Slide 1

Administrivia

- Any questions about the syllabus or the course?

Slide 2

Introduction

- "Computers are everywhere" — you know about desktops and servers, 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.
- Factors that affect performance include both the ones you learn about in programming courses (order of magnitude of algorithms, e.g.) and lower-level ones (how well the compiler can map HLL onto hardware in various respects, how fast the hardware can execute instructions).

Slide 3

"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 4

"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., Java) are compiled to machine language for a virtual processor (the JVM), which may then be translated into "native code" at runtime.

Slide 5

Hardware Components — An Abstract View

- Input devices — way to get info into computer from outside. Examples include mouse 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 6

Hardware Components, Continued

- Other noteworthy components (really I/O devices):
  - Storage devices (e.g., disk).
  - Network interfaces.

Slide 7

"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 8

"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 9

"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 10

A Little About Integrated Circuits

- Conceptual view of hardware:
  - *Transistor* — on/off switch controlled by electrical current.
  - Combine/connect a lot of transistors to get *circuit* that does interesting things (e.g., addition).
  - Put a bunch of circuits together to get a *chip* / *integrated circuit* (IC). If lots of transistors, *VLSI chip*.

Slide 11

A Little About Integrated Circuits, Continued

- Manufacturing process starts with a thin flat piece of silicon, adds metal and other stuff to make wires, insulators, transistors, etc.
- Of course, this is all automated! Low-level chip designers use CAD-type tools, which save designs in a standard format, which the chip designers simulate/test with other software, and then send off to be *fabricated*.
- Typically make many chips on a *wafer*, discard those with defects, bond each good one to something larger with *pins* to allow connections to other parts of computer.

Slide 12

Parallelism

- Executive-level definition of "parallelism" might be "doing more than one thing at a time". In that sense, it's been used in processors for a very long time, via *pipelining* and (in high-performance processors) *vector processing*.
- For a (relatively!) long time, hardware designers were able to make single processors faster using these and other techniques (e.g., reducing sizes of things). In the mid-2000s, however, they ran out of ways to do that. But they could still put larger numbers of transistors on the chip. How to use that to get better performance?

Slide 13

Parallelism, Continued

- All that time there were people saying we would hit a limit on single-processor performance, and the only answer would be paralleism at a higher level — executing multiple instruction streams at the same time.
- So . . . use all those transistors to put multiple *cores* (processing elements) on a chip!
- Why wasn't this done even earlier? because alas the "magic parallelizing compiler" —- the one that would magically turn "sequential" programs into "parallel" versions — has proved elusive, and (re)training programmers is not trivial.

Slide 14

Defining Performance

- What does it mean to say that computer A "has better performance than" computer B?
- Really — "it depends". Some answers:
  - Computer A has better response time / smaller execution time.
  - Computer A has higher throughput.
- We'll use execution time, and say

$$\frac{\text{Performance}_A}{\text{Performance}_B} = n$$

exactly when

$$\frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Slide 15

Measuring Performance

- If we use execution time as criterion, how to measure?
- Wall-clock time seems fairest, since it includes
  - Time for CPU to execute instructions.
  - Any waiting for memory access.
  - Any waiting for I/O.
  - Any waiting for operating system.
- Is that easy to measure reliably / repeatably?

Slide 16

Measuring Performance, Continued

- No — to get repeatable measure of wall clock time, need an otherwise unused system.
- So instead we could use “CPU performance” — amount of time CPU needs to run program. Easier to measure, more consistent.
- Or we could try “clock speed”. Can define in terms of “clock period / cycle” or “clock rate” (inverse of clock period).
- Example — for 1GHz processor, what’s its clock cycle?

Slide 17

How These Metrics Relate

- CPU execution time for program X is given by

$$\text{CPU cycles} \times \text{clock cycle}$$

- How would you write that using clock rate instead of clock cycle?
- How would you write it if you know number of instructions and (average) number of cycles per instruction?
- What if you can define different classes of instructions, each with a different number of cycles per instruction?
- So, to double performance for a program, is it enough to double the clock rate?

Slide 18

How These Metrics Relate, Continued

- Not necessarily —
  - Could number of instructions change?
  - Could cycles per instruction change?
- Well, but at least it's better to have fewer instructions?

Slide 19

How These Metrics Relate, Continued

- Also not necessarily — e. g., if you replace instructions that take a few cycles each with a few that take a lot of cycles.

Slide 20

Evaluating / Comparing Performance

- Trickier than it sounds to come up with one number that means something.
- Approaches include
  - Use the actual workload, on the actual hardware platform(s), and compare times.
  - Put together a representative simulated workload — “benchmark”; run and compare times.
  - Compare code size.
  - Compare number of instructions per second (“MIPS” or “MFLOPS”).
- Alas, all of these are flawed in some way.

  (Paraphrasing someone whose name I don’t remember, “peak MIPS is just the number you can’t go any faster than.”)

Minute Essay

- The textbook uses cell phones as an example of something that uses an embedded processor. (I think they do not mean to include “smart phones”, which seem more similar to general-purpose computers.) What are some other devices or products you think probably use embedded processors?

Slide 21