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 instruction pipelines

• Notices
  – Lab 2, status check next week, due week of 2/25
  – HW 2, due 2/20 **before class**
  – Irregular office hours this week:
    Mon 10~10:30; Wed cancelled (Jury Duty)

• Readings
  – P&H Ch 4
Instruction Pipeline Reality

• Not identical tasks
  – coalescing instruction types into one “multi-function” pipe
  – external fragmentation (some idle stages)

• Not uniform suboperations
  – group or sub-divide steps into stages to minimize variance
  – internal fragmentation (some too-fast stages)

• Not independent tasks
  – dependency detection and resolution
  – next lecture(s)

Even more messy if not RISC
Data Dependence

Data dependence

\[ r_3 \leftarrow r_1 \text{ op } r_2 \]

... 

\[ r_5 \leftarrow r_3 \text{ op } r_4 \]

Read-after-Write (RAW)

Anti-dependence

\[ r_3 \leftarrow r_1 \text{ op } r_2 \]

... 

\[ r_1 \leftarrow r_4 \text{ op } r_5 \]

Write-after-Read (WAR)

Output-dependence

\[ r_3 \leftarrow r_1 \text{ op } r_2 \]

... 

\[ r_3 \leftarrow r_6 \text{ op } r_7 \]

Write-after-Write (WAW)

Don’t forget memory instructions
RAW Dependency and Hazard

\[
\begin{array}{ccccccc}
\text{addi} & ra & r- & - & \text{IF} & \text{ID} & \text{EX} \\
\text{addi} & r- & ra & - & \text{IF} & \text{ID} & \text{EX} \\
\text{addi} & r- & ra & - & \text{IF} & \text{ID} & \text{EX} \\
\text{addi} & r- & ra & - & \text{IF} & \text{ID} & \text{EX} \\
\text{addi} & r- & ra & - & \text{IF} & \text{ID} & \text{EX} \\
\end{array}
\]
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 dependent instructions?
  – dependence type: RAW, WAR, WAW?
  – instruction types involved?
  – distance between the two instructions?
Hazard in In-order Pipeline

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

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

<table>
<thead>
<tr>
<th></th>
<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>
<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>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></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>
<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>
<td></td>
</tr>
</tbody>
</table>

- 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)
  - $\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID}, \text{WB}) = 3$

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

Stall==make younger instruction wait until hazard passes
1. stop all up-stream stages
2. drain all down-stream stages
## Pipeline Stall

<table>
<thead>
<tr>
<th></th>
<th>$t_0$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$t_5$</th>
<th>$t_6$</th>
<th>$t_7$</th>
<th>$t_8$</th>
<th>$t_9$</th>
<th>$t_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IF</strong></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><strong>ID</strong></td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>i</td>
<td>k</td>
<td>l</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>EX</strong></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><strong>MEM</strong></td>
<td>h</td>
<td>i</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td>h</td>
<td>i</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ i: \text{rx} \leftarrow _{-} \]

\[ j: _{-} \leftarrow \text{rx} \]
Pop Quiz: What happens in this case?

Inst_{h}

<table>
<thead>
<tr>
<th>t_0</th>
<th>t_1</th>
<th>t_2</th>
<th>t_3</th>
<th>t_4</th>
<th>t_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID</td>
<td>ALU</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>

Inst_{i}

| i   | IF  | ID  | ALU | MEM | WB  |

Inst_{j}

| j   | IF  | ID  | ALU | MEM | WB  |

Inst_{k}

| k   | IF  | ID  | ALU | MEM | WB  |

Inst_{l}

| i:  | r_x &laquo; _  |
| j:  | r_y &laquo; r_z |
| k:  | _ &laquo; r_x   | dist(i,k)=2   |
• Stall

– disable PC and IR latching

– set RegWrite\_ID=0 and MemWrite\_ID=0
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)
  - $\text{dist}(I_A, I_B) \leq \text{dist(ID, WB)} = 3$

- More plainly, 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 in RF is “stale”)

Watch out for x0!!
Stall Condition

- **Helper functions**
  - \texttt{use\_rs1(I)} returns true if \texttt{I} uses \texttt{rs1} \&\& \texttt{rs1!}=\texttt{x0}

- **Stall IF and ID when**
  - \((\texttt{rs1}\text{\texttt{\_id}}==\texttt{rd}_\text{\texttt{ex}}) \&\& \texttt{use\_rs1(\texttt{ir}\text{\texttt{\_id}})} \&\& \texttt{RegWrite}_{\texttt{ex}}\) or
  - \((\texttt{rs1}\text{\texttt{\_id}}==\texttt{rd}_\text{\texttt{mem}}) \&\& \texttt{use\_rs1(\texttt{ir}\text{\texttt{\_id}})} \&\& \texttt{RegWrite}_{\texttt{mem}}\) or
  - \((\texttt{rs1}\text{\texttt{\_id}}==\texttt{rd}_\text{\texttt{wb}}) \&\& \texttt{use\_rs1(\texttt{ir}\text{\texttt{\_id}})} \&\& \texttt{RegWrite}_{\texttt{wb}}\) or
  - \((\texttt{rs2}\text{\texttt{\_id}}==\texttt{rd}_\text{\texttt{ex}}) \&\& \texttt{use\_rs2(\texttt{ir}\text{\texttt{\_id}})} \&\& \texttt{RegWrite}_{\texttt{ex}}\) or
  - \((\texttt{rs2}\text{\texttt{\_id}}==\texttt{rd}_\text{\texttt{mem}}) \&\& \texttt{use\_rs2(\texttt{ir}\text{\texttt{\_id}})} \&\& \texttt{RegWrite}_{\text{\texttt{mem}}}\) or
  - \((\texttt{rs2}\text{\texttt{\_id}}==\texttt{rd}_\text{\texttt{wb}}) \&\& \texttt{use\_rs2(\texttt{ir}\text{\texttt{\_id}})} \&\& \texttt{RegWrite}_{\texttt{wb}}\)

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

• Each stall cycle corresponds to 1 lost ALU cycle
• A program with $N$ instructions and $S$ stall cycles:
  \[
  \text{average IPC} = \frac{N}{N+S}
  \]
• $S$ depends on
  – frequency of hazard-causing dependencies
  – distance between hazard-causing instruction pairs
  – distance between hazard-causing dependencies
    (suppose $i_1, i_2$ and $i_3$ 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]

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

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)
slt $t0, $t4, $t3
beq $t0, $zero, exit2

........
addi $s1, $s1, -1
j for2tst

exit2:
Data Forwarding (or Register Bypassing)

• What does “ADD rx ry rz” mean? Get inputs from RF[ry] and RF[rz] and put result in RF[rx]?
• But, RF is just a part of an abstraction
  – a way to connect dataflow between instructions
    “inputs to ADD are resulting values of the last instructions to assign to RF[ry] and RF[rz]”
  – RF doesn’t have to exist as a literal object
• If only dataflow matters, don’t wait for WB . . .
Resolving RAW Hazard by Forwarding

- 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)
  - $\text{dist}(I_A, I_B) \leq \text{dist}($ID, WB$) = 3$
- More plainly, 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 in RF is “stale”)
- Before: $I_B$ need to stall for RF to update
- Now: $I_B$ need to stall for $I_A$ to produce result
  - retrieve $I_A$ result from datapath when ready
  - must retrieve from youngest if multiple hazards
Forwarding Paths (v1)

Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.
Forwarding Paths (v2)

better if EX is the fastest stage

Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.
Forwarding Logic (for v1)

if \((rs1_{id}\neq 0) \&\& (rs1_{id}==rd_{ex}) \&\& \text{RegWrite}_{ex}\) then
  forward writeback value from EX \hspace{1cm} // dist=1
else if \((rs1_{id}\neq 0) \&\& (rs1_{id}==rd_{mem}) \&\& \text{RegWrite}_{mem}\) then
  forward writeback value from MEM \hspace{1cm} // dist=2
else if \((rs1_{id}\neq 0) \&\& (rs1_{id}==rd_{wb}) \&\& \text{RegWrite}_{wb}\) then
  forward writeback value from WB \hspace{1cm} // dist=3
else
  use \(A_{id}\) \hspace{1cm} // dist > 3

Must check in right order

Why doesn’t \text{use}_rs1\()\) appear?
Isn’t it bad to forward from 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 produce</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td>produce</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 forwarding, RAW dependence on immediate preceding LW results in hazard
- \( \text{Stall} = \{ [(rs_{1d} == rd_{ex}) && use\_rs1(IR_{id})] \mid \mid [(rs_{2d} == rd_{ex}) && use\_rs2(IR_{id})] \} \land \text{MemRead}_{ex} \)  
  i.e., \( op_{ex} = Lx \)
Historical: MIPS Load “Delay Slot”

- R2000 defined LW with arch. latency of 1 inst
  - invalid for \( l_2 \) (in LW’s delay slot) to ask for LW’s result
  - any dependence on LW at least distance 2

- Delay slot vs dynamic stalling
  - fill with an independent instruction (no difference)
  - if not, fill with a NOP (no difference)

- Can’t lose on 5-stage ... good idea?

Hint: 1. non-atomic instruction; 2. \( \mu \)arch influence
Sample Assembly [P&H]

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

    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)
    slt $t0, $t4, $t3
    beq $t0, $zero, exit2

    ........

    addi $s1, $s1, -1

exit2:

    j for2tst

1 stall or 1 nop (MIPS)
Why not very deep pipelines?

- With only 5 stages, still plenty of combinational logic between registers
- “Superpipelining” ⇒ increase pipelining such that even intrinsic operations (e.g. ALU, RF access, memory access) require multiple stages
- What’s the problem?  
  \[
  \text{Inst}_0: \text{r1} \leftarrow \text{r2} + \text{r3} \\
  \text{Inst}_1: \text{r4} \leftarrow \text{r1} + 2
  \]
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)**
Dependencies and Pipelining
(architecture vs. microarchitecture)

Sequential and atomic instruction semantics

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

Defines what is correct; doesn’t say do it this way
Lab 2 with 6 Stages

200ps
IF: Instruction fetch

100ps
ID: Instruction decode/register file read

200ps
EX: Execute/address calculation

200ps
MEM: Memory access

100ps
WB: Write back

Ignore for today

Basic pipeline diagram with stages:
- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execute (EX)
- Memory Access (MEM)
- Write Back (WB)

Instruction memory connected to PC, which is used to fetch instructions. Instructions flow through the pipeline stages, with each stage represented by a box and the time each stage takes indicated (200ps, 100ps, etc.). The diagram also shows connections for data flow and register file reads.

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
# Lab 2: Data Hazard Analysis with Stalling

<table>
<thead>
<tr>
<th></th>
<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><strong>IF</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ID</strong></td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
<td></td>
<td>read RF</td>
</tr>
<tr>
<td><strong>EX</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MEM1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MEM2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td>write RF</td>
<td>write RF</td>
<td>write RF</td>
<td></td>
<td>write RF</td>
<td>write RF</td>
</tr>
</tbody>
</table>
# Lab 2: 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>use</td>
<td>use</td>
<td>use</td>
<td>produce</td>
</tr>
<tr>
<td>MEM1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>produce</td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **use** indicates that the value is used in the current stage.
- **produce** indicates that the value is produced in the current stage.
- A red cross indicates a hazard that needs to be resolved.

Note: The table shows how data hazards can be analyzed and resolved through forwarding in a pipeline stage of a computer architecture.