The Processor (3)

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Last time… (Pipelined Datapath and Control)
Hazards

• Situations that prevent starting the next instruction in the next cycle

• Hazard 1: Structure hazards
  – A required resource is busy

• Hazard 2: Data hazard
  – Need to wait for previous instruction to complete its data read/write

• Hazard 3: Control hazard
  – 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
Outline

Textbook: P&H 4.7-4.8 (For both 4th and 5th Editions)

• Overview

• Data Hazards
  – Forwarding (Bypassing)
  – Stalling for Load-Use Hazards

• Control Hazards
Data Hazards (Forwarding)
Data Hazards

• An instruction depends on completion of data access by a previous instruction
  – add $s0, $t0, $t1
  – sub $t2, $s0, $t3
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
Data Hazards in ALU Instructions

• 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>Value of register $2$:</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></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($2$)
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
Detecting the Need to Forward

• 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 Paths

b. With forwarding
Forwarding Conditions

• EX hazard
  – if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  – if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10

• MEM hazard
  – if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  – if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01
Double Data Hazard

• Consider the 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 fwd if EX hazard condition isn’t true
Revised Forwarding Condition

• 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
Data Hazards (Stalling)
Load-Use Data Hazard

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

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 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
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`

Diagram showing the pipeline with stall/bubble inserted.
Stall/Bubble in the Pipeline

Program execution order (in instructions)

- `lw $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 with 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
Code Scheduling to Avoid Stalls (by Compiler)

• Reorder code to avoid use of load result in the next instruction

• C code for \( A = B + E; \ C = B + F; \)

\[
\begin{align*}
\text{lw} & \quad \$t1, 0($t0) \\
\text{lw} & \quad \$t2, 4($t0) \\
\text{add} & \quad \$t3, \$t1, \$t2 \\
\text{sw} & \quad \$t3, 12($t0) \\
\text{lw} & \quad \$t4, 8($t0) \\
\text{add} & \quad \$t5, \$t1, \$t4 \\
\text{sw} & \quad \$t5, 16($t0)
\end{align*}
\]

\[
\begin{align*}
\text{lw} & \quad \$t1, 0($t0) \\
\text{lw} & \quad \$t2, 4($t0) \\
\text{lw} & \quad \$t4, 8($t0) \\
\text{add} & \quad \$t3, \$t1, \$t2 \\
\text{sw} & \quad \$t3, 12($t0) \\
\text{add} & \quad \$t5, \$t1, \$t4 \\
\text{sw} & \quad \$t5, 16($t0)
\end{align*}
\]

13 cycles

11 cycles
Control Hazards
Control Hazards

• 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
Stall on Branch

- Wait until branch outcome determined before fetching next instruction
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
MIPS with Predict Not Taken

Prediction correct

Program execution order (in instructions)

add $4, $5, $6
beq $1, $2, 40
lw $3, 300($0)

Prediction incorrect

Program execution order (in instructions)

add $4, $5, $6
beq $1, $2, 40

or $7, $8, $9

bubble bubble bubble bubble bubble
More-Realistic Branch Prediction

• 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
Branch Hazards

- If branch outcome determined in MEM

![Diagram of branch hazards]

Flush these instructions (Set control values to 0)
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)
```
Example: Branch Taken
Example: Branch Taken

IF.ID | IF.Flush
--- | ---
lw $4, 50($7)

Bubble (nop)

beq $1, $3, 7

sub $10, ...

before<1>

Clock 4
Data Hazards for Branches

• If a comparison register is a destination of 2\textsuperscript{nd} or 3\textsuperscript{rd} preceding ALU instruction

```
add $1, $2, $3
add $4, $5, $6
...  
beq $1, $4, target
```

• Can resolve using forwarding
Data Hazards for Branches

• If a comparison register is a destination of preceding ALU instruction or 2\textsuperscript{nd} preceding load instruction
  – Need 1 stall cycle

\begin{verbatim}
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
\end{verbatim}
Data Hazards for Branches

• 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
```
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
I-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
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