18-447 Lecture 8: Data Hazard and Resolution

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today
  - detect and resolve data hazards in in-order pipelines
  - control flow will come next lecture
- Notices
  - Lab 2, status check next week, due wk of 2/27
  - Midterm 2/27 in class; covers Lectures 1~10
- Readings
  - P&H Ch 4
Instruction Pipeline Reality

- Identical tasks ... NOT!
  - coalescing instruction types
  - external fragmentation (some idle stages)
- Uniform suboperations ... NOT!
  - balance pipeline stages
  - group or sub-divide steps to minimize variance
  - internal fragmentation (some too-fast stages)
- Independent tasks ... NOT!
  - resolve data and resource hazards
    - duplicate contended resources
    - inter-instruction dependency detection and resolution

MIPS ISA features are engineered for 5-stage pipelining

Data Dependence (on registers)

Data dependence

\[ r_3 \leftarrow r_1 \text{ op } r_2 \]
\[ \ldots \]
\[ r_5 \leftarrow r_3 \text{ op } r_4 \]

Read-after-Write (RAW)

Anti-dependence

\[ r_3 \leftarrow r_1 \text{ op } r_2 \]
\[ \ldots \]
\[ r_1 \leftarrow r_4 \text{ op } r_5 \]

Write-after-Read (WAR)

Output-dependence

\[ r_3 \leftarrow r_1 \text{ op } r_2 \]
\[ \ldots \]
\[ r_3 \leftarrow r_6 \text{ op } r_7 \]

Write-after-Write (WAW)

We discuss control dependence next lecture
RAW Dependency and Hazard

- Following RAW dependencies lead to hazards in the 5-stage pipeline (from last lecture)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addi ra r</code></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td><code>addi r-ra</code></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td><code>addi r-ra</code></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td><code>addi r-ra</code></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>addi r-ra</code></td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>addi r-ra</code></td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Data Hazard Analysis

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Bxx</th>
<th>Jal</th>
<th>Jalr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>write RF</td>
<td>write RF</td>
<td></td>
<td>write RF</td>
<td>write RF</td>
</tr>
</tbody>
</table>

- For a given pipeline, when is there a register data hazard between 2 instructions?
  - dependence type: RAW, WAR, WAW?
  - instruction types involved?
  - distance between the two instructions?
Necessary Condition for Data Hazard

\[ \text{dist}_{\text{dependence}}(i,j) \leq \text{dist}_{\text{hazard}}(X,Y) \Rightarrow \text{Hazard!!} \]
\[ \text{dist}_{\text{dependence}}(i,j) > \text{dist}_{\text{hazard}}(X,Y) \Rightarrow \text{Safe} \]

RAW Hazard Analysis Example

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Bxx</th>
<th>Jal</th>
<th>Jalr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>write RF</td>
<td>write RF</td>
<td></td>
<td>write RF</td>
<td>write RF</td>
</tr>
</tbody>
</table>

- Older \( I_A \) and younger \( I_B \) have RAW hazard iff
  - \( I_B \) \((R/I, \text{LW}, \text{SW}, \text{Bxx} \text{ or } \text{JALR})\) reads a register written by \( I_A \) \((R/I, \text{LW}, \text{or } \text{JAL}/\text{R})\)
  - \( \text{dist}(I_A, I_B) \leq \text{dist}(\text{ID}, \text{WB}) = 3 \)

What about WAW and WAR hazard?
What about memory data hazard?
Pipeline Stall:
universal hazard resolution

Stall==make the younger instruction wait until the hazard has passed
1. stop all up-stream stages
2. drain all down-stream stages

What should happen in this case?

What should happen in this case?
### Pipeline Stall

<table>
<thead>
<tr>
<th></th>
<th>t₀</th>
<th>t₁</th>
<th>t₂</th>
<th>t₃</th>
<th>t₄</th>
<th>t₅</th>
<th>t₆</th>
<th>t₇</th>
<th>t₈</th>
<th>t₉</th>
<th>t₁₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>k</td>
<td>k</td>
<td>k</td>
<td>l</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>h</td>
<td>i</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>h</td>
<td>i</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>h</td>
<td>i</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

i: \(rx \leftarrow _-\)

j: \( _- \leftarrow rx\)

### Stall

- disable \( pc \) and \( ir \) latching
- control should set \( \text{RegWrite}=0 \) and \( \text{MemWrite}=0 \)

*Based on original figure from [P&H, CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED]*
Stall Condition

- Older I_A and younger I_B have RAW hazard iff
  - I_B (R/I, LW, SW, Bxx or JALR) reads a register written by
    I_A (R/I, LW, or JAL/R)
  - dist(I_A, I_B) ≤ dist(ID, WB) = 3

- Stated constructively, before I_B in ID reads a register, I_B needs to check if any I_A in EX, MEM or WB is going to update it (if so, value currently in RF is “stale”)

Watch out for x0!!

Stall Condition

- Helper functions
  - \( rs1(I) \) returns the \( rs1 \) field of \( I \)
  - \( use_{rs1}(I) \) returns true if \( I \) requires RF[rs1] and rs1 \( \neq x0 \)

- Stall IF and ID when
  - \((rs1(IR)^{rd}_{EX})\&\& use_{rs1}(IR)\&\& RegWrite_{EX}\) or
  - \((rs1(IR)^{rd}_{MEM})\&\& use_{rs1}(IR)\&\& RegWrite_{MEM}\) or
  - \((rs1(IR)^{rd}_{WB})\&\& use_{rs1}(IR)\&\& RegWrite_{WB}\) or
  - \((rs2(IR)^{rd}_{EX})\&\& use_{rs2}(IR)\&\& RegWrite_{EX}\) or
  - \((rs2(IR)^{rd}_{MEM})\&\& use_{rs2}(IR)\&\& RegWrite_{MEM}\) or
  - \((rs2(IR)^{rd}_{WB})\&\& use_{rs2}(IR)\&\& RegWrite_{WB}\)

It is crucial that the EX, MEM and WB continue to advance normally during stall cycles
Impact of Stall on Performance

- Each stall cycle corresponds to 1 lost ALU cycle
- For a program with N instructions and S stall cycles, Average IPC=N/(N+S)
- S depends on
  - frequency of hazard-causing dependencies
  - exact distance between the hazard-causing instruction pair
  - distance between hazard-causing dependencies
    (suppose i_2, i_3 and i_4 all depend on i_0, once i_1's hazard is resolved by stalling, i_2 and i_3 do not stall)

Sample Assembly [P&H]

```assembly
for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ...... }

addi $s1, $s0, -1  # 3 stalls
for2tst:
  slti $t0, $s1, 0   # 3 stalls
  bne $t0, $zero, exit2
  sll $t1, $s1, 2    # 3 stalls
  add $t2, $a0, $t1  # 3 stalls
  lw $t3, 0($t2)     # 3 stalls
  lw $t4, 4($t2)     # 3 stalls
  slt $t0, $t4, $t3  # 3 stalls
  beq $t0, $zero, exit2
      ...........
addi $s1, $s1, -1  # 3 stalls
j       for2tst
exit2:
```
Data Forwarding (aka Register Bypassing)

- It is intuitive to think of RF as state
  - “add rx ry rz” literally means get input values from RF[ry] and RF[rz] and put result in RF[rx]
- But, RF is just a part of a computing abstraction
  - “add rx ry rz” means 1. inputs are the results of the last instructions to have defined the values of RF[ry] and RF[rz], and 2. until another instruction redefines RF[rx], younger instructions that refers to RF[rx] should use this instruction’s result
- What matters is to maintain the correct “dataflow” between operations, thus

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>add ra r- r-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi r- ra r-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Resolving RAW Hazard by Forwarding

- Older IA and younger IB have RAW hazard iff
  - IB (R/I, LW, SW, Bxx or JALR) reads a register written by IA (R/I, LW, or JAL/R)
  - dist(IA, IB) ≤ dist(ID, WB) = 3
- Stated constructively, before IB in ID reads a register, IB needs to check if any IA in EX, MEM or WB is going to update it (if so, value currently in RF is “stale”)
- If the value is already produced, don’t stall!
  - retrieve value from datapath before RF write
  - retrieve from the youngest definition if multiple definitions are outstanding
Forwarding Paths (v1)

With forwarding

dist(i, j) = 1

dist(i, j) = 2

dist(i, j) = 3

internal forward?

better if EX is the fastest stage
**Forwarding Logic (for v1)**

if \( (rs_{1,ID} \neq 0) \&\& (rs_{1,ID} == rd_{EX}) \&\& \text{RegWrite}_{EX} \) then
  forward operand from EX // dist=1
else if \( (rs_{1,ID} \neq 0) \&\& (rs_{1,ID} == rd_{MEM}) \&\& \text{RegWrite}_{MEM} \) then
  forward operand from MEM // dist=2
else if \( (rs_{1,ID} \neq 0) \&\& (rs_{1,ID} == rd_{WB}) \&\& \text{RegWrite}_{WB} \) then
  forward operand from WB // dist=3
else
  use \( A_{ib} \) (operand from RF) // dist > 3

Ordering matters!! Must check youngest match first

Why doesn’t \texttt{use_rs1()\hspace{1em}} appear in the forwarding logic?

Wrong value forwarded if matched against LW in EX?

---

**Data Hazard Analysis (with Forwarding)**

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Bxx</th>
<th>Jal</th>
<th>Jalr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>use</td>
<td>produce</td>
<td>use</td>
<td>use</td>
<td>produce</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>produce</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>produce</td>
<td>(use)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Even with data-forwarding, RAW dependence on an immediate preceding LW instruction produces a hazard
- \( \text{Stall} = (rs(IR_{ID}) == rd_{EX}) \&\& \text{use}_rs(IR_{ID}) \&\& \text{MemRead}_{EX} \) & “\text{opcode}_{EX}=Lx”
**MIPS Load “Delay Slot”**

- **R2000 defined load with arch. latency of 1 inst**
  - instruction immediately following a load (in the “delay slot”) still sees the old value
  - dependent instruction at least distance 2, **no more hazard!**
- **Delay slot vs dynamic stalling**
  - fill with an independent instruction (no difference)
  - if not, fill with a would-be WAR instruction (gain 1 cycle)
  - if not, fill with a NOP (no difference)
- **Can’t lose on 5-stage . . . good idea?**
  
  Hint: ISA feature made to fit microarchitecture choice

---

**Sample Assembly [P&H]**

```assembly
for (j=i-1; j>=0 && v[j] > v[j+1]; j--) { .... }  

addi $s1, $s0, -1
for2tst:
  slti $t0, $s1, 0
  bne $t0, $zero, exit2
  sll $t1, $s1, 2
  add $t2, $a0, $t1
  lw $t3, 0($t2)
  lw $t4, 4($t2)
  nop
  slt $t0, $t4, $t3
  beq $t0, $zero, exit2
  ........
addi $s1, $s1, -1
j for2tst

exit2:
```
Terminology

- **Dependency**
  - ordering requirement between instructions
- **Pipeline Hazard**
  - (potential) violation of dependencies
- **Hazard Resolution**
  - static \(\Rightarrow\) schedule instructions at compile time to avoid hazards
  - dynamic \(\Rightarrow\) detect hazard and adjust pipeline operation

- **Pipeline Interlock** (i.e., stall)

MIPS = Microprocessor without Interlocked Pipeline Stages

---

Dividing into Stages

Is this the correct partitioning? Why not 4 or 6 stages? Why not different boundaries?
Why not very deep pipelines?

- 5-stage pipeline still has plenty of combinational delay between registers
- “Superpipelining” ⇒ increase pipelining such that even intrinsic operations (e.g. ALU, RF access, memory access) require multiple stages
- What’s the problem?

![Diagram]

Inst₀: \( r₁ \leftarrow r₂ + r₃ \)
Inst₁: \( r₄ \leftarrow r₁ + 2 \)

Intel P4’s Superpipelined Adder Hack

32-bit addition pipelined over 2 stages, \( BW=1/\text{latency}_{16\text{-bit-add}} \)
No stall between back-to-back dependencies
When you can’t split a stage . . .

I (BW=2/T) → A (BW=1/T) → B (BW=1/T) → O (BW=2/T)

Dependencies and Pipelining
(architecture vs. microarchitecture)

Sequential and atomic instruction semantics

True dependence between two instructions may only require ordering of certain sub-operations

This is an overspecification. It defines what is correct but doesn’t say must actually do it this way