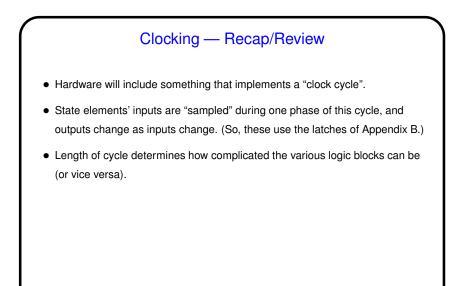


Some Components We Want

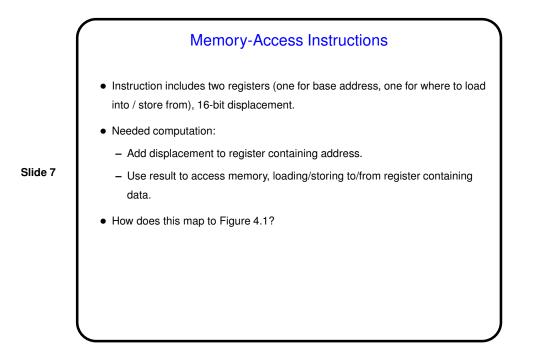
- A register file.
- Some memory, which for simplicity we'll separate into instruction memory and data memory. Why? Simplifies some aspects of the design.
- Slide 4
- Some way of representing where to find the "next" instruction a "special purpose" register typically called "program counter" (PC).
- 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.

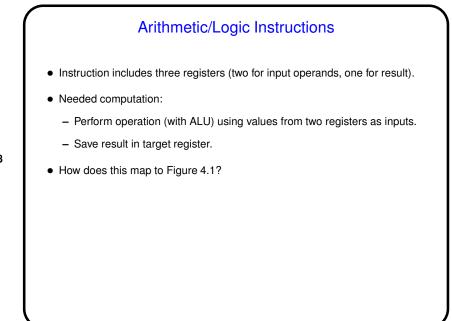


Slide 6

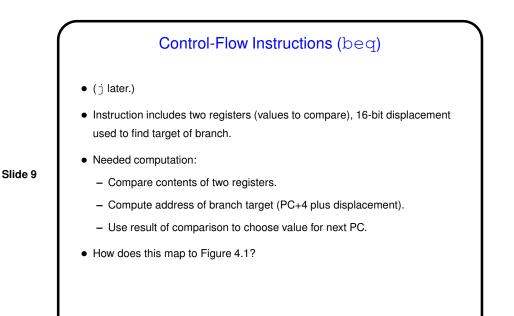
Fetching Instructions and Updating PC For all instructions, start by getting instruction from memory. (What do we need? How does this map to Figure 4.1?) For most instructions, at some point we need to increment PC. (What do we need? How does this map to the figure?) And then the three groups of instructions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

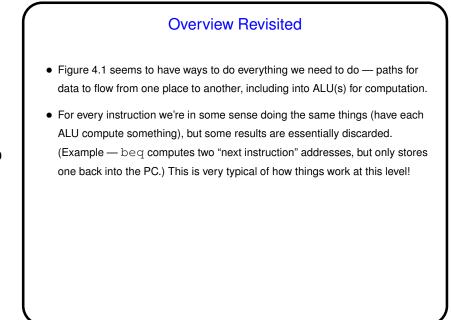
3

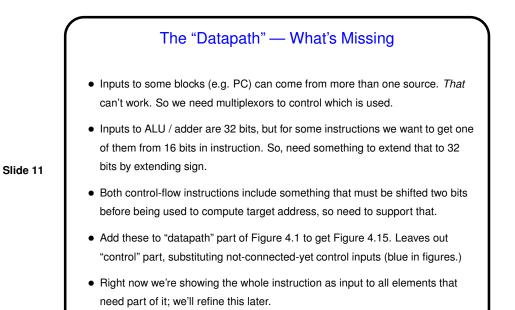




4







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.

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.)

Slide 13

• 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.

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?"...

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?

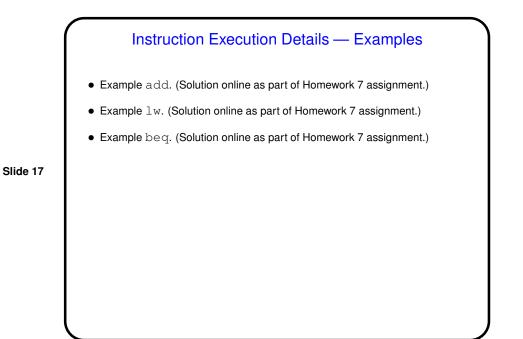
Slide 15

• 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.

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.



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 18

9

