Slide 1

Administrivia

- (By e-mail.)

Slide 2

Minute Essay From Last Lecture

- Most people (though not all) recognized that computing 64-bit values might take longer than computing 32-bit values.

  However, it *does* depend on the instruction: In principle all bits of the answer are computed simultaneously, but in practice results are only available simultaneously if all the bits are independent. True for `and`; not true for `add`.
- (I did mean to exclude operations such as multiplication and division, which don't really fit into the simplified design we're developing.)

Slide 3

Designing a Processor — Recap

- The goal is to sketch out an implementation of a small but (we hope) representative selection of MIPS instructions, consisting of three groups:
  - Memory-access instructions (`lw`, `sw`).
  - Arithmetic/logical instructions (`add`, `sub`, `and`, `or`, `slt`).
  - Control-flow instructions (`beq`, `j`).
- Implementation is in terms of combinational logic blocks and state elements, all ultimately constructed from AND and OR gates and inverters. Note however the frequent use of layers of abstraction.
- To make it possible for state elements to be changed in some controlled way, we use "clocking".

Slide 4

Some Components We Want

- A register file.
- Some memory, which for simplicity we'll separate into instruction memory and data memory. Why? Simplifies some aspects of the design.
- Some way of representing where to find the "next" instruction — a "special purpose" register typically called "program counter" (PC).
- One or more ALUs (why more than one? should become obvious soon).
- "Control logic". (More soon.)
- How does Figure 4.1 relate to what we need to do . . . First a small digression about clocking.

Slide 5

Clocking — Recap/Review

- Hardware will include something that implements a “clock cycle”.
- State elements’ inputs are “sampled” during one phase of this cycle, and outputs change as inputs change. (So, these use the latches of Appendix B.)
- Length of cycle determines how complicated the various logic blocks can be (or vice versa).

Slide 6

Fetching Instructions and Updating PC

- For all instructions, start by getting instruction from memory. (What do we need? How does this map to Figure 4.1?)
- For most instructions, at some point we need to increment PC. (What do we need? How does this map to the figure?)
- And then the three groups of instructions do different things, but there are some commonalities . . .

Slide 7

Memory-Access Instructions

- Instruction includes two registers (one for base address, one for where to load into / store from), 16-bit displacement.
- Needed computation:
  - Add displacement to register containing address.
  - Use result to access memory, loading/storing to/from register containing data.
- How does this map to Figure 4.1?

Slide 8

Arithmetic/Logic Instructions

- Instruction includes three registers (two for input operands, one for result).
- Needed computation:
  - Perform operation (with ALU) using values from two registers as inputs.
  - Save result in target register.
- How does this map to Figure 4.1?

Slide 9

Control-Flow Instructions (`beq`)

- (`j` later.)
- Instruction includes two registers (values to compare), 16-bit displacement used to find target of branch.
- Needed computation:
  - Compare contents of two registers.
  - Compute address of branch target (PC+4 plus displacement).
  - Use result of comparison to choose value for next PC.
- How does this map to Figure 4.1?

Slide 10

Overview Revisited

- Figure 4.1 seems to have ways to do everything we need to do — paths for data to flow from one place to another, including into ALU(s) for computation.
- For every instruction we're in some sense doing the same things (have each ALU compute something), but some results are essentially discarded. (Example — `beq` computes two "next instruction" addresses, but only stores one back into the PC.) This is very typical of how things work at this level!

Slide 11

The "Datapath" — What's Missing

- Inputs to some blocks (e.g. PC) can come from more than one source. *That* can't work. So we need multiplexors to control which is used.
- Inputs to ALU / adder are 32 bits, but for some instructions we want to get one of them from 16 bits in instruction. So, need something to extend that to 32 bits by extending sign.
- Both control-flow instructions include something that must be shifted two bits before being used to compute target address, so need to support that.
- Add these to "datapath" part of Figure 4.1 to get Figure 4.15. Leaves out "control" part, substituting not-connected-yet control inputs (blue in figures.)
- Right now we're showing the whole instruction as input to all elements that need part of it; we'll refine this later.

Slide 12

Control Logic

- So we have a "datapath" that can do things, but there are some inputs that aren't connected to anything. An analogy — the datapath is a puppet, and these inputs are its strings.
- Who/what pulls the strings? the "control logic" — combinational logic whose input is the current instruction plus any other needed information and whose output is those disconnected inputs to datapath.

Slide 13

Control Logic

- Figure 4.16 shows names of "control signals" and what they mean. (Note: Textbook uses "asserted" and "deasserted"; I'll just use 1 and 0 — textbook indicates that this is a bit sloppy but I think okay for our purposes.)

  (Why `MemRead`? textbook says/implies that data memory is constructed such that attempts to read from invalid address could cause problems, and sometimes address *won't* be valid.)
- How to generate them? As mentioned in Appendix B, tools exist to transform truth tables into combinational logic, so it will be enough to come up with a truth table that maps inputs to the needed signals.

Slide 14

Control Logic, Continued

- Figure 4.17 adds needed combinational logic blocks to Figure 4.15.
- Section 4.4 works through details. A lot of it should seem like common sense (viewed from the right angle?). Only potentially tricky part is input to ALU "which operation?" . . .

Slide 15

ALU Control Input

- ALU as designed in Appendix B uses 4 bits to represent which operation is to be done — 2-bit input to multiplexor plus two "inverted input" signals (see Figure B.5.10 and table on p. 259 — e.g., 0010 to add, 0110 to subtract). Seems like it would be simple enough for the main control unit to generate these directly?
- 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 arithmetic instructions) function field to tell the ALU what to do.

Slide 16

Instruction Execution Details — Tracing What Happens

- Tracing through what happens as various instructions are executed is tedious but (I think!) instructive:
- Work from Figure 4.17 (revised/improved version of 4.15) and the tables in Figure 4.18 and Figure 4.13.
- Start out by writing down what you know: Output of PC (its current value), fields of instruction.
- Use figure and tables to fill in other things, tracing through how bits flow.
- What you come up should be consistent with what the instruction is supposed to do.

Slide 17

Instruction Execution Details — Examples

- Example `add`. (Solution online as part of Homework 7 assignment.)
- Example `lw`. (Solution online as part of Homework 7 assignment.)
- Example `beq`. (Solution online as part of Homework 7 assignment.)

Slide 18

Minute Essay

- The design sketched so far has two separate memory blocks, one for instructions and one for data. This turns out to be needed for the simplest implementation, one in which each instruction executes in a single cycle. Why? is there something different about the types of values to be stored, or is there some other reason? (*Hint:* Think about what has to happen for `lw`.)

Slide 19

Minute Essay Answer

- For `lw`, you need to be to both load the instruction and also load something from the specified address. (This is an open-ended version of one of the textbook's "check yourself" questions for section 4.3.)