Administrivia

- Reminder: Homework X1 due today (if you plan to turn it in). Homework 6 due Monday.
- Reminder: Quiz 5 Monday. Topics likely to include something about the simple single-cycle implementation, key ideas of pipelining.

Slide 1

Minute Essay From Last Lecture

- (Review first two questions and answers. I think many people were not quite clear on what was being asked?)
- About Homework 5, two people said they liked it better than any of the others (hm!), and while some found it difficult, some found it the easiest.

Slide 2

Slide 3

Pipelined Implementation — Review/Recap

- Idea is to break up processing of each instruction into several phases/stages, and overlap processing. Textbook compares to laundry room; another widely-used analogy is an assembly line.
- For MIPS architecture (subset), five stages makes sense. To make this all work, logically separate datapath into five pieces and add "registers" (place to save data) between stages as needed, as shown in Figure 4.35. Next few figures show execution of different kinds of instructions and may help this make sense. Also note that we now spot a flaw in the design — Figure 4.41 shows how to correct.
- 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).

Slide 4

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 — and here, need to also add (some of) them to those interstage registers.
  Figure 4.51 shows result.
- Need to deal with data and control hazards. (Structural hazards don't exist for MIPS ISA — well, 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 . . .

Data Hazards — Executive-Level Summary

- Some kinds of data hazards can be addressed by providing additional paths for data to flow ("forwarding"). For others we have to stall the pipeline. (Figures 4.53, 4.58.)
- "Stall the pipeline"? can get that effect by not changing registers or memory, and not changing the program counter (so in effect the instruction being fetched is fetched again), and/or by inserting a `nop` instruction on the fly.

Slide 5

Control Hazards — Executive-Level Summary

- Several ways to deal with control hazards.
- One would be to stall the pipeline (though apparently that isn't done).
- Another way is to implement "delayed branches" — always execute the instruction after the branch. (Look at figures and confirm that this will work.) Apparently this is what MIPS ISA does? (So SPIM isn't quite an accurate implementation of the ISA.) Seems like an annoyance 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 6

Slide 7

Exceptions

- As in higher-level programming languages, there are 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, there are situations in which you want to alter normal flow of control to deal with something happening outside the processor — e.g., an 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 it's all kind of the same thing, so MIPS doesn't — all "exceptions".
- What should happen on exception? Several possibilities . . .

Slide 8

Exceptions, Continued

- Some exceptions are errors from which we can't reasonably recover (e.g., program has tried to execute something not an instruction, or has tried to do something beyond its current level of privilege).

  What should happen then? probably terminate the offending program.
- Other exceptions are errors from which recovery is possible, or things that have nothing to do with the currently-running application.

  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 9

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 10

Hardware for Exceptions

- So, on exceptions (any type) need to bypass the normal flow of control and branch to — somewhere, and fixed location(s) seems reasonable(?).
- Also need some way of indicating what kind of exception we have, plus address of interrupted instruction (in case we need to go back).
- MIPS architecture uses two registers — one to hold cause of exception ("Cause register"), another to hold address of interrupted instruction (EPC), and always transfers control to the same place (where there should be code that's part of the operating system).

  (Compare Figures 4.65, 4.66.)

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

Minute Essay

- Many processors have a notion of two modes of operation, a privileged one for when they're doing operating-system stuff and an unprivileged one for regular applications. Attempts to do privileged things while in unprivileged mode generate exceptions. What if anything can you say about how this might help in making the whole system (hardware plus software) robust and secure? (Speculate!)

Minute Essay Answer

- If regular applications execute in unprivileged mode, the hardware can enforce some restrictions on what they can do (e.g., only request I/O by going through the operating system). How do you get from unprivileged mode to privileged mode then? As part of exception processing — hardware transfers control to fixed location(s) and switches to privileged mode.