MIPS Pipeline Hazards

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Data Hazards

- Consider this sequence:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>$2$</th>
<th>$1$</th>
<th>$3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>$12$</td>
<td>$2$</td>
<td>$5$</td>
</tr>
<tr>
<td>or</td>
<td>$13$</td>
<td>$6$</td>
<td>$2$</td>
</tr>
<tr>
<td>add</td>
<td>$14$</td>
<td>$2$</td>
<td>$2$</td>
</tr>
<tr>
<td>sw</td>
<td>$15$, $100$($2$)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- We can resolve hazards with forwarding
  - How do we detect when to forward?
Dependencies & Forwarding

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of register $s2$:</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
</tr>
</tbody>
</table>

Program execution order (in instructions):

- sub $s2, $1, $3
- and $12, $s2, $5
- or $13, $6, $2
- add $s14, $2, $2
- sw $15, 100($s2)
Forwarding Conditions (1)

- Detecting the need to forward
  - Pass register numbers along pipeline
    - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
  - ALU operand register numbers in EX stage are given by ID/EX.RegisterRs & ID/EX.RegisterRt

- Data hazards when
  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

\begin{align*}
\text{Fwd from EX/MEM pipeline reg} \quad & \text{Fwd from MEM/WB pipeline reg} \\
\end{align*}
Forwarding Conditions (2)

- Detecting the need to forward (cont’d)
  - But only if forwarding instruction will write to a register!
    - EX/MEM.RegWrite, MEM/WB.RegWrite
  - And only if Rd for that instruction is not $zero
    - EX/MEM.RegisterRd ≠ 0,
      MEM/WB.RegisterRd ≠ 0
Forwarding Conditions (3)

- Forwarding paths
Forwarding Conditions (4)

- **EX hazard**
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    \[\text{ForwardA} = 10\]
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    \[\text{ForwardB} = 10\]

- **MEM hazard**
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    \[\text{ForwardA} = 01\]
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    \[\text{ForwardB} = 01\]
Double Data Hazard

- Consider this sequence:

```
add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
```

- Both hazards occur
  - Want to use the most recent

- Revise MEM hazard condition
  - Only forward if EX hazard condition isn’t true
Rev’d Forwarding Conditions

- **MEM hazard**
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and 
    not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and
    (EX/MEM.RegisterRd = ID/EX.RegisterRs)) and
    (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01

  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and 
    not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and
    (EX/MEM.RegisterRd = ID/EX.RegisterRt))) and
    (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01
Datapath with Forwarding
Load-Use Data Hazard (1)

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC 1</td>
</tr>
</tbody>
</table>

Program execution order (in instructions)

- lw $2, 20($1)
- and $4, $2, $5
- or $8, $2, $6
- add $9, $4, $2
- slt $1, $6, $7

Need to stall for one cycle
Load-Use Data Hazard (2)

- Load-use hazard detection
  - Check when using instruction is decoded in ID stage
  - ALU operand register numbers in ID stage are given by IF/ID.RegisterRs & IF/ID.RegisterRt
  - Load-use hazard when
    - ID/EX.MemRead and
      - ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
        (ID/EX.RegisterRt = IF/ID.RegisterRt))
  - If detected, stall and insert bubble
How to stall the pipeline?

- Force control values in ID/EX register to 0
  - EX, MEM, and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for \texttt{lw}
    » Can subsequently forward to EX stage
Load-Use Data Hazard (4)

- Stall/Bubble in the pipeline

Program execution order (in instructions)

- `lw $2, 20($1)` and becomes `nop`
- `and $4, $2, $5`
- `or $8, $2, $6`
- `add $9, $4, $2`

Time (in clock cycles)

CC 1  CC 2  CC 3  CC 4  CC 5  CC 6  CC 7  CC 8  CC 9  CC 10

Stall inserted here
Load-Use Data Hazard (5)

- Stall/Bubble in the pipeline (cont’d)

Program execution order (in instructions)

- \( \text{lw } \$2, \ 20(\$1) \)
- and becomes nop
- and \( \$4, \$2, \$5 \) stalled in ID
- or \( \$8, \$2, \$6 \) stalled in IF
- \( \text{add } \$9, \$4, \$2 \)

Or, more accurately...

---

ICE3003: Computer Architecture  |  Fall 2009  |  Jin-Soo Kim (jinsookim@skku.edu)
Datapath + Hazard Detection
Stalls and Performance

- Stalls reduce performance
  - But are required to get correct results

- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure
Branch Hazards (1)

- If branch outcome determined in MEM

Flush these instructions (Set control values to 0)
Branch Hazards (2)

- Reducing branch delay
  - Move hardware to determine outcome to ID stage
    - Target address adder
    - Register comparator

- Example: branch taken

```assembly
36:   sub  $10, $4, $8
40:   beq $1, $3, 7
44:   and $12, $2, $5
48:   or  $13, $2, $6
52:   add $14, $4, $2
56:   slt $15, $6, $7
   ... 
72:   lw  $4, 50($7)
```
Branch Hazards (3)

and $12, $2, $5

beq $1, $3, 7

sub $10, $4, $6

before<1>

before<2>
Data hazards for branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction
  - Can resolve using forwarding
Data hazards for branches (cont’d)

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  → Need 1 stall cycle

lw $1, addr

add $4, $5, $6

beq stalled

beq $1, $4, target
Branch Hazards (7)

- Data hazards for branches (cont’d)
  - If a comparison register is a destination of immediately preceding load instruction
    → Need 2 stall cycles

\[
\text{lw} \quad \text{addr}
\]

\[
\text{beq} \quad \text{stalled}
\]

\[
\text{beq} \quad \text{stalled}
\]

\[
\text{beq} \quad \text{addr}, \text{target}
\]
Dynamic branch prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    » Check table, expect the same outcome
    » Start fetching from fall-through or target
    » If wrong, flush pipeline and flip prediction
1-bit predictor: shortcoming

- Inner loop branches mispredicted twice!

```
outer: ...
...
inner: ...
...
beq ..., ..., inner
...
beq ..., ..., outer
```

- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around
2-bit predictor

- Only change prediction on two successive mispredictions
Branch Hazards (11)

- Calculating the branch target
  - Even with predictor, still need to calculate the target address
    - 1-cycle penalty for a taken branch
  - Branch target buffer
    - Cache of target addresses
    - Indexed by PC when instruction fetched
      » If hit and instruction is branch predicted taken, can fetch target immediately
**Branch Hazards (12)**

- **Delayed branch**
  - Branch delay slot filled by a useful instruction
  - Done by compilers and assemblers