Slide 1

Administrivia

- Reminder: Quiz 5 Wednesday. Topics from Appendix C.

Slide 2

Minute Essay From Last Lecture

- No one really came very close.
- Something to be considered is that “real” systems seem not to make this distinction, so there must be some way to design a processor with a single memory to contain both instructions and data!
- Key point is that if we want to do everything in a single cycle — that includes both getting the instruction and potentially getting some data from memory.

Slide 3

Control Logic — Review/Recap

- First step was to sketch a "datapath" — combinational logic blocks to perform needed computation, state elements to save values. Notice that sometimes we need what seem to be redundant logic blocks (e.g., multiple things that can add) — in part because for right now we're trying to do everything in a single cycle, so potentially we need to do several additions concurrently.
- Several parts of the datapath need additional information — "control signals" — that depends on what instruction is being executed. "Control logic" transforms (parts of) instruction into control signals.

Slide 4

Control Logic — A Bit More

- Section 4.4 discusses in some detail how to get from the 32 bits of the instruction (really just the opcode and function fields) to the needed control signals. To some extent it's common sense, with one possible exception . . .
- ALU as designed in Appendix C uses 4 bits to represent which operation is to be done. Seems like it would be simple enough for the main control unit to generate these directly, no? However, turns out to be even simpler to split functionality into two parts — generate a 2-bit "ALU operation" from just the opcode field, and then use that plus (for some instructions) the function field to tell the ALU what to do.

Instruction Execution Details

Slide 5

- Section 4.4 gives some details of what happens for each kind of instruction in the subset (initially omitting jumps). What we need to add for jumps — end of section.
- We won't discuss more in class, but you should read carefully — not to memorize, but to understand.

Multi-Cycle Implementations

Slide 6

- So, we have a sketch for an implementation that executes one instruction per cycle. But clearly this isn't how all real systems work (if nothing else, many don't separate instruction memory from data memory).
- Why not? means cycle time is limited by length of longest path through the whole path, while many instructions can be done faster.
- What to do? break up work into multiple pieces . . .

Slide 7

Instruction Phases

- Work involved in fetching and executing a MIPS instruction can be split into phases:
  - Fetch instruction.
  - Read register operands and (at the same time) decode instruction. "At the same time" because of instruction format(s).
  - Do operation or address calculation.
  - Access data memory.
  - Write register result.
- How does this help? Two possibilities . . .

Slide 8

Simple Multi-Cycle Implementation

- One approach is to stick to the idea of executing one instruction at a time, but break things up so instructions potentially take multiple cycles. (How's *that* going to help? Well . . .)
- Control logic is now going to be more complex — must do everything we were doing before, plus keep track of which phase we're in. (Recall discussion of finite state machines from Appendix C.)
- However, one potential payoff is skipping unused phases (e.g.., the R-format (arithmetic/logic) instructions don't need to access data memory, and indeed we don't need separate instruction/data memories).

Slide 9

Pipelined Implementation

- Another approach is to use “pipelining”: Modeled after assembly line; many real-world analogies possible. Textbook describes a laundry “assembly line”, with stages corresponding to washing, drying, folding, and putting away.
- Could base a pipelined implementation of MIPS on the same phases used for a multi-cycle implementation, with one pipeline stage per phase.
- How does this help? well, it doesn’t make individual instructions faster, but it means you can get more of them done in a given time.
- Like the simple multi-cycle implementation, it means added hardware complexity (next time). Also introduces some new potential problems . . .

Slide 10

Pipelining — “Hazards”

- Another potential downside to pipelining (in addition to increased complexity) is that we have to worry about “hazards” — ways in which one instruction might interfere with another.
- Several ways in which things could go wrong . . .

Slide 11

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 go back to separate instruction and data memories.

Slide 12

Pipelining Complications — “Control Hazards”

- Idea is that we need to make a decision but can’t yet — e.g., we can’t know what instruction should logically follow a conditional branch until we have the branch 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 the assembler programs we’ve written don’t really represent how things work.)

Slide 13

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 a “load” to use the result. (This is what original MIPS did.)

Slide 14

Minute Essay

- None — sign in.