18-447 Lecture 11: Pipelined Implementations: Hazards and Resolutions

James C. Hoe
Dept of ECE, CMU
February 25, 2009

Announcements: Project 1 due this week

Handouts:

Instruction Pipeline Reality

- Identical operations ... NOT!
  ⇒ unifying instruction types
    - coalescing instruction types into one "multi-function" pipe
    - external fragmentation (some idle stages)

- Uniform Suboperations ... NOT!
  ⇒ balance pipeline stages
    - stage quantization to yield balanced stages
    - internal fragmentation (some too-fast stages)

- Independent operations ... NOT!
  ⇒ resolve data and resource hazards
    - duplicate contended resources
    - inter-instruction dependency detection and resolution

MIPS ISA features are engineered for improved pipelineability
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
\[ r_3 \leftarrow r_1 \text{ op } r_2 \] \[ r_5 \leftarrow r_3 \text{ op } r_4 \] \[ r_3 \leftarrow r_6 \text{ op } r_7 \]
Write-after-Write (WAW)

We discuss control-flow dependence in a later lecture.

Dependencies and Pipelined Execution

Sequential and atomic instruction semantics

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

This semantics is an overspecification. It defines what is correct but doesn't say to do it that way only.
RAW Dependency and Hazard

- Following RAW dependencies lead to hazards in the 5-stage pipelined from L10

```
addi r, ra, -    IF  ID  EX  MEM  WB
addi r, ra, -    IF  ID  EX  MEM  WB
addi r, ra, -    IF  ID  EX  MEM  WB
addi r, ra, -    IF  ID  EX  MEM  WB
addi r, ra, -    IF  ID  EX  MEM  WB
addi r, ra, -    IF  ID  EX  MEM  WB
```

Register Data Hazard Analysis

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Br</th>
<th>J</th>
<th>Jr</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></td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
\text{dist}(i,j) \leq \text{dist}(X,Y) & \Rightarrow \text{Hazard!!} \\
\text{dist}(i,j) > \text{dist}(X,Y) & \Rightarrow \text{Safe}
\end{align*}
\]

RAW Hazard Analysis Example

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Br</th>
<th>J</th>
<th>Jr</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></td>
<td></td>
</tr>
</tbody>
</table>

- Instructions \(I_A\) and \(I_B\) (where \(I_A\) comes before \(I_B\)) have RAW hazard iff
  - \(I_B\) (R/I, LW, SW, Br or JR) reads a register written by \(I_A\) (R/I or LW)
  - \(\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID, WB}) = 3\)

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

i: \( r_x \leftarrow \_ \) bubble bubble bubble
j: \( \_ \leftarrow r_x \) \( \text{dist}(i,j)=4 \)
### 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>IF</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>k</td>
<td>k</td>
<td>k</td>
<td>k</td>
<td>k</td>
<td>k</td>
<td>l</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>bub</td>
<td>j</td>
<td>k</td>
<td>l</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></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>
</tr>
</tbody>
</table>

i: $rx \leftarrow _{-}$

j: $-_{} \leftarrow rx$

### Stall

- disable PC and IR latching
- control should set RegWrite=0 and MemWrite=0
Stall Conditions

- Instructions $I_A$ and $I_B$ (where $I_A$ comes before $I_B$) have RAW hazard iff
  - $I_B$ ($R/I$, $LW$, $SW$, $Br$ or $JR$) reads a register written by $I_A$ ($R/I$ or $LW$)
  - $\text{dist}(I_A, I_B) \leq \text{dist}(ID, WB) = 3$

- In other words, must stall when $I_B$ in ID stage wants to read a register to be written by $I_A$ in EX, MEM or WB stage

Stall Condition

- Helper functions
  - $\text{rs}(I)$ returns the $rs$ field of $I$
  - $\text{use_rs}(I)$ returns true if $I$ requires $RF[rs]$ and $rs \neq r0$

- Stall when
  - $(\text{rs}(\text{IR}_{ID}) = \text{dest}_{EX}) \& \& \text{use_rs}(\text{IR}_{ID}) \& \& \text{RegWrite}_{EX}$ or
  - $(\text{rs}(\text{IR}_{ID}) = \text{dest}_{MEM}) \& \& \text{use_rs}(\text{IR}_{ID}) \& \& \text{RegWrite}_{MEM}$ or
  - $(\text{rt}(\text{IR}_{ID}) = \text{dest}_{WB}) \& \& \text{use_rt}(\text{IR}_{ID}) \& \& \text{RegWrite}_{WB}$ or
  - $(\text{rt}(\text{IR}_{ID}) = \text{dest}_{MEM}) \& \& \text{use_rt}(\text{IR}_{ID}) \& \& \text{RegWrite}_{MEM}$ or
  - $(\text{rt}(\text{IR}_{ID}) = \text{dest}_{WB}) \& \& \text{use_rt}(\text{IR}_{ID}) \& \& \text{RegWrite}_{WB}$

It is crucial that the EX, MEM and WB stages 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, \( \text{Average IPC} = \frac{N}{N+S} \)
- \( S \) depends on
  - frequency of RAW hazards
  - exact distance between the hazard-causing instructions
  - distance between hazards
    - suppose \( i_1, i_2 \) and \( i_3 \) all depend on \( i_0 \), once \( i_1 \)'s hazard is resolved, \( i_2 \) and \( i_3 \) must be okay too

Sample Assembly [p126, P&H]

```assembly
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

- It is intuitive to think of RF as state
  - “add rx ry rz” literally means get values from RF[ry] and RF[rz], respectively and put result in RF[rx]
- But, RF is just a part of a computing abstraction
  - “add rx ry rz” means 1. get the results of the last instructions to define the values of RF[ry] and RF[rz], respectively, 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

```
add ra r- r-
addi r- ra r-
```

Resolving RAW Hazard by Forwarding

- Instructions $I_A$ and $I_B$ (where $I_A$ comes before $I_B$) have RAW hazard iff
  - $I_B$ (R/I, LW, SW, Br or JR) reads a register written by $I_A$ (R/I or LW)
  - dist($I_A$, $I_B$) ≤ dist(ID, WB) = 3
- In other words, if $I_B$ in ID stage reads a register written by $I_A$ in EX, MEM or WB stage, then the operand required by $I_B$ is not yet in RF
  ⇒ retrieve operand from datapath instead of the RF
  ⇒ retrieve operand from the youngest definition if multiple definitions are outstanding
b. With forwarding

Assumes RF forwards internally

[Based on figures from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Forwarding Logic (for v2)

if \((rs_{EX} != 0) \&\& (rs_{EX} == dest_{MEM}) \&\& RegWrite_{MEM}\) then
forward operand from MEM stage \(\text{ // dist}=1\)
else if \((rs_{EX} != 0) \&\& (rs_{EX} == dest_{WB}) \&\& RegWrite_{WB}\) then
forward operand from WB stage \(\text{ // dist}=2\)
else
use \(A_{EX}\) (operand from register file) \(\text{ // dist} \geq 3\)

Ordering matters!! Must check youngest match first

Why doesn't \text{ use } rs( ) appear in the forwarding logic?

Data Hazard Analysis (with Forwarding)

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Br</th>
<th>J</th>
<th>Jr</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>use</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td>use</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
Load Delay Slot

R2000 defined load with arch. latency of 1 inst
- the instruction immediately following a load (in the "delay slot") still sees the old register value (this is the behavior if we don't do anything special beyond forwarding)
- ISA feature tailored to the 5-stage pipelined microarchitecture  Warning!! Implementation exposed!!

If loads are defined normally, i.e., atomic
- a dependent immediate successor to LW must stall 1 cycle in ID
- \[\text{Stall} = (rs(\text{IR}_{ID})==\text{dest}_{EX}) \&\& \text{use}_{rs}(\text{IR}_{ID}) \&\& \text{MemRead}_{EX}\]

Sample Assembly [p126, 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) 
nop 
slt $t0, $t4, $t3 
beq $t0, $zero, exit2 
........ 
addi $s1, $s1, -1 
j for2tst 
exit2:
```
Terminology

- Dependencies
  - ordering requirement between instructions

- Pipeline Hazards:
  - (potential) violations of dependencies

- Hazard Resolution:
  - static ⇒ schedule instructions at compile time to avoid hazards
  - dynamic ⇒ detect hazard and adjust pipeline operation
    
    Stall, Flush or Forward

- Pipeline Interlock:
  - hardware mechanisms for dynamic hazard resolution
  - detect and enforce dependences at run time

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? Inst\(_0\): \( r1 \leftarrow r2 + r3 \)  
  Inst\(_1\): \( r4 \leftarrow r1 + 2 \)
Intel P4's Superpipelined Integer ALU

32-bit addition pipelined over 2 stages, BW=1/latency
No stall between back-to-back dependencies

What if you really can’t superpipeline?

If you can’t double the bandwidth by pipelining, doubling the resource also doubles the bandwidth