
- carefully, step by step?

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

```
            [execute SortedList.aList.insert(6)]
            aValue = 6
            i = 0
            if list[i]>aValue goto end-while (1>6)
            i = 0+1 (1)
            if list[i]>aValue goto end-while (3>6)
            i = 1+1 (2)
            if list[i]>aValue goto end-while (5>6)
            i = 2+1 (3)
            if list[i]>aValue goto end-while (7>6)
end-while:  j = size-1 (4)
            if j<i goto end-for (4<3)
            list[i+1] = list[i] (list[5]=9)
            j = 4-1 (3)
            if j<i goto end-for (3<3)
            list[i+1] = list[i] (list[4]=7)
            j = 3-1 (2)
            if j<i goto end-for (2<3)
end-for:    list[i] = aValue (list[3] = 6)
            size = size+1 (6)
            [statement after SortedList.aList.insert(6)]
```



Execution of insert

- how would you describe this execution?
- carefully, step by step?

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

Instruction Types?

Execution of insert

- how would you describe this execution?
- carefully, step by step?

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

Instruction Types?

* read/write variable
* arithmetic
* conditional goto

Execution: What you Already Knew

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

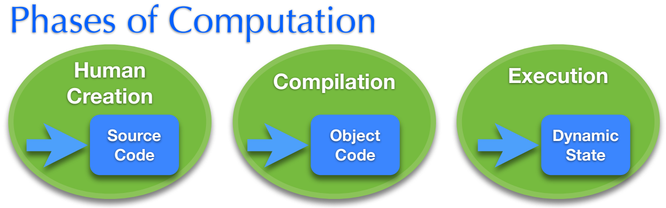
An Overview of Computation

Reading

Companion

- 1, 2.1

Phases of Computation



Human creation

- design program and describe it in high-level language

Compilation

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

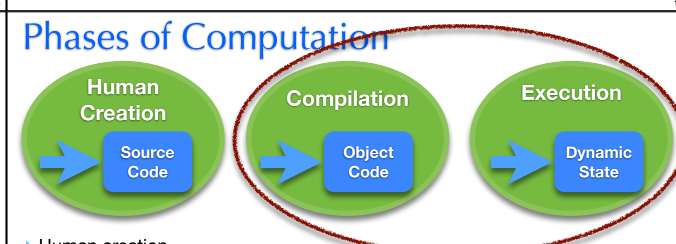
Execution

- 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***

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

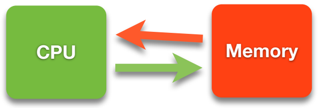
- anything that can be determined **before execution** is called ***static***
- anything that can only be determined **during execution** is called ***dynamic***

Examples of Static vs Dynamic State

Static state in Java

Dynamic state in Java

A Simple Machine that can Compute

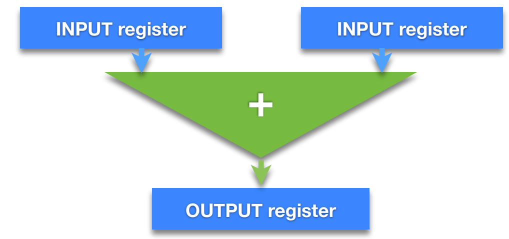


- Memory
  - stores programs and data
  - everything in memory has a unique name: its memory location (***address***)
  - two operations: read or write value at location X
- CPU
  - machine that executes programs to transform memory state
  - reads program from memory on demand one step at a time
  - 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.

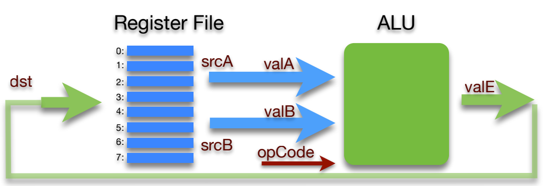


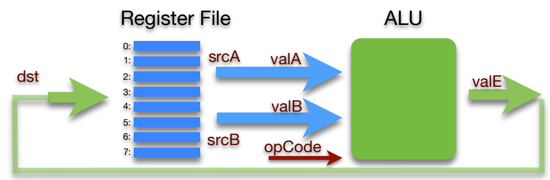
Generalizing the Adder

- 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 an ***operation code*** (opCode)
  - output value (valE) = operation-code (operand$_0$, operand$_1$)
- 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)

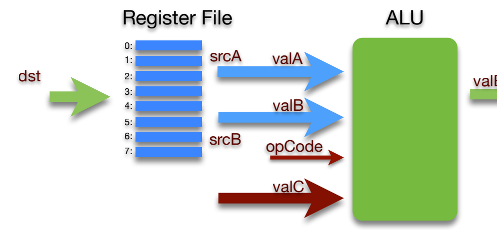


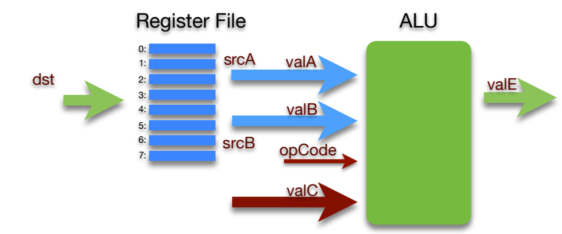


- Functional View
  - input for one step: opCode, srcA, srcB, and dst
  - a program is a sequence of these steps (and others)

Putting Initial Values into Registers

- 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)

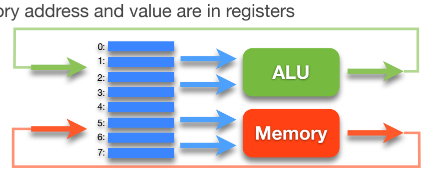




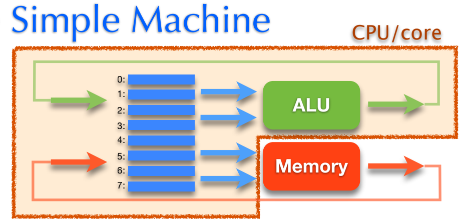
- Functional View
  - we now have an additional input, the immediate value, valC

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

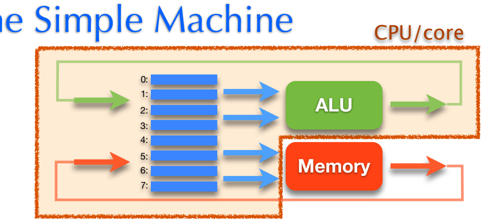


The Simple Machine



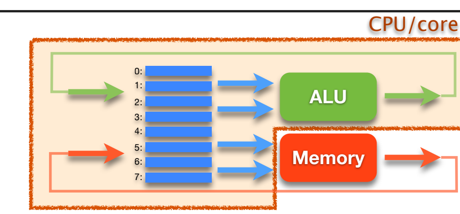
- Central Processing Unit or Core (CPU)
  - a register file
  - logic for ALU, memory access and control flow
  - a clock to sequence instructions
  - memory ***cache*** of some active parts of memory (e.g., instructions)
- Memory
  - is too big to fit on the CPU chip, so it is stored off chip
  - much slower than registers or cache (200 x slower than registers)

The Simple Machine



- 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 is stored off chip
  - much slower than registers or cache (200 x slower than registers)

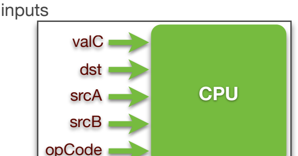
30



- 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

A Program

0: valC=?, dst=?, srcA=?, srcB=?, opCode=?
1: valC=?, dst=?, srcA=?, srcB=?, opCode=?
2: valC=?, dst=?, srcA=?, srcB=?, opCode=?
3: valC=?, dst=?, srcA=?, srcB=?, opCode=?



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)

add rA, rB

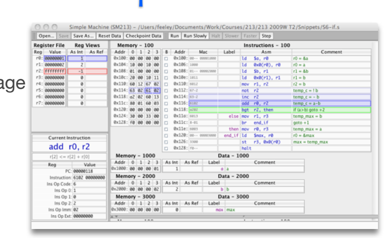
0110	0001	ssss	dddd

add r0, r1

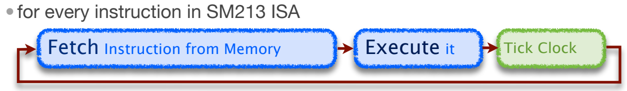
0110	0001	0000	0001

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