CIS 372
Computer Organization and Design Lab

Prof. Amir Roth

Unit 5: Pipelined LC4 Processor

Lab 1 → Lab 2

- Lab 1: fully-functional LC4 processor, nothing fancy
  - "CIS 240" level of sophistication
  - Single-cycle
- Lab 2: better (more realistic) LC4 processor, slightly fancy
  - "CIS 371" level of sophistication (use what you have learned)
  - 5-stage pipeline, stall/flush logic
  - Simple branch prediction: 8-entry tagged BTB
  - Performance counters
- Will be posted on Monday
  - Get started over break (if you have nothing else to do)

Lab 1 (Almost) Post Mortem

- Comments, problems?
- Any design worked in ModelSim but not on the board?
  - No? That's the beauty of restricted Verilog!
- How many new lines of Verilog did you write?
  - <200?
- How many hours did this lab take?
  - <60 (total for all group members)?
- What was the single nastiest design bug you encountered?

Lab 2

- Much harder than lab 1
  - Students often shocked at how much harder it really is
  - Common refrain: "thrown into the deep end"
  - Why? interaction between instructions
- Lab 1: one instruction in datapath at a time
  - Can test and debug each instruction individually
- Lab 2: up to five instructions in datapath at a time
  - Instructions may work individually, but interactions may be broken
  - Must carefully think about possible interactions ahead of time
  - "Design"
“Design”

- Goal of this lecture
  - Alert/expose you to the many design decisions ahead
  - So you can make them explicitly and holistically
  - Instead of implicitly and greedily

- Try to avoid following scenario
  - Implement quick and dirty pipeline with no bypassing
  - Try to add bypassing, realize original design makes this nasty
  - Start over: right approach going forward, but time consuming
  - Patch current implementation: risky, bugs are born this way
  - Best to think of as much as you can early

Important First Step: Design Document

- Datapath
  - Where are the pipeline registers? Which signals are latched?
- Control
  - Datapath control vs. pipeline control
- Implementation milestones and testing strategy
  - How do you plan to “stage” your design?
- Don’t skimp on this part
  - Time spent now will more than pay for itself in form of less debugging
    - Bugs are basically corner cases / interactions you didn’t think of
    - Try to think of as much as you can ahead of time
  - This is why we are making you do this document

Pipelined LC4

- 5-stage pipeline
  - Fetch, Decode, eXecute, Memory, Writeback
  - Full bypassing: MX (both inputs), WX (both inputs), WM
- One-cycle “load-use” penalty
  - A dependent instruction right after the load
- Branch handling
  - Resolved in “execute” stage, two-cycle penalty
  - Simple branch predictor: 8-entry BTB
- Four memory-mapped performance counters
  - Cycles, instructions, branch stalls, load stalls
  - \#cycles = \#insns + \#branch stalls + \#load stalls

Where To Put The Pipeline Registers?

- But can put them elsewhere if you want
Naming and Register Conventions

- **Naming:** prepend signal names with name of stage
  - Will avoid confusion

```verilog
code
wire [15:0] F_PC, F_INSN;
wire [15:0] D_PC, D_INSN;
```

- **Register:** use single wide register and wire concatenation
  - Will avoid mistakes from differing pipeline control signals

```verilog
code
Nbit_reg #(32) FD_pipeline (.in({F_PC, F_INSN}),
                       .out({D_PC, D_INSN}),
                       .we(FD_WE), .rst(FD_RST),
                       .clk(CLK), .gwe(GWE));
```

Stalling and Flushing

- **Two related features**
  - **Stalling:** for data-dependences you can’t bypass
  - **Flushing:** for mis-predicted branches

- **How are mechanics of stall/flush implemented?**
  - How to “inject a nop” (for stall)?
  - Or “overwrite with nop” (for flush)?
  - May be able to use `we` (write_enable) and `rst` (reset)?
    - Handy: x0000 instruction is nop

```verilog
code
Nbit_reg #(32) FD_pipeline (.in({F_PC, F_INSN}),
                       .out({D_PC, D_INSN}),
                       .we(FD_WE), .rst(FD_RST),
                       .clk(CLK), .gwe(GWE));
```

Datapath Control vs. Pipeline Control

- **Pipeline control:** advancement of insns through pipeline
- **Datapath control:** datapath at each stage
  - Pipeline control controls advancement of datapath control
  - Keep these separated (thank me later)

Datapath Control vs. Bypass Control

- **Bypass control:** bypass datapath muxes
- **Datapath control:** non-bypass datapath muxes
  - Keep these separate too
    - Don’t use giant mux for both bypass and reg/imm or reg/pc select
    - Enforced via external “bypass” switch
Controller vs. Decoder

- **Controller**: generates control signals directly
  - E.g., DMEM_WE, REG_WE
- **Decoder**: generates intermediary signals
  - Which make it easier to generate control locally
  - E.g., IS_STORE, HAS_NZP
- For single-cycle datapath, doesn’t really matter
  - Only have datapath control
- For pipeline, decoder style is easier, why?
  - Need intermediate decode signals for stall/bypass anyway
    - RD, HAS_RD, RS, HAS_RS, RT, HAS_RT
  - Rather than propagating control and decode down pipeline...
  - Propagate just decode and generate control on the fly
  - Makes code more transparent as well (control logic in datapath)

Performance Counters

- Four 16-bit memory-mapped counters
  - Instruction, cycle, branch stall, data stall
  - Cycle = instruction + branch stall + data stall
  - Updated at writeback: cycle + one of the other three
    - How do you know which one?
  - Don’t forget to bypass these
  - Implement these internally to lc4_pipeline
  - Will be used in test-benches to test bypassing, branch prediction

How To Proceed

- Clean up your single-cycle datapath
  - Easier to pipeline a “clean” implementation than something nasty

- Design pipeline to work with bpred and bypassing first
  - Before implementing anything

- Add performance counters immediately, don’t retrofit
  - Will help you debug branch prediction and bypassing

- Use short programs to test pipeline features
  - Instruction sequences that exercise specific bypass paths
  - Instruction sequences that exercise BTB
  - Instruction sequences that manipulate performance counters