PIPELINING

B649
Parallel Architectures and Programming
Why Pipelining?

• Instruction-Level Parallelism (ILP)
• Reducing Cycles Per Instruction (CPI)
  ★ if instructions may take multiple cycles

\[
u = \text{Time per instruction on unpipelined machine} \\
n = \text{Number of pipelined stages} \\
time\text{ per pipelined instruction} = \frac{u}{n}
\]

• Decreasing the clock cycle time
  ★ if each instruction takes one (long) cycle
• Invisible to the programmer
Basics of a RISC Instruction Set

• All operations on data registers
• Only load and store access memory
• All instructions are of one (fixed) size
• MIPS64 (64-bit) instructions for example
Instruction Set Overview

• ALU instructions
  ★ R₁ ← R₂ op R₃
  * R₁, R₂, R₃: registers
  ★ R₁ ← R₂ op I
  * I: signed extended 16-bit immediate

• Load and store instructions
  ★ LD R₁:O, R₂
  * R₁, R₂: registers, O: 16-bit signed extended 16-bit immediate

• Branch and jump instructions
  ★ comparison between two registers, or register and zero
  ★ no unconditional jump
Digression: Recall Multiplexer
Digression: Sign Extension

• Positive Number: extend with zeroes

\[
x \text{ (16 bits)} = \begin{array}{c}
0 \\
\hline
x
\end{array}
\]

\[
x \text{ (32 bits)} = \begin{array}{c}
000000000000000000000000000000 \\
\hline
x
\end{array}
\]

• Negative number: extend with ones

\[
-x \text{ (16 bits)} = \begin{array}{c}
1 \\
\hline
x
\end{array}
\]

\[
-x \text{ (32 bits)} = \begin{array}{c}
111111111111111111111111111111 \\
\hline
x
\end{array}
\]
Digression: Sign Extension

• Positive Number: extend with zeroes

\[
x (16 \text{ bits}) = \begin{array}{c}
0 \ 00000000000000000000000000000000 \\
\end{array}
\]

\[
x (32 \text{ bits}) = \begin{array}{c}
0 \ 00000000000000000000000000000000 \\
\end{array}
\]

• Negative number: extend with ones

\[
-x (16 \text{ bits}) = \begin{array}{c}
1 \\
\end{array}
\]

\[
-x (32 \text{ bits}) = \begin{array}{c}
1 \\
\end{array}
\]

-x (16 bits) = (2^{16} - x)
-x (extended) = (2^{16} - x) + 2^{16}(2^{16} - 1)
= (2^{32} - x)
= -x (32 bits)
Simple Implementation

- **IF**: Instruction fetch cycle
- **ID**: Instruction decode / register fetch cycle
- **EX**: Execute / effective address cycle
- **MEM**: Memory access cycle
- **WB**: Write-back cycle
Simple Implementation

- **IF**: Instruction fetch cycle
  - Fetch current instruction from PC, add 4 to PC
- **ID**: Instruction decode / register fetch cycle
- **EX**: Execute / effective address cycle
- **MEM**: Memory access cycle
- **WB**: Write-back cycle
Simple Implementation

• IF: Instruction fetch cycle
• ID: Instruction decode / register fetch cycle
  ★ decode instruction, read registers \((\text{fixed field decoding})\)
  ★ do equality test on registers for possible branch
  ★ sign extend offset field, in case it is needed
  ★ add offset to possible branch target address
• EX: Execute / effective address cycle
• MEM: Memory access cycle
• WB: Write-back cycle
Simple Implementation

• IF: Instruction fetch cycle
• ID: Instruction decode / register fetch cycle
• EX: Execute / effective address cycle
  ★ memory reference: base address + offset to compute effective address
  ★ register-register ALU instruction: perform the operation
  ★ register-immediate ALU instruction: perform the operation
• MEM: Memory access cycle
• WB: Write-back cycle
Simple Implementation

- IF: Instruction fetch cycle
- ID: Instruction decode / register fetch cycle
- EX: Execute / effective address cycle
- MEM: Memory access cycle
  ★ read or write based on effective address computed in last cycle
- WB: Write-back cycle
Simple Implementation

- IF: Instruction fetch cycle
- ID: Instruction decode / register fetch cycle
- EX: Execute / effective address cycle
- MEM: Memory access cycle
- WB: Write-back cycle

* register-register ALU instruction or Load: write result (computed or loaded from memory) into register file
Simple Implementation

- **IF**: Instruction fetch cycle
- **ID**: Instruction decode / register fetch cycle
- **EX**: Execute / effective address cycle
- **MEM**: Memory access cycle
- **WB**: Write-back cycle

branch = 2 cycles
store = 4 cycles
all else = 5 cycles
CPI = 4.54, assuming 12% branches, 10% stores
Simple Pipelined Implementation
Some Considerations

• Resource evaluation
  ★ avoid resource conflicts across stages

• Separate instruction and data memories
  ★ typically, with separate I and D caches

• Register access
  ★ write in first half, read in second half

• PC not shown
  ★ also need an adder to compute branch target
  ★ branch does not change PC until ID (second) stage
  ★ ignore for now!
Prevent Interference
Observations

- Each instruction takes the same number of cycles
- Instruction throughput increases
  - hence programs run faster
- Imbalance among pipeline stages reduces performance
- Overheads
  - pipeline delays (register setup time)
  - clock skew (clock cycle ≥ clock skew + latch overhead)
- Hazards ahead!
Pipeline Hazards

• Structural hazards
  ★ not all instruction combinations possible in parallel

• Data hazards
  ★ data dependence

• Control hazards
  ★ control dependence
Pipeline Hazards

- Structural hazards
  - not all instruction combinations possible in parallel
- Data hazards
  - data dependence
- Control hazards
  - control dependence

Hazards make it necessary to **stall** the pipeline
Quantifying the Stall Cost

Average instruction time unpipelined
\[
\text{Speedup} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}
\]

CPI unpipelined × Clock cycle unpipelined
\[
= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}
\]

CPI pipelined = Ideal CPI + Pipeline stall cycles per instruction
\[
= 1 + \text{Pipeline stall cycles per instruction}
\]

Ignoring pipeline overheads, assuming balanced stages,

Clock cycle unpipelined = Clock cycle pipelined

CPI Unpipelined (≈ Pipeline depth)
\[
\text{Speedup} = \frac{\text{CPI Unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}
\]
STRUCTURAL HAZARDS
Structural Hazard Example: Mem. Port Conflict
DATA HAZARDS
Data Hazard Types

• RAW: Read After Write
  ★ true dependence
• WAR: Write After Read
  ★ anti-dependence
• WAR: Write After Write
  ★ output dependence
• RAR: Read After Read
  ★ input dependence
Data Hazard Types

• RAW: Read After Write
  ★ true dependence
• WAR: Write After Read
  ★ anti-dependence
• WAR: Write After Write
  ★ output dependence
• RAR: Read After Read
  ★ input dependence
Data Hazard Example

Program execution order (in instructions): IM, Reg, ALU, DM, Reg

Instruction schedule:
- DADD R1, R2, R3
- DSUB R4, R1, R5
- AND R6, R1, R7
- OR R8, R1, R9
- XOR R10, R1, R11

Time (in clock cycles): CC 1 to CC 6
Ameliorating Data Hazards

- Idea:
  - ALU results from EX/MEM and MEM/WB registers fed back to ALU inputs
  - if previous ALU operation wrote the register needed by the current operation, select the forwarded result
Forwarding

Time (in clock cycles)

CC 1  CC 2  CC 3  CC 4  CC 5  CC 6

DADD R1, R2, R3

DSUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

XOR R10, R1, R11

© 2007 Elsevier, Inc. All rights reserved.
Ameliorating Data Hazards

• Idea:
  ★ ALU results from EX/MEM and MEM/WB registers fed back to ALU inputs
  ★ if previous ALU operation wrote the register needed by the current operation, select the forwarded result

• Observations:
  ★ forwarding needed across multiple cycles (how many?)
  ★ forwarding may be implemented across functional units
    ★ e.g., output of one unit may be forwarded to input of another, rather than the input of just the same unit
Forwarding Across Multiple Units

Diagram showing the execution order and forwarding across multiple execution units (CC 1 to CC 6) for different instructions: DADD R1, R2, R3, LD R4, 0(R1), and SD R4, 12(R1). The diagram illustrates how data is forwarded between the IM (Instruction Memory), Reg (Register), ALU (Arithmetic Logic Unit), and DM (Data Memory) units across different clock cycles.
Not All Stalls Avoided
CONTROL HAZARDS
Handling Branch Hazard

<table>
<thead>
<tr>
<th>Branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch successor</td>
<td>IF</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>Branch successor + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>Branch successor + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reducing Branch Penalty

• “Freeze” or “flush” the pipeline
• Treat every branch as not-taken (“predicted-untaken”)
  ★ need to handle taken branches by roll-back
• Treat every branch as taken (“predicted-taken”)
• Delayed branch
### Freezing Pipeline

<table>
<thead>
<tr>
<th>Untaken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction $i + 1$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction $i + 2$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction $i + 3$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction $i + 4$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Taken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction $i + 1$</td>
<td>IF</td>
<td><strong>idle</strong></td>
<td><strong>idle</strong></td>
<td><strong>idle</strong></td>
<td><strong>idle</strong></td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
Delayed Branch

branch instruction
sequential successor_1
branch target if taken
### Behavior of Delayed Branch

<table>
<thead>
<tr>
<th>Untaken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch delay instruction ((i + 1))</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 3)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 4)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Taken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch delay instruction ((i + 1))</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
Schedule of Branch Delay Slot

(a) From before
DADD R1, R2, R3
if R2 = 0 then
Delay slot
becomes
if R2 = 0 then
DADD R1, R2, R3

(b) From target
DSUB R4, R5, R6
DADD R1, R2, R3
if R1 = 0 then
Delay slot
becomes
DSUB R4, R5, R6
DADD R1, R2, R3

(c) From fall-through
DADD R1, R2, R3
if R1 = 0 then
Delay slot
OR R7, R8, R9
DSUB R4, R5, R6
becomes
DADD R1, R2, R3
if R1 = 0 then
DSUB R4, R5, R6
OR R7, R8, R9
DSUB R4, R5, R6
Schedule of Branch Delay Slot

(a) From before
DADD R1, R2, R3
if R2 = 0 then

Delay slot

becomes

if R2 = 0 then
DADD R1, R2, R3

(b) From target
DSUB R4, R5, R6

DADD R1, R2, R3
if R1 = 0 then

Delay slot

becomes

DSUB R4, R5, R6

(c) From fall-through
DADD R1, R2, R3
if R1 = 0 then

Delay slot

OR R7, R8, R9

DSUB R4, R5, R6

becomes

DADD R1, R2, R3
if R1 = 0 then

OR R7, R8, R9

DSUB R4, R5, R6

© 2007 Elsevier, Inc. All rights reserved.
HOW IS PIPELINING IMPLEMENTED?
Simple MIPS Implementation

- Instruction fetch cycle (IF)
- Instruction decode/register fetch cycle (ID)
- Execution / effective address cycle (EX)
- Memory access / branch completion cycle (MEM)
- Write-back cycle (WB)
Simple MIPS Implementation: IF
(IF ➔ ID ➔ EX ➔ MEM ➔ WB)

• Fetch

IR ← Mem[PC];
NPC ← PC + 4;
Simple MIPS Implementation: ID
(IF $\rightarrow$ ID $\rightarrow$ EX $\rightarrow$ MEM $\rightarrow$ WB)

• Decode

\[
\begin{align*}
A & \leftarrow \text{Regs}[rs]; \\
B & \leftarrow \text{Regs}[rt]; \\
\text{Imm} & \leftarrow \text{sign-extended immediate field of IR};
\end{align*}
\]
Simple MIPS Implementation: ID
(IF ➔ ID ➔ EX ➔ MEM ➔ WB)

• Execution

★ Memory reference

\[
\text{ALUOutput} \leftarrow A + \text{Imm};
\]

★ Register/Register ALU instruction

\[
\text{ALUOutput} \leftarrow A \ \text{func} \ B;
\]

★ Register-Immediate ALU instruction

\[
\text{ALUOutput} \leftarrow A \ \text{op} \ \text{Imm};
\]

★ Branch

\[
\text{ALUOutput} \leftarrow \text{NPC} + (\text{Imm} \ll 2);
\text{Cond} \leftarrow (A == 0);
\]
Simple MIPS Implementation: ID
(\text{IF} \rightarrow \text{ID} \rightarrow \text{EX} \rightarrow \text{MEM} \rightarrow \text{WB})

• Memory access / branch completion

★ Memory reference

\begin{align*}
\text{LMD} & \leftarrow \text{Mem[ALUOutput]} \text{ or } \\
\text{Mem[ALUOutput]} & \leftarrow B;
\end{align*}

★ Branch

\begin{align*}
\text{if (cond) PC} & \leftarrow \text{ALUOutput};
\end{align*}
Simple MIPS Implementation: ID
(IF ➔ ID ➔ EX ➔ MEM ➔ WB)

• Write-back
  ★ Register-Register ALU instruction
  \[
  \text{Regs}[rd] \leftarrow \text{ALUOutput};
  \]
  ★ Register-Immediate ALU instruction
  \[
  \text{Regs}[rt] \leftarrow \text{ALUOutput};
  \]
  ★ Load instruction
  \[
  \text{Regs}[rt] \leftarrow \text{LMD};
  \]
MIPS Data Path
MIPS Data Path: Pipelined
# Situations for Data Hazard

<table>
<thead>
<tr>
<th>Situation</th>
<th>Example code sequence</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>No dependence</td>
<td>LD ( \textbf{R1}, 45(R2) ) DADD ( R5, R6, R7 ) DSUB ( R8, R6, R7 ) OR ( R9, R6, R7 )</td>
<td>No hazard possible because no dependence exists on R1 in the immediately following three instructions.</td>
</tr>
<tr>
<td>Dependence requiring stall</td>
<td>LD ( \textbf{R1}, 45(R2) ) DADD ( R5, R1, R7 ) DSUB ( R8, R6, R7 ) OR ( R9, R6, R7 )</td>
<td>Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX.</td>
</tr>
<tr>
<td>Dependence overcome by forwarding</td>
<td>LD ( \textbf{R1}, 45(R2) ) DADD ( R5, R6, R7 ) DSUB ( R8, R1, R7 ) OR ( R9, R6, R7 )</td>
<td>Comparators detect use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX.</td>
</tr>
<tr>
<td>Dependence with accesses in order</td>
<td>LD ( \textbf{R1}, 45(R2) ) DADD ( R5, R6, R7 ) DSUB ( R8, R6, R7 ) OR ( R9, R1, R7 )</td>
<td>No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.</td>
</tr>
</tbody>
</table>
Logic to Detect Data Hazards

<table>
<thead>
<tr>
<th>Opcode field of ID/EX (ID/EX.IR&lt;sub&gt;.0..5&lt;/sub&gt;)</th>
<th>Opcode field of IF/ID (IF/ID.IR&lt;sub&gt;.0..5&lt;/sub&gt;)</th>
<th>Matching operand fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Register-register ALU</td>
<td>ID/EX.IR&lt;sub&gt;[rt]&lt;/sub&gt; == IF/ID.IR&lt;sub&gt;[rs]&lt;/sub&gt;</td>
</tr>
<tr>
<td>Load</td>
<td>Register-register ALU</td>
<td>ID/EX.IR&lt;sub&gt;[rt]&lt;/sub&gt; == IF/ID.IR&lt;sub&gt;[rt]&lt;/sub&gt;</td>
</tr>
<tr>
<td>Load</td>
<td>Load, store, ALU immediate, or branch</td>
<td>ID/EX.IR&lt;sub&gt;[rt]&lt;/sub&gt; == IF/ID.IR&lt;sub&gt;[rs]&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
## Forwarding

<table>
<thead>
<tr>
<th>Pipeline register containing source instruction</th>
<th>Opcode of source instruction</th>
<th>Pipeline register containing destination instruction</th>
<th>Opcode of destination instruction</th>
<th>Destination of the forwarded result</th>
<th>Comparison (if equal then forward)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX/MEM</td>
<td>Register-register ALU</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>EX/MEM.IR[rd] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>EX/MEM.IR[rt] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>EX/MEM.IR[rt] == ID/EX.IR[rt]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>MEM/WB.IR[rt] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>MEM/WB.IR[rt] == ID/EX.IR[rt]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Load</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>MEM/WB.IR[rt] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Load</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>MEM/WB.IR[rt] == ID/EX.IR[rt]</td>
</tr>
</tbody>
</table>
### Forwarding

<table>
<thead>
<tr>
<th>Pipeline register containing source instruction</th>
<th>Opcode of source instruction</th>
<th>Pipeline register containing destination instruction</th>
<th>Opcode of destination instruction</th>
<th>Destination of the forwarded result</th>
<th>Comparison (if equal then forward)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX/MEM</td>
<td>Register-register ALU</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>EX/MEM.IR[rd] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>EX/MEM.IR[rt] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>EX/MEM.IR[rt] == ID/EX.IR[rt]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>MEM/WB.IR[rt] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>MEM/WB.IR[rt] == ID/EX.IR[rt]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Load</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>MEM/WB.IR[rt] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Load</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>MEM/WB.IR[rt] == ID/EX.IR[rt]</td>
</tr>
</tbody>
</table>
Forwarding Implemented
Reducing the Branch Delay
Types of Exceptions

- I/O device request
- Invoking an OS service
- Tracing
- Breakpoint
- Integer arithmetic overflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory access
- Memory protection violation
- Undefined or unimplemented instruction
- Hardware malfunction
- Power failure
Exception Categories

- Synchronous vs Asynchronous
- User requested vs coerced
- User maskable vs nonmaskable
- Within vs between instructions
- Resume vs terminate
### Exception Categorization

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Synchronous vs. asynchronous</th>
<th>User request vs. coerced</th>
<th>User maskable vs. nonmaskable</th>
<th>Within vs. between instructions</th>
<th>Resume vs. terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O device request</td>
<td>Asynchronous</td>
<td>Coerced</td>
<td>Nonmaskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Invoke operating system</td>
<td>Synchronous</td>
<td>User request</td>
<td>Nonmaskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Tracing instruction execution</td>
<td>Synchronous</td>
<td>User request</td>
<td>User maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>Synchronous</td>
<td>User request</td>
<td>User maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Integer arithmetic overflow</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>User maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Floating-point arithmetic overflow or underflow</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>User maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Page fault</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>Nonmaskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Misaligned memory accesses</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>User maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Memory protection violations</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>Nonmaskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Using undefined instructions</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>Nonmaskable</td>
<td>Within</td>
<td>Terminate</td>
</tr>
<tr>
<td>Hardware malfunctions</td>
<td>Asynchronous</td>
<td>Coerced</td>
<td>Nonmaskable</td>
<td>Within</td>
<td>Terminate</td>
</tr>
<tr>
<td>Power failure</td>
<td>Asynchronous</td>
<td>Coerced</td>
<td>Nonmaskable</td>
<td>Within</td>
<td>Terminate</td>
</tr>
</tbody>
</table>
Handling Exceptions

• Force a “trap” into the pipeline on the next IF
• Turn of all writes until the trap is taken off
  ★ place zeros into pipeline latches of instructions following the one causing exception
• Exception handler saves PC and returns to it
• Delayed branches cause a problem
  ★ need to save delay slots plus one number of PCs
## (Precise) Exceptions in MIPS

<table>
<thead>
<tr>
<th>Pipeline stage</th>
<th>Problem exceptions occurring</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on instruction fetch; misaligned memory access; memory protection violation</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined or illegal opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic exception</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data fetch; misaligned memory access; memory protection violation</td>
</tr>
<tr>
<td>WB</td>
<td>None</td>
</tr>
</tbody>
</table>
### (Precise) Exceptions in MIPS

<table>
<thead>
<tr>
<th>Pipeline stage</th>
<th>Problem exceptions occurring</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on instruction fetch; misaligned memory access; memory protection violation</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined or illegal opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic exception</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data fetch; misaligned memory access; memory protection violation</td>
</tr>
<tr>
<td>WB</td>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LD</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DADD</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
(Precise) Exceptions in MIPS

<table>
<thead>
<tr>
<th>Pipeline stage</th>
<th>Problem exceptions occurring</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on instruction fetch; misaligned memory access; memory protection violation</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined or illegal opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic exception</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data fetch; misaligned memory access; memory protection violation</td>
</tr>
<tr>
<td>WB</td>
<td>None</td>
</tr>
</tbody>
</table>

Use exception status vector for each instruction
Complications due to Complex Instructions

• Instructions that change processor states before they are committed
  ★ autoincrement
  ★ string copy (change memory state)

• State bits
  ★ implicitly condition codes (flags)
    ★ instructions setting condition codes not allowed in delay slots

• Multicycle operations
  ★ use of microinstructions
ADDING FLOATING POINT SUPPORT
Functional Units

- Main integer unit
  - handles loads, stores, integer ALU operations, branches
- FP and integer multiplier
- FP adder
  - handles FP add, subtract, and conversion
- FP and integer divider
Extended Pipeline: Expanded View
Extended Pipeline: Expanded View
Extended Pipeline: Expanded View
Additional Complications

- Structural hazard because divide unit is not pipelined
- Number of register writes in a cycle may more than one
- WAW hazards possible
  - WAR hazards not possible
- Maintaining precise exceptions
  - out-of-order completion
- More number of stalls due to RAW hazards, due to longer pipeline
# Example: RAW Hazard

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F4,0(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL.D F0,F4,F6</td>
<td>IF</td>
<td>ID</td>
<td>stall</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>M7</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D F2,F0,F8</td>
<td>IF</td>
<td>stall</td>
<td>ID</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F2,0(R2)</td>
<td>IF</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>ID</td>
<td>EX</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Example 2

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL.D F0,F4,F6</td>
<td>IF</td>
<td>ID</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>M7</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D F2,F4,F6</td>
<td>IF</td>
<td>ID</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F2,0(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Handling the Problems

• Register file conflict detection
  ★ detect at ID stage
  ★ detect at MEM or WB stage
Handling the Problems

• Register file conflict detection
  ★ detect at ID stage, or
  ★ detect at MEM or WB stage

• WAW hazard detection
  ★ delay the issue of second instruction, or
  ★ stamp out the result of the first instruction (convert it into noop)
Summary of Checks

- Check for structural hazards
- Check for RAW data hazard
- Check for WAW data hazard
Maintaining Precise Exceptions

- Ignore the problem
  - settle for imprecise exceptions

- Buffer the results
  - simple buffer could be very big
  - history file
  - future file

- Let trap handling routines clean up

- Hybrid scheme: allow issue when earlier instructions can no longer raise exception
DYNAMIC SCHEDULING
Idea Behind Dynamic Scheduling

• Split ID stage:
  ★ Issue -- decode, check for structural hazards
  ★ Read operands -- wait until no data hazards, then read

• Other stages remain as before
Idea Behind Dynamic Scheduling

• Split ID stage:
  ★ Issue -- decode, check for structural hazards
  ★ Read operands -- wait until no data hazards, then read

• Other stages remain as before

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register 1</th>
<th>Register 2</th>
<th>Register 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIV.D</td>
<td>F0, F2, F4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F10, F0, F8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB.D</td>
<td>F8, F8, F14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Must take care of WAR and WAW hazards.
Scoreboarding (CDC 6600)
Scoreboarding (CDC 6600)

• Issue
  ★ check for free functional units
  ★ check if another instruction has the same destination register (WAW hazard detection)

• Read operands
  ★ monitor availability of source operands (RAW hazard detection)

• Execution
  ★ functional unit notifies scoreboard of completion

• Write result
  ★ check for WAR hazards, stall write if necessary
Scoreboarding: Effectiveness

• Relative easy to implement the logic
  ★ only as much as a functional unit
  ★ but four times as many buses as without scoreboard

• Reduces Clocks Per Instruction (CPI)
  ★ tries to make use of the available Instruction Level Parallelism (ILP)
  ★ 1.7x speedup for Fortran, 2.5x for hand-coded assembly
Scoreboarding: Limitations

• Overlapping instructions must be picked from a single basic block
• Window: number of scoreboard entries
• Number and types of functional units
• Presence of anti- and output-dependences
Pitfalls

• Unexpected execution sequences may cause unexpected hazards

```
BNEZ   R1, foo
DIV.D  F0,F2,F4
...
...
foo: L.D  F0,qrs
```

• Extensive pipelining can impact other aspects of a design

• Evaluating dynamic or static scheduling on the basis of unoptimized code