

Digression — Drawing Figures Programmatically
I only had one person express interest in using Lag to draw figures as for Homework 5, but it's kind of cool, so I'll digress here ...
As I was preparing a sample solution for Homework 5 in a previous year, I got interested in whether there wasn't some nice tool to do this programmatically — rather than me drawing a bunch of gates with a drawing program and connecting them, well, it just seemed like something a computer could help a lot with, and similarly with the state machines.
Being a Lag Kanatic, I looked for Lag X-based approaches, and found ...

## Digression — Drawing Figures Programmatically

 ... something called TikZ (short for German for "TikZ is not a drawing program). There's quite a learning curve, but the results can be really nice.
 Examples on "sample programs" page.

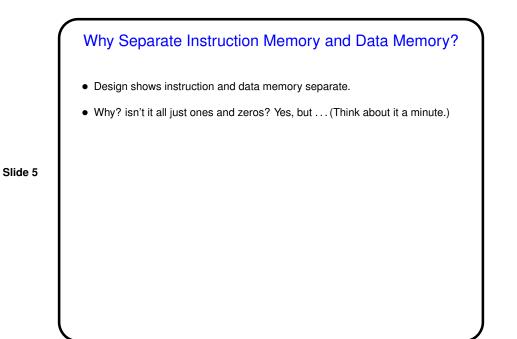
(I got carried away and spent part of my summer drawing some of the figures in Chapter 4 with it! And I *think* it really is easier for me now to produce nice-looking diagrams like the ones in Appendix B.)

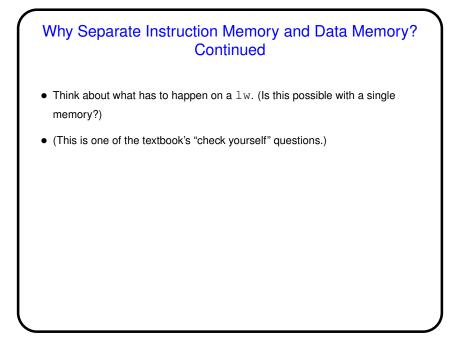
• Take-home message, maybe: LaTEX is really good in general at converting "logical markup" into something more graphical. That this can apply to turning a logical(?) representation of a figure into something graphical — maybe surprising, maybe not? Other tools could work the same way (and maybe some do)?

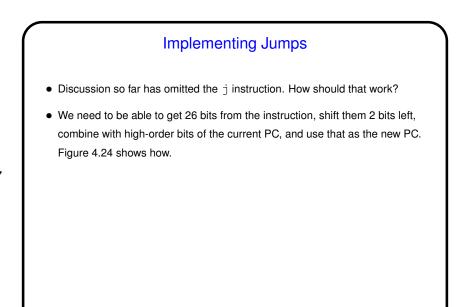
Slide 3

## Designing a Processor — Review/Recap

- So we've sketched the design of a processor that implements a supposedly representative set of instructions.
- A few more things to fill in ...



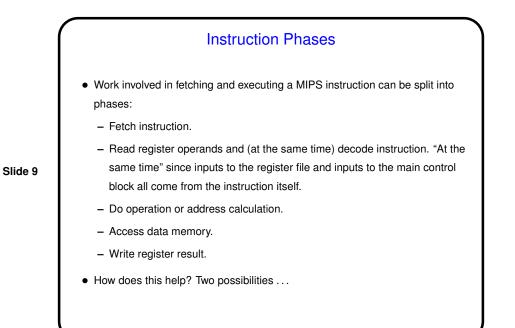




Slide 7

# Multi-Cycle Implementations

- 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 circuit, while many instructions can be done faster.
- What to do? break up work into multiple pieces ....



# 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 becomes 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 B.)



 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.
 (This kind of implementation — remember the discussion back in Chapter 1, in which different instructions took different numbers of cycles?

Slide 11

• A previous edition of the textbook lays out a design for this (details later, maybe).

## **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 ...

