Agenda: Monday, June 13

Introductions

Requirements and expectations (http://www.cs.wisc.edu/~cs354-1/)

Chapter 1: Abstractions and Computers

- Introduction
- Computer Architecture
- Basic Computer Operation
- The CPU's Fetch/Execute Cycle

Assignment 0

Introductions:

1. name
2. area of concentration
3. something you like to do in your free time

Requirements and expectations: printouts from course web site

Introduction

Computer architecture- interface between a computer's hardware and its software

Computer hardware- collection of physical elements that make up the machine and its related pieces

i.e. circuit boards, chips, wires, disk drives, keyboards, monitors, printers

Computer software- collection of programs that provide the instructions that a computer carries out

i.e. Word, iTunes, emacs, mozilla

Basic concepts used in Computer Science for designing programs and computers systems:

- **levels of abstraction**
- **hierarchy**
- **models**

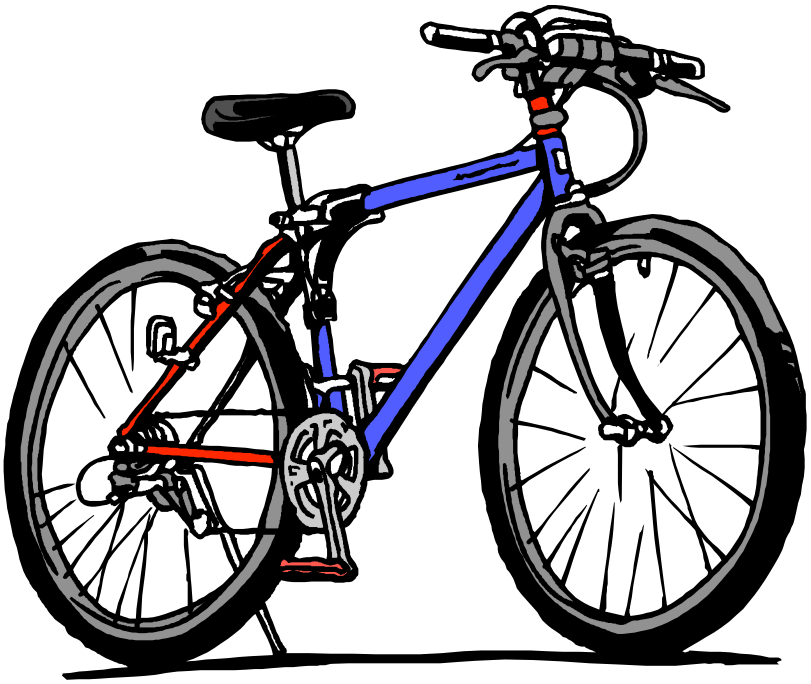
Abstraction

Idea: Abstractions reduce complexity by hiding "unimportant" details.

- Solutions are designed focusing on the important part of the problem.
- Details come "later".
- Reduces overall cost (in time and money) of the solution, since each part is easier to understand and solve.

Example.

This picture is an abstraction that shows some details about a bicycle.



A **hierarchy** (level) **of abstraction** focuses on each detail:

- Initially you see an object to take you places
- Looking one level down, you see: wheels, frame, seat, pedals, handlebars
- Expanding the wheels, you see: rubber tires, metal spokes, …etc

---

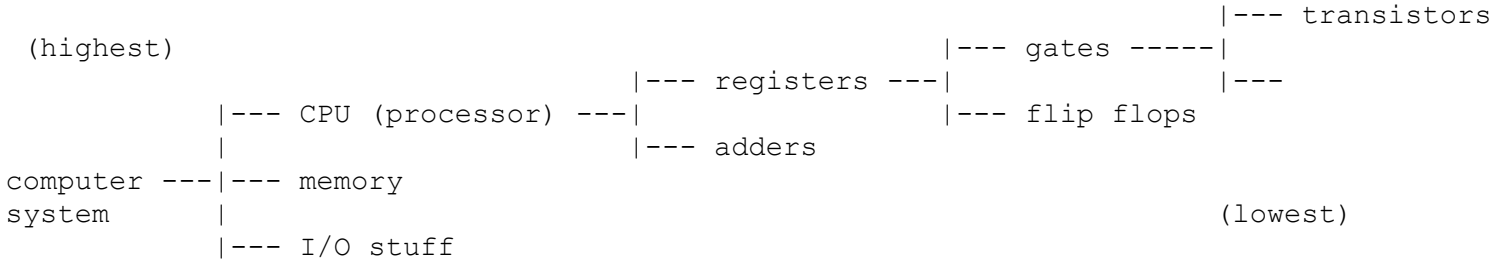
```
We use the same idea to design programs and computers

     When a problem is large, it needs to be broken down --

We "divide and conquer"- breaking large problems (programs) into smaller
problems (that are easier to solve).

One way is by introducing a hierarchy (level), and solving
the problem at each level
```

Example

The design of a computer can be divided into the following hierarchy. (partially shown)

1. transistors (lowest level of the hierarchy)
2. gates and flip flops can be made from transistors
3. components, like registers and adders
4. CPU, memory system, I/O devices
5. computer system (highest level)

In this way, more complex problems are solved using less complex parts.



We can use a "Top down" or a "Bottom up" approach when designing solutions.

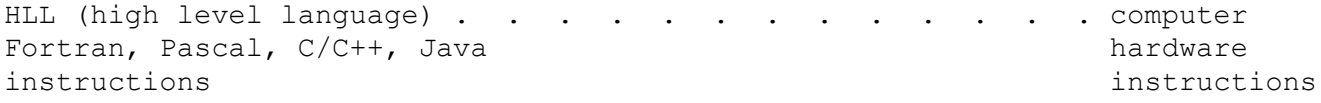
Approach: "Top down"

Goal: Writing a large program

1. divide into logical parts (modules) and design each module separately
2. each module is then divided further into functions and procedures
3. specify the interface for each procedure

   Note: *The implementation of the procedure is a different level of abstraction then the interface specification.*

A different abstraction hierarchy can be used to describe a computer with software running on it.



Problem

- The computer hardware can only execute the set of instructions that it has been designed to execute.
  This is known as its **machine language**.
- However, programmer's want to write programs in HLL (i.e. C#, Java)

What is Machine Language?
a binary sequence (1's and 0's) that can be interpreted by a computer as instructions.
It is not very readable by humans.

(see ch1machineLang)

Solution

- **Develop a language (called assembly language) that is easier for us to read.**
- **Assembly language programs are easier to read because they use "mnemonics" and operands instead of 1's and 0's.**
- **But programs written in assembly lanquage need to be translated into the machine's language.**
  This translator program is called an ASSEMBLER.

Example

```
sub a, b, c
```

"sub" is the mnemonic (or *opcode*) for the addition operation

"a", "b" and "c" are the operands.

Now our computer language abstraction hierarchy looks like this.

```
HLL (high level language) .  .  .  .  assembly  --> ASSEMBLER --> computer
Fortran, Pascal, C/C++, Java          language                     hardware
instructions                          instructions                 instructions
```

How do we get from a HLL to assembly language?

Write a program that translates HLL into assembly language.

i.e. javac is the compiler for java

This program is called a COMPILER.

The complete programming language abstraction hierarchy

```
HLL (high level language) --> COMPILER --> assembly  --> ASSEMBLER --> computer
Fortran, Pascal, C/C++, Java               language                    hardware
instructions                               instructions                instructions
(most abstract)                                                        (least abstract)
(top level)                                                            (bottom level)
```

Programming language sample from Prof. James Larus

Levels of abstraction within Programming Language Hierarchy

HLL	-->	Assembly Language	-->	Machine Language
sum.c		sum.s, sum.nolabels		sum.machine_lang

---
sum.c

```
#include <stdio.h&gt>

int
main (int argc, char *argv[])
{
  int i;
  int sum = 0;

  for (i = 0; i <= 100; i++) sum += i * i;
  printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

---
sum.s

```
-----------------------------------------
      .text
      .align      2
      .globl      main
      .ent  main 2
main:
      subu  $sp, 32
      sw    $31, 20($sp)
      sd    $4, 32($sp)
      sw    $0, 24($sp)
      sw    $0, 28($sp)
loop:
      lw    $14, 28($sp)
      mul   $15, $14, $14
      lw    $24, 24($sp)
      addu  $25, $24, $15
      sw    $25, 24($sp)
      addu  $8, $14, 1
      sw    $8, 28($sp)
      ble   $8, 100, loop
      la    $4, str
      lw    $5, 24($sp)
      jal   printf
      move  $2, $0
      lw    $31, 20($sp)
      addu  $sp, 32
      j     $31
      .end  main

      .data
      .align      0
str:
      .asciiz     "The sum from 0 .. 100 is %d\n"

-----------------------------------------
sum.nolabels
-----------------------------------------

addiu sp,sp,-32
sw    ra,20(sp)
sw    a0,32(sp)
sw    a1,36(sp)
sw    zero,24(sp)
sw    zero,28(sp)
lw    t6,28(sp)
lw    t8,24(sp)
multu t6,t6
addiu t0,t6,1
slti  at,t0,101
sw    t0,28(sp)
mflo  t7
addu  t9,t8,t7
bne   at,zero,-9
sw    t9,24(sp)
lui   a0,4096
lw    a1,24(sp)
jal   1048812
```

```
addiu a0,a0,1072
lw    ra,20(sp)
addiu sp,sp,32
jr    ra
move  v0,zero
```

```
-----------------------------------------
sum.machine_lang
-----------------------------------------

00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

Focus of Course

- software (programming) aspects of assembly language, assemblers and machine language.
- hardware (execution cycle) aspects of computer operation.

This an introduction to the study of *Computer Architechure*.

What is Computer Architecture?

- **the interface between hardware and software**
- **the relationship between hardware and software**

Computers can be designed that directly execute programs in any programming language. For example, a computer that directly executes C/C++ programs. The input to the computer is C/C++ source code.

Why don't we do that?

- **hardware that executes HLL programs directly are slower than hardware that executes a more simple, basic set of instructions.**
- **a computer that executes only C/C++ programs is only useful to those who want to work only in C/C++.**

Any HLL can be translated (compiled) into assembly language.

This course uses a program (*simulator*) to simulate a machine that can execute directly any program that we (you) write.

Programming language abstractions used in this course

```
HLL --> SAL --> MAL --> TAL
```

HLL- High Level Language
SAL- Simple Abstract Language
MAL- MIPS Assembly Language
TAL- True Assembly Language

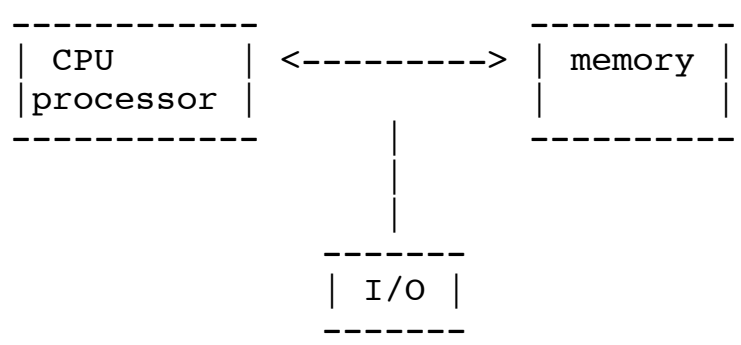
Assumption: you know at least one (maybe several) HLLs

> we will use HLLs code fragments to help you learn the various assembly language abstractions of the course. You will write programs in both SAL and MAL.

HLLs, SAL and MAL are each abstractions.
Each of these abstractions defines a computer architecture.
TAL is a real (manufactured) architecture.

Basic Computer Operation

simplified diagram of a computer system (hardware!)



CPU (Central Processing Unit)

- controls the running (execution) of programs
- executes instructions
- makes requests of the memory
- CPU and processor are synonyms

Memory

- stores programs
- stores data (variables)
- handles requests from the CPU

The STORED PROGRAM COMPUTER concept

We use a memory to save our program, instead of actually setting up the hardware (buttons/switches/pluggable wires) to know what to do.

This was a big deal in the 1940s! It allowed programs to be permanently saved on other media, and then used I/O devices to place the program in the computer memory.

How do the CPU and memory interact with each other?

- The CPU makes requests of memory
- memory does something based on that request

There are three ways that the **CPU** interacts with memory.

1. **FETCH- CPU requests a new instruction to execute**
2. **LOAD- CPU requests the current values of operands**
3. **STORE- CPU requests a store of a result (operand) for use later**

However, **memory** really only has two actions.

1. **READ: memory returns a value to the CPU in response (FETCH or LOAD)**
2. **WRITE: memory overwrite the specified address with new information (STORE)**

How does memory know which data to return for a read request?
The CPU specifies it with an address.

The Fetch/Execute Cycle

How do [most] processors execute instructions?

A simple assembly language instruction for discussion

```
mult aa, bb, cc
```

- **`mult`** **is the opcode (mnemonic) for a multiplication operation**
- **`aa, bb, cc`** **is the list of operands (variables) required for the operation**

Instructions and their operands are stored in memory (in a special format - machine code)

Before they can be used by the CPU, they must be fetched/loaded from memory.

Steps that the CPU takes to execute an instruction

	Operation	*term*
1	Get the instruction from memory. ***Which instruction, what address?***	FETCH INSTRUCTION
2	Interpret the instruction. ***This is a multiplication instruction.***	DECODE INSTRUCTION
3	Get the source operands from memory. ***Source operands are `aa` and `bb`.***	LOAD OPERANDS
4	Complete the operation specified by the instruction. ***`bb` is multiplied by `cc`.***	EXECUTE OPERATION
5	Store the result of the operation in the destination operand. ***The result of the multiplication is stored in `aa`.***	STORE RESULTS

What is a program?

- a collection of variables
- a set of two basic types of instructions
  1. ***instructions that act on those variables (i.e mult, add, div)***
  2. ***instructions that control the order of execution (i.e. mv, jump, loops)***

How do we execute that set of instructions?
We just fetch the next instruction and repeat the execution steps.

But, what's the "next" instruction?
Except for control instructions, we execute instructions sequentially in the order stored.

So, how does the CPU keep track of the next instruction?
It maintains an extra variable called the Program Counter or PC.
The PC contains the address of the next instruction to execute.
Intel calls their version, the IP or Instruction Pointer.

To accomplish this, we must add a step to our list. At some point after each instruction has been fetched, we must update the PC to point to the next instruction.

Revised list of steps that the CPU takes to execute an instruction

1	Get the instruction from memory.	FETCH INSTRUCTION
2	Change the value of the Program Counter. ***This is the address of the next instruction.***	UPDATE PC
3	Interpret the instruction.	DECODE INSTRUCTION
4	Get the operands from memory.	LOAD OPERANDS
5	Complete the operation specified by the instruction.	EXECUTE OPERATION
6	Store the result of the operation.	STORE RESULTS

Control instructions must be considered differently from other instructions

control instruction- any instruction that may modify the contents of the PC to the address of an instruction other than the next stored instruction.

How does the execution cycle change for control instructions?
The final step does not store a result, but instead will modify the PC to the address of the next instruction as determined by the current instruction.

Example: A simple assembly language control instruction

```
beq x, y, label
```

- **`beq`** **is the opcode (mnemonic) for a "branch if equal" instruction**
- **`x, y, label`** **is the list of operands required for the operation**

Steps that the CPU takes to execute a CONTROL instruction

1	Get the instruction from memory.	FETCH INSTRUCTION
2	Change the value of the Program Counter. ***This is always done, since we don't even know that this is control instruction.***	UPDATE PC
3	Interpret the instruction. ***This is a "branch if equal" instruction.***	DECODE INSTRUCTION
4	Get the source operands from memory. ***Source operands are `x` and `y`.***	LOAD OPERANDS
5	Complete the operation specified by the instruction. ***`x` is compared for equality with `y`.***	EXECUTE OPERATION
6	If test is true, overwrite the PC with address implied by 3rd operand (label). ***If (`x == y`), the PC is overwritten.***	STORE RESULTS

How does the computer start the execution of a program?
The Program Counter must be initialized to the address of the first instruction of the program.

How does a program end?
The CPU continues fetching and executing instructions repeatedly until it reaches a special instruction that indicates the end of the program.

This cycle of steps is so important that it has a name. It is called the ... **Instruction Fetch / Execute Cycle.**

There are two (at least) different methods of defining control instructions.

1. decision based on comparison of operands

- as described above, is used by **MIPS**, the architecture that we will study in this course, and the **ALPHA** architecture

- Operands (included in the instruction) are compared. Based on that comparison, the PC may be changed.

2. decision based on setting of condition codes

- used by some architectures, like **Intel (Pentium)** and **SPARC**

Condition Codes

"condition codes"- maintained by the CPU, similar to the way the Program Counter is used to keep track of the next instruction.

condition codes- set by special instructions based on whether the result of the operation was positive, negative, zero or caused overflow.

The condition codes are then used by the control instruction being executed.
The PC is updated depending on the value of the condition codes.

```
An example of an instruction (invented) might be
       bpos  label
  This instruction would overwrite the PC with the address implied
  by the operand label if the Condition Codes indicated a positive
  result.

Note- the ordering of instructions (so that the intended one
  sets the Condition Codes) becomes important with this method.
```