Announcements

Project proposals returned
Exams returned
• Average 68 (this is a B)
• If you are below 60 (grad) or 50 (undergrad)
  you should stop by and talk to me!
We are starting with pipelining today (Chapter 3)
Reader 3 coming up on Wednesday
Protection

Goal:
- one process should not interfere with another

Process model
- privileged kernel
- independent user processes

Privileges and policy
- architecture provided primitives
- OS implements policy

Pipelining

Principles of pipelining
Simple pipelining
Structural hazards
Data hazards
Control hazards
Interrupts
Multicycle operations
Pipeline clocking
Principles of Pipelining

Let $t$ be the time to execute an instruction
Instruction execution requires $n$ stages taking $T = \Sigma (t_i)$

Without pipelining
- throughput = $1/T = 1/\Sigma (t_i)$ and latency = $T = 1/$throughput

With $n$-stage pipeline
- throughput = $1/\max(t_i) \leq n/T$
- latency = $n \max(t_i) \geq T = n/$throughput
- speedup $\Sigma (t_i) / \max(t_i) \leq n$
- if all $t_i$ are equal, speedup = $T/t = n$
**Principles of Pipelining (Cont.)**

Let $x$ be extra delay per stage for
- latch overhead
- clock/data skew

$x$ limits the useful pipeline depth

With $n$-stage pipeline (all $t_i$ equal)
- throughput $= \frac{1}{x + t} < \frac{n}{T}$
- latency $= n(x + t) = nx + T$
- speedup $= \frac{T}{x + t} \leq n$
- real pipelines usually do not achieve this due to hazards

---

**Simple Pipelines**

IF-fetch, ID-decode, EX-execute, MEM-memory, WB-writeback

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright © 2001 Falsaï, from Hill, Smith, Sohi, Vijaykumar, and Wood  © Chapter 3 Pipelining
Hazard conditions that lead to incorrect behavior if not fixed

Structural hazards
- two different instructions use same h/w in same cycle

Data hazards
- two different instructions use same storage
- must appear as if the instructions execute in correct order

Control hazards
- one instruction affects which instructions is next
Resolving Hazards

Pipeline interlock logic detects hazards and fixes them
Simple solution: stall - increases CPI, decreases performance
Better solution: partial stall - some instruction stall, others go
Better to stall early than late

Structural Hazards

When two or more different instructions want to use
- same hardware resource in same cycle
- e.g., load and stores use the same memory port as inst fetch
- e.g., multiple integer add insts in the same cycle
Dealing with Structural Hazards

Stall
- low cost
- simple
- decreases IPC
- use for rare case

Pipeline hardware resource
- useful for multicycle resources
- good performance
- sometimes complex e.g., RAM

Dealing with Structural Hazards (Cont.)

Replicate resource
- good performance
- increases cost
- interconnect delay
- use for cheap or divisible resources
Dealing with Structural Hazards

Structural hazards are reduced
- if each instruction uses a resource at most once
- always in same pipeline stage
- for one cycle
- many RISC ISAs designed with the above in mind

Data Hazards

When two different instructions use the same storage location
- it must appear as if they executed in sequential order
  - add --, R2, R1
  - sub R1, --, R2
  - or --, --, R1
- read-after-write (RAW): true dependence, artifact of computation
- write-after-read (WAR): anti dependence, artifact of storage
- write-after-write (WAW): output dependence, artifact of storage
- read-after-read: no hazard
Examples of RAW

```
add --, --, R1          IF ID EX MEM WB
       \                     R1 written
sub --, R1, --          IF ID EX MEM WB
       \                     R1 read  R1 written
Id  [--], R1            IF ID EX MEM WB
sub --, R1, --          IF ID EX MEM WB
       \                     R1 read  Memory written
st  --, [100 + R2]      IF ID EX MEM WB
Id  [100 + R2], --      IF ID EX MEM WB
       \                         Memory read
```

Simple Solution to RAW

```
add --, --, R1          IF ID EX MEM WB
       \                     R1 written
sub --, R1, --          stall stall IF ID EX MEM WB
       \                     R1 read

Assumes register written then read (half cycle ops)

• low cost
• simple
• decreases IPC
• must minimize stalls
```
Minimizing RAW Stalls

Bypass/forward/short-circuit

Data available  R1 written

add --, --, R1   IF ID EX MEM WB

sub --, R1, --   IF ID EX MEM WB

R1 read  Data used

Use data before it is in register
• reduces/avoids stalls
• complex
• crucial for common RAW hazards

Bypass Hardware

ID → EX → MEM → WB

Copyright © 2001 Falsafi, from Hill, Smith, Sohi, Vijaykumar, and Wood

© Chapter 3 Pipelining
Bypass Hardware (Cont.)

- Interlock logic
  - detect hazard
  - bypass correct result to ALU

- Hardware detection requires extra hardware
  - instruction latches for each stage
  - comparators to detect hazards

- Mux control
  - if inst(EX) uses immediate then select IMM
  - else if RS2(EX) = RD(MEM) then ALOUT(MEM)
  - else if RS2(EX) = RD(WB) then ALOUT(WB)
  - else select B

RAW Solutions

- Hybrid required sometimes
  - Data available
  - R1 written

- DLX has one cycle bubble if load result used in next instruction

- Try to separate stall logic from bypass logic
  - avoid irregular bypasses
Stall Methods

Compare ahead in pipe
- if rs1(EX) = rd(MEM) or rs2(EX) = rd(MEM) then stall
- assumes MEM instruction is a load, EX instruction is ALU