Slide 1

Administrivia

- Reminder: Reading quiz due Monday.
- Homework 1 posted; due a week from Monday (but I say do it earlier if you can).

Slide 2

Recap/Review

- Last time I briefly discussed the first few sections of Chapter 1, trying to focus on what parts you should read and what parts can be skimmed.
- Expanding on that a bit . . .

Slide 3

Abstraction

- Idea of abstraction used over and over in CS, Goal is often to "manage complexity" by dividing big complicated problem into manageable parts. Layered abstractions especially useful for that.
- Software example: If designing an online-shopping application, you might design in terms of a rather abstract "shopping cart", and think later or separately about how to implement the abstraction (e.g., with a collection data type). Details of implementation could be changed without affecting top-level design.
- Same idea can be used in hardware, for the same reasons.

Slide 4

Abstraction in Hardware — ISA

- *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 5

Abstraction in Hardware — ABI

- "Application Binary Interface" (ABI) is a somewhat broader term.
- Includes ISA and and other details of how programs are translated into something the computer can execute, how they interact with their environment. (A bit more about this later.)

Slide 6

Compiling and Executing Programs — Recap/Review

- Several ways source code can be executed:
- Interpreted directly (e.g., shell scripts).
- Compiled to intermediate form, interpreted/executed by possibly-language-specific runtime system (e.g., Scala and Java).
- Compiled to "native code" (machine language), usually producing "executable", and executed. (We will focus on this one — referred to as "compiling to native code".)

Slide 7

Running Executable Files — Overview

- What a processing element can do is fetch machine-language instructions from memory (RAM) and execute them, one at a time. That's it! (Caveat: Conceptually this is what's going on, though current processors include performance enhancements that mean it's something of a simplification. Good enough for now!)
- So to execute a program: Somehow get machine-language instructions into memory and transfer control to a starting instruction.
- Most (not all, but most!) platforms involve an operating system. which reads executable file from storage device into memory and transfers control to its first instruction.

Slide 8

From Programs to Execution — Compiling, Assembling

- Source code translated into assembly language (symbolic representation of machine language) via a compiler. Compilers can be quite complicated, especially if goal is code that's not only correct but also efficient. Worth noting that all compilers for a platform generally follow some conventions that make it easy for subprograms in different languages to call each other. Details are part of ABI.
- Assembly language then converted to object code (machine language, plus other information) via an assembler. Assemblers are much simpler!
- "Other information" in object code includes such details as information about calls to library functions. Format of object code is part of ABI.

Slide 9

From Programs to Execution — Linking

- Linker combines object code from multiple sources, including libraries, to form an executable file, which also consists of machine language plus other information. In static linking, resulting machine language includes all library code. In dynamic linking, some references to library code may get turned into something that gets resolved when the program is started. (More about this later.)
- Executable file's "other information" includes program size, location of starting instruction information about any references to library code not included in the executable. Format of this file also part of ABI.

Slide 10

From Programs to Execution — Loading/Executing

- At runtime, operating system loads machine language from executable file, resolves any calls to dynamically-linked library code, transfers control to starting instruction.
- At that point, processor is executing machine language for program. Typically application programs not allowed to do certain things (e.g., I/O) directly; instead they make requests of operating system. Details of how they do that and what services are available are also part of the ABI.

Slide 11

A Little About Integrated Circuits — Conceptual View

- *Transistor* — on/off switch controlled by electrical current. (Number of transistors is what Moore's law says doubles . . . Well, it used to be every two years but pace seems to be slowing.)
- Combine/connect a lot of transistors to get *circuit* that does interesting things (e.g., addition). At least conceptually, circuits are built up from "logic gates" — simplest are NOT, AND, and OR, pretty much same as Boolean algebra.
- Put a bunch of circuits together to get a *chip* / *integrated circuit* (IC). If lots of transistors, *VLSI chip*.

Slide 12

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*. (These days, at least some design is done more or less by programming, using a notation that describes what the circuit does.)
- 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 13

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.
- Trickier than it might seem to come up with one number that means something!

Slide 14

Evaluating / Comparing Performance — Approaches

- 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", once).

Slide 15

Evaluating / Comparing Performance, Continued

- Alas, all the methods just mentioned are flawed in some way.

  (In particular, paraphrasing someone whose name I don't remember, "peak MIPS is just the number you can't go any faster than.")

- Textbook chooses to focus in this chapter on "execution time". Might not be meaningful for comparing systems but seems like reasonable way to compare processors at least.

Slide 16

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 17

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, and at least says *something* about the processor.
- Even that, though, is not as simple as it might seem.

Slide 18

Defining Performance

- Textbook chooses to focus on CPU (processor) time, and say

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

exactly when

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

(Key equation!)

Slide 19

Sidebar: Clocking and Cycles

- Circuits in typical chip are "clocked" — all parts kept in synch by something that ticks so many times per second. Each tick is a "clock cycle". Each instruction takes one or more cycles. More about this later.
- Clock frequency typically expressed (these days) in gigahertz (GHz, 10^9 ticks per second).

Slide 20

Calculating Program Execution Time (CPU Only)

- CPU execution time for program X is given by

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

  and then CPU cycles in turn is the product of count of instructions and cycles per instruction.
- And then it *might* seem like we can say something meaningful about what happens if we change one of these numbers — but only if all other things remain the same, which might or might not be true!

Slide 21

Calculating Program Execution Time, Continued

- Starting from the basic equation

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

  we can expand a bit to get

$$\text{instruction count} \times \text{cycles per instruction} \times \text{clock cycle}$$

- We can then come up with many variations — e.g., one that uses clock rate rather than clock cycle time — based largely on consideration of units of measure (e.g., clock cycle time is seconds per cycle, while clock rate is cycles per second).

Slide 22

Calculating Execution Time — Example

- Given the following about some program P:
  - On computer A, execution requires 2×10^9 instructions, Instructions take 3 cycles each, and clock rate is 1GHz (so cycle time is $1/10^9$).
  - On computer B, execution requires 1.5×10^9 instructions, instructions take 5 cycles each. and clock rate is also 1GHz.
- Calculate execution times for P . . .

Slide 23

Calculating Execution Time — Example Continued

- Execution times:
  - On computer A, $2 \times 10^9 \times 3 \times 10^{-9}$, i.e., 6
  - On computer B, $1.5 \times 10^9 \times 5 \times 10^{-9}$, i.e., 7.5
- So for P, A's performance is 1.25 times as good as B's (7.5/6).

Slide 24

Sidebar: Dimensional Analysis

- (Or at least I think that's close to the term I want.)
- Idea here is to approach "word problems" in terms of units, treating them almost like factors in multiplication and division. (Example is converting, say, inches to cm by multiplying by 1 in the form 2.54cm/1in.)
- If the formula you propose to use produces the right units (e.g., seconds for execution time), there's at least a good chance it's the right one.

Slide 25

Calculating Execution Time, Continued

- One factor in the basic formula is cycles per instruction. What if that isn't the same for all instructions?
- Common sense(?) may tell you . . .
- If different types of instructions need different numbers of cycles, have to do something like a weighted sum. Usually instructions fall into one of a few "classes", each with a common number of cycles per instruction.
- So, compute times for each "class" of instruction and add. Would also allow you to compute an average CPI.

Slide 26

Calculating Execution Time — Example Continued

- Suppose we change computer A so that there are two "classes" of instructions, a class 2 in which instructions take 2 cycles and a class 4 in which instructions take 4 cycles, and suppose 3/4 of all instructions are class 2 while the other 1/4 are class 4.
- Now execution time is

  $(2 \times 10^9 \times 3/4) \times 2 \times 10^{-9} +$
  $(2 \times 10^9 \times 1/4) \times 4 \times 10^{-9}$

  i.e., 5
- We can also compute average CPI (cycles per instruction) . . .

Slide 27

Calculating Execution Time — Example Continued

- Isn't average CPI just 3? average of 2 and 4?
- One *could* define it that way, but more sensible is to also include information about relative frequencies of the two classes of instructions:
- For a program with N instructions, first compute total number of cycles:
  $((N \times 3/4) \times 2) + ((N \times 1/4) \times 4)$
  $= N \times 2.5$
  and then divide by N to get average CPI of 2.5.

Slide 28

Parallelism (Hardware)

- 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 some high-performance processors) *vector processing*. Some also use *instruction-level parallelism*. All invisible to programmer!
- 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 29

Parallelism (Hardware), Continued

- All that time there were people saying we would hit a limit on single-processor performance, and the only answer would be parallelism 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 30

Parallelism — Hardware

- Several ways to achieve "more than one thing at a time" in hardware:
- Multiple independent processing elements sharing memory (multicore processors, multiple processors).
- "Hyperthreading" — hardware to enable very fast context switching. Not true concurrency but helps with "hiding latency".
- Computers connected by a network.
- Multiple processing elements operating in lockstep (e.g., GPU). For GPU, also involves separate memory, with need to move data back and forth between it and main RAM.
- (Also forms of parallelism that are invisible to programmer.)

Slide 31

Parallelism — Software

- Multithreading — for multicore processors, multiple processors: Single "process" (from operating-system perspective) with multiple "threads" (software streams) interacting via shared single memory space.
- Message-passing — for computers connected by network: Multiple "processes", not sharing memory, interacting by sending each other messages.
- SIMD ("single instruction, multiple data") — for graphics processing units: Single software stream, executing in-effect-simultaneously on all elements of an array (or other coolection?). May require explicit data copying.

Slide 32

Parallelism — Performance

- One use of multithreading is to make the code simpler, at least for the programmer. (Example: typical GUI-based program, where it makes sense to think in terms of one thread of control for getting user input and one for drawing.) Doable on a single processor via interleaving. May improve performance by "hiding latency".
- But it *can* also be used to improve performance. Performance often discussed in terms of "speedup".
- Here, "speedup" is defined thus:

  For P processing elements (cores, fully independent processors, etc.), speedup $S(P)$ is execution time using 1 PE to execution time using P PEs.

Slide 33

Parallel Performance, Continued

- Might seem like with *P* processing elements you could get a speedup of *P*? *But* in fact most if not all programs have at least a few parts that have to be executed sequentially. This limits *S(P)*, and if we can estimate what fraction of the program is sequential we can calculate an upper bound on *S(P)*.
- Further, typically "parallelizing" programs involves adding some sort of overhead for managing and coordinating more than one stream of control.
- But even ignoring those, as long as any part must remain sequential . . .

Slide 34

One More Thing About Performance — Amdahl's Law

- (Named after Gene Amdahl, a key figure in developing some of IBM's early mainframes who left to start his own company to make hardware "plug-compatible" with IBM's. Aside: Interaction between the two companies was — interesting?)
- His observation ("Amdahl's law") can be more generally stated, but in the context of parallel programming it's this:
  If γ is the "serial fraction", speedup on P PEs is (at best, i.e., ignoring overhead)
  $$S(P) = \frac{1}{\gamma + \frac{1-\gamma}{P}}$$
  and as P increase, this approaches $\frac{1}{\gamma}$ — upper bound on speedup.

Slide 35

Minute Essay

- Suppose you are trying to decide which of two computers, call them `Foo` and `Bar`, will give you the best performance. You run two test programs on `Foo` and observe execution times of 10 seconds for one and 20 seconds for the other. If the first program takes 5 seconds on `Bar`, how long does the second program take? (Hint: This might be something of a trick question.)
- Other questions? Be advised that you can ask me anything in a minute essay (preferably about this class or computer science in general), and I'll try to respond.

Slide 36

Minute Essay Answer

- It might seem like that second program would take 10 seconds on `Bar`, but in truth you probably can't be sure without doing the experiment, since the two machines, or the two test programs, could differ in ways that would make this obvious answer wrong.