Pipeline Hazards

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

- **What are hazards?**
  - Situations that prevent starting the next instruction in the next cycle

- **Structure hazards**
  - A required resource is busy

- **Data hazards**
  - Need to wait for previous instruction to complete its data read/write

- **Control hazards**
  - Deciding on control action depends on previous instruction
Structure Hazards

- Conflict for use of a resource

- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    → Would cause a pipeline “bubble”

- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
Data Hazards (1)

- An instruction depends on completion of data access by a previous instruction.

```plaintext
add $s0, $t0, $t1
sub $t2, $s0, $t3
```
Data Hazards (2)

- Read After Write (RAW)
  - Inst J tries to read operand before Inst I writes it:
    
    \[
    \begin{align*}
    I: & \text{ add } \$t1, \$t2, \$t3 \\
    J: & \text{ sub } \$t4, \$t1, \$t3
    \end{align*}
    \]
  
  - Caused by a “(true) dependence”. This hazard results from an actual need for communication.
Data Hazards (3)

- **Write After Read (WAR)**
  - Inst J writes operand before Inst I reads it:

    I: sub $t4, $t1, $t3
    J: add $t1, $t2, $t3
    K: add $t6, $t1, $t7

  - Called an "anti-dependence" by compiler writers. This results from the reuse of the name "$t1".
Data Hazards (4)

- Write After Write (WAW)
  - Inst J writes operand before Inst I writes it:

    I: sub $t1, $t4, $t3
    J: add $t1, $t2, $t3
    K: add $t6, $t1, $t7

  - Called an “output dependence” by compiler writers. This also results from the reuse of the name “$t1”.
Forwarding (aka Bypassing)

- Use result when it is computed
- Don’t wait for it to be stored in a register
- Requires extra connections in the datapath
Forwarding Example

- Consider this sequence:

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

- 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 $2$</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 $2$, $1$, $3$
- and $12$, $2$, $5$
- or $13$, $6$, $2$
- add $14$, $2$, $2$
- sw $15$, 100($2$)
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

Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg
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)

Load-Use data hazards

- Can’t always avoids stalls by forwarding
- If value not computed when needed
- Can’t forward backward in time!
Load-Use Data Hazard (2)

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 (3)

- 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
Load-Use Data Hazard (4)

- **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 \( lw \)
      - Can subsequently forward to EX stage
Load-Use Data Hazard (5)

- **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**

  - **Stall inserted here**
Load-Use Data Hazard (6)

Stall/Bubble in the pipeline (cont’d)

Program execution order (in instructions)

- Iw $2, 20($1)
- and becomes nop
- and $4, $2, $5 stalled in ID
- or $8, $2, $6 stalled in IF
- add $9, $4, $2

Or, more accurately...
Datapath + Hazard Detection
Compiler Approach (1)

- **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
Compiler Approach (2)

- Code scheduling to avoid stalls
  - Reorder code to avoid use of load result in the next instruction

```
(C code)   A = B + E;   C = B + F;
```

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add  $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add  $t5, $t1, $t4
sw  $t5, 16($t0)
```

13 cycles

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add  $t3, $t1, $t2
sw  $t3, 12($t0)
add  $t5, $t1, $t4
sw  $t5, 16($t0)
```

11 cycles
Control Hazards (1)

- **Branch determines flow of control**
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch

- **In MIPS pipeline**
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage
Control Hazards (2)

- Example: branch outcome determined in MEM
  - 3 stall cycles for taken branch

```
Time (in clock cycles)
CC 1   CC 2   CC 3   CC 4   CC 5   CC 6   CC 7   CC 8   CC 9
```

Program execution order (in instructions)

- 40 beq $1, $3, 28
- 44 and $12, $2, $5
- 48 or $13, $6, $2
- 52 add $14, $2, $2
- 72 lw $4, 50($7)

Flush these instructions (Set control values to 0)
Control Hazards (3)

- **Branch handling in ID + Stall on branch**
  - Branch outcome at the end of ID stage
  - Wait until branch outcome determined before fetching next instruction
  - One cycle penalty for every branch instruction

![Diagram showing program execution order and time](image)
Control Hazards (4)

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

- Example: branch taken

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)
Control Hazards (5)
Control Hazards (6)
Data Hazards for Branches (1)

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

```
add $1, $2, $3
add $4, $5, $6
...
beq $1, $4, target
```
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
```
Data Hazards for Branches (3)

Data hazards for branches (cont’d)

• If a comparison register is a destination of immediately preceding load instruction
  → Need 2 stall cycles

```
lw  $1, addr
beq stalled
beq stalled
beq $1, $0, target
```
Branch Prediction (1)

- Branch prediction
  - Longer pipelines can’t readily determine branch outcome early
    - Stall penalty becomes unacceptable
  - Predict outcome of branch
    - Only stall if prediction is wrong
  - In MIPS pipeline
    - Can predict branches not taken
    - Fetch instruction after branch, with no delay
Branch Prediction (2)

- **Static branch prediction**
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken

- **Dynamic branch prediction**
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history
Static Branch Prediction

- MIPS with “Predict Not Taken”
Dynamic Branch Prediction (1)

- **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
Dynamic Branch Prediction (2)

- 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
Dynamic Branch Prediction (3)

- **2-bit predictor**
  - Only change prediction on two successive mispredictions

![Diagram showing 2-bit predictor logic]

- Taken
  - Predict taken
  - Not taken
  - Taken
  - Predict not taken
  - Not taken
  - Taken

- Not taken
  - Predict taken
  - Taken
  - Predict not taken
  - Not taken
  - Taken
Dynamic Branch Prediction (4)

- 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
Delayed Branch (1)

- Branch delay slot filled by a useful instruction
  - A branch always executes the following instruction.
  - Done by compilers and assemblers

![Diagram of delayed branch example]
Delayed Branch (2)

- **Compiler approach**
  - Simple and effective for a single branch delay slot
  - Used in MIPS
    - jal Addr: R[31]=PC+8; PC=PC+4[31..28]|Addr*4

- **Modern processors favor dynamic branch prediction**
  - Branch delay slot becomes longer
    - Longer pipelines
    - Multiple instructions issue per clock cycle
  - Dynamic branch prediction is more flexible
  - The growth in available transistors