Slide 1

Administrivia

- Reminder: Homework 7 due Wednesday.
- Reminder: Quiz 5 Wednesday. Roughly same material as homework.

Slide 2

Minute Essay From Last Lecture

- (Review question, answer.)
- Point is that if we want to be able to access an instruction every cycle, we can't also load/store a value unless we have two memories.

Slide 3

Pipelining — Recap/Review

- Could break down each instruction into "phases":
  - Fetch instruction.
  - Read register operands and (at the same time) "decode instruction" (generate control signals).
  - Do operation or address calculation.
  - Access data memory.
  - Write register result.
- "Multi-cycle" implementation would let us skip any steps not needed, but for whatever reason isn't done much these days.
- Instead, widely-used approach sets up "assembly line" (pipeline). Individual instructions don't execute faster, but can get more done in given time, if all goes well (more shortly).

Slide 4

Pipelining — Implementation Overview

- First might observe that the five phases into which we've divided instruction processing seem to map onto the picture of our datapath: What we're doing is breaking up the flow of information through it into steps(!).
- So the idea will be: Somehow partition the datapath so each piece can work on a different instruction. For that to work, we have to add something ("pipeline registers") between pieces that saves results of one step for next step.
- Ignoring complications ("hazards", shortly next slides), this gives what's sketched in Figure 4.35.
- Textbook comments that MIPS ISA was designed for pipelining, and some aspects of the design reflect that (e.g., fixed-size instructions, fields common to all or at least many instruction formats). "Hm!"?

Slide 5

Pipelining — "Hazards"

- Another potential downside to pipelining (in addition to increased complexity): Have to worry about "hazards" — ways in which one instruction might interfere with another.
- Several ways in which things could go wrong . . .
- (Executive-level summary today; more another time maybe.)

Slide 6

Pipelining Complications — "Structural Hazards"

- Idea is that two things we want to do at the same time conflict: E.g., read instruction from memory and read data from memory.
- Only solution is to avoid. For MIPS, we could just stick to separate instruction and data memories.
- (Note that avoiding this problem is why there are three separate things that can add.)

Slide 7

Pipelining Complications — "Control Hazards"

- Idea is that we need to make a decision but can't yet: E.g., can't know what instruction should logically follow a conditional branch until branch instruction is partly executed.
- Several possible solutions:
  - Stall: Just wait until we can be sure.
  - Predict: Make a guess, and if we guess wrong undo/redo.
  - Use delayed branches: Always execute instruction after conditional branch, then jump / don't jump. (This is what MIPS does — meaning that assembler programs we've written don't really represent how things work!)

Slide 8

Pipelining Complications — "Data Hazards"

- Idea is that we need data computed by one instruction before it would normally be available: E.g., two successive R-type instructions, or a load followed by an R-type instruction.
- Several possible solutions:
  - Stall: Just wait until data is available. (Probably not a good solution.)
  - Add hardware for "forwarding": Special hardware to route results to next instruction in addition to regular destination. May or may not be possible.
  - Use delayed loads: Don't allow instruction after "load" to use the result. (This is what original MIPS did.)

Slide 9

Pipelined Implementation — Some Details

- Figures 4.36 through 4.40 show some details of how this implementation works for different groups of instructions. Textbook's notation is that state elements whose right side is highlighted (blue) are being read, and those whose left side is highlighted are being written.
- Note that we now spot a flaw in the design: At the point where we need "which register to write to?", it's no longer correct. Figure 4.41 shows how to correct.

Slide 10

Pipelined Implementation — What's Left

- Need to be explicit about exactly what's needed for those "registers" between stages, but should sort of be common sense(?).
- Need to generate control signals, as in single-cycle implementation. Note that some of them must be saved in those interstage registers. Figure 4.51 shows result.
- Need to deal with data and control hazards. (Structural hazards don't exist for MIPS ISA, assuming we have separate instruction/data memories, as in the single-cycle implementation.)

  Textbook shows many details, interesting but a bit much for this course. But good to get key ideas . . .

Slide 11

Data Hazards — Overview

- Some kinds of data hazards can be addressed by providing additional paths for data to flow ("forwarding"). For others, have to stall the pipeline.

  (Figures 4.53, 4.56.)

- "Stall the pipeline"? can get that effect by not changing registers or memory, and not changing program counter (so in effect the instruction being fetched is fetched again), and/or by inserting a `nop` instruction on the fly.

- Smart compilers can (at least sometimes) avoid stalls by reordering instructions.

Slide 12

Control Hazards — Overview

- Several ways to deal with control hazards:
- Could just stall pipeline. (Apparently not done.)
- Or could implement "delayed branches" — always execute instruction after the branch. (Look at figures and confirm that this will work.) Apparently what MIPS does? (So SPIM not quite accurate implementation of ISA.) Annoying if writing assembly-language programs, but few people do, and compilers can cope?
- Still other ways (used in other architectures?) involve "flushing" in-progress instructions (before they change anything!), possibly combined with various schemes for predicting branch outcome. Details no doubt interesting, but not trivial!

Slide 13

Exceptions

- As in higher-level programming languages, situations at this level where you want to bail out of the normal flow of control because something has gone wrong (e.g., arithmetic overflow).
- Further, situations in which you want to alter normal flow of control to deal with something happening outside processor (e.g., I/O device has finished something you previously asked it to do). (You could check it periodically, yes, but usually that's inefficient.)
- Some architectures distinguish between "exceptions" (first case) and "interrupts" (second case), but all kind of the same thing, so MIPS doesn't; all "exceptions".
- What should happen on exception? Several possibilities . . .

Slide 14

Exceptions, Continued

- Some exceptions errors from which we can't reasonably recover (e.g., program tried to execute something not an instruction).

  What should happen then? probably terminate the offending program.
- Other exceptions errors from which recovery is possible, or things that have nothing to do with currently-running application (e.g., signal from I/O device).

  What should happen then? operating system should do something and then return to interrupted application.
- Exception/interrupt mechanism turns out to also be useful as a way for applications to request operating-system services.

Slide 15

Exceptions — Hardware Versus Software

- Hardware must save current PC (with a caveat) and transfer control to fixed location(s) with an indication of cause of exception.
- Code at fixed location(s) must "do the right thing" for the exception, as described previously. Normally this code is part of operating system.
- Caveat: Pipelining complicates exception processing — must allow instructions prior to the interrupted one to complete, complete or flush the interrupted one, etc. Textbook has (some of) details.

Slide 16

Hardware for Exceptions

- So, on exceptions (any type) need to bypass normal flow of control and branch to — somewhere, and fixed location(s) seems reasonable(?).
- Also need some way of indicating type of exception, plus address of interrupted instruction (in case we need to go back).

Slide 17

Hardware for Exceptions, Continued

- MIPS architecture uses two registers
  - cause of exception ("Cause register")
  - address of interrupted instruction (EPC)

  and always transfers control to same place (where there should be code that's part of operating system).

  (Compare Figures 4.65, 4.66.)

  (Try, in SPIM, a program that forces an exception — `sw` to an invalid address seems to work.)
- Other architectures transfer control to different places depending on type of exception — "vectored interrupts".

Slide 18

Minute Essay

- Had you heard of pipelining? (You may have if you're interested in hardware?) if so, in what context, and how does the discussion in this class fit with what you know?