Slide 1

Administrivia

- One purpose of the syllabus is to spell out policies (next slides).
- Most other information will be on the Web, either on my home page (here, office hours) or the course Web page (here).

  A request: If you spot something wrong with course material on the Web, please let me know!

Slide 2

Course FAQ

- "What will my grade be based on?" (See syllabus.)
- "When are the exams?" (See syllabus.)
- "What happens if I can't turn in work on time, or I miss a class?" (See syllabus.)
- "What's your policy on collaboration?" (See syllabus.)

Slide 3

Course FAQ, Continued

- "When is the next homework due?" (See "Lecture topics and assignments" page.)
- "When are your office hours?" (See my home page.)

  Note that part of my job is to answer your questions outside class, so if you need help, please ask! in person or by e-mail.

Slide 4

Course FAQ, Continued

- "What computer(s) can I use to do programming homework?"

  Easiest option may be department's Linux classroom/lab machines. You should have physical access (via your TigerCard) to all the classrooms and labs. You should also be able to log in remotely to any that are booted into Linux, or to a cluster of Linux-only machines in ITS's server room (names `diasnn`, where `nn` ranges from 01 to 05).

Slide 5

"Why Do I Have To Take This Course?"

- We could view computer systems (hardware/software) in terms of layers of abstraction:
  - User interface.
  - Operating system / application programs / tools (compilers, e.g.).
  - High-level programming language / ADTs.
  - Machine language / data representations ("it's all 1s and 0s").
  - Hardware (could break this down, maybe, into logical design and EE).
- A goal of a CS degree program is to "demystify" as many of these as we can.

Slide 6

"Why Do I Have To Take This Course?", Continued

- Relating courses to layers of abstraction:
  - Programming courses — bridge gap between user interface and high-level languages.
  - Operating systems course — bridge gap between user interface / applications programs and hardware.
  - Course on compilers — bridge gap between application programs and machine language.
  - This course — bridge gaps between application programs and machine language (a bit) and between machine language and hardware.

Slide 7

Course Topics

- An overview of how hardware is structured logically and how hardware and software are related.
- A little about defining and measuring performance.
- Assembly language (MIPS because it's simple and representative).
- Machine language (also MIPS).
- Hardware (at level of AND/OR gates).

Slide 8

Why Study Assembly / Machine Language?

- Understand the general principles of how things work at this level helps you:
  - Write more efficient programs.
  - Understand operating systems (which also helps you write more efficient programs).
  - Generally understand better what's really happening in the machine.
- It might be fun?

Slide 9

Introduction

- "Computers are everywhere" — you know about servers and desktops and smaller computing devices, all of which are more and more central to our lives, but also consider "embedded processors", largely invisible but even more prevalent.
- It seems to be a truism that however fast computers can process information, they can't keep up with humans' ability to imagine things for them to do. So performance matters.
- We'll start with an overview of hardware and software and how they interact (cf. textbook subtitle) and also talk a little about measuring performance.

Slide 10

"Below Your Program" — Review?

- Most programming these days is done in a high-level language (HLL), often using a lot of library code. Processors, however, can't execute it directly. How do we get from HLL to something the processor can do?
- First step is to *compile* — conceptually, to *assembly language* (symbolic representation of instructions the processor can execute).
- Next step is to translate assembly language into *machine language* (actual instructions for processors, in 1s and 0s), a.k.a. *object code*. Might be combined with compiling.

Slide 11

"Below Your Program", Continued

- Final step is to combine object code for your program with library object code. Can be done as part of compiling process to create an *executable file* or at runtime, or some combination of the two.
- Actual execution of program typically involves operating system (something manages physical resources / provides abstraction for applications). Contents/format of executable files depends on operating system as well as hardware.
- Worth noting that some languages/implementations don't exactly follow this scheme — some languages (e.g., shell scripts) are translated/interpreted at runtime, and others (e.g., Scala and Java) are compiled to machine language for a virtual processor (the JVM), which may then be translated into "native code" at runtime.

Slide 12

Hardware Components — An Abstract View

- Input devices — way to get info into computer from outside. Examples include keyboard and . . . ?
- Output devices — way to get information back to outside world. Examples include display and . . . ?
- Processor — "brain" that does actual calculations, etc. Can divide into
  - *Datapath* — stores values, performs operations (e.g., addition).
  - *Control* — "puppet master" for datapath.
- Memory — stores values, "scratch pad" for calculations. Now typically includes "main memory" and "cache memory" (possibly multiple levels).

Slide 13

Hardware Components, Continued

- Other noteworthy components (really I/O devices):
  - Storage devices (e.g., disk). (Often this is also called "memory", especially by the nontechnical but increasingly by everyone. But in this course, no.)
  - Network interfaces.

Slide 14

"Layers of Abstraction" Idea

- Idea of "layers of abstraction" used over and over in CS.
- In software, you know how this works.

  Example — "shopping cart" abstraction, implemented using "resizable array" abstraction, implemented using "linked list" abstraction . . .

  Goal — "manage complexity" by dividing big complicated problem into manageable parts.
- Same idea can be used in hardware design, for the same reason . . .

Slide 15

"Layers of Abstraction" Idea in Hardware

- *Instruction set architecture* (ISA or architecture) — a definition/specification of how the hardware behaves, detailed enough for programming at assembly-language level.

  E.g, "x86 architecture", "MIPS architecture", "IBM 360 architecture".

- *Implementations of an architecture* — actual hardware that behaves as defined. Can have many implementations of an architecture, allowing the same program executable to run on (somewhat) different hardware systems.

  E.g., Intel chips, IBM 360 family of processors.

Slide 16

"Layers of Abstraction" Idea in Hardware, Continued

- For programs that will run on a computer with an operating system, also define *application binary interface* (ABI) that describes application's interface with both hardware and operating system.

Slide 17

Minute Essay

- (Most lectures will end with a "minute essay" — as a quick check on your understanding, a way for me to get some information, etc., and also to track attendance. Just put your answer in the body of the message; no Word documents please, and put "minute essay" and the course in the Subject line.)
- Tell me about your background: What programming classes have you taken (at Trinity or elsewhere)? What programming languages are you reasonably comfortable with?
- What are your goals for this course? Anything else you want to tell me?