MIPS Instruction pipeline

- Five-stage pipeline: IF, ID, EX, MEM, WB
- Focus on Data hazards

Three Generic Data Hazards - RAW

♦ Read After Write (RAW)
  Instr\textsubscript{J} tries to read operand before Instr\textsubscript{I} writes it

I: \texttt{add r1,r2,r3}
J: \texttt{sub r4,r1,r3}

♦ Caused by a “Dependence” (in compiler nomenclature).
  This hazard results from an actual need for communication

---

Data Hazard on R1

Time (clock cycles)

\begin{tabular}{|c|c|c|c|c|}
\hline
Instr. Order & IF & ID/RF & EX & MEM & WB \\
\hline
add r1,r2,r3 & Fetch & Reg & ALU & Reg & Mem \\
sub r4,r1,r3 & Fetch & Reg & ALU & Mem & Reg \\
and r6,r1,r7 & Fetch & Reg & ALU & Mem & Reg \\
or r8,r1,r9 & Fetch & Reg & ALU & Mem & Reg \\
xor r10,r1,r11 & Fetch & Reg & ALU & Mem & Reg \\
\hline
\end{tabular}
2nd Generic Data Hazard - WAR

- **Write After Read (WAR)**
  Instr\textsubscript{J} writes operand before Instr\textsubscript{I} reads it.

\[
\begin{align*}
  I: & \text{div r4, r1, r3} \\
  J: & \text{add r1, r2, r3} \\
  K: & \text{mul r6, r1, r7}
\end{align*}
\]

- Called an “anti-dependence” by compiler writers. This results from reuse of the register r1.

- Can not happen in MIPS 5-stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2 (ID/RF), and
  - Writes are always in stage 5 (WB)

---

3rd Generic Data Hazard - WAW

- **Write After Write (WAW)**
  Instr\textsubscript{J} writes operand before Instr\textsubscript{I} writes it.

\[
\begin{align*}
  I: & \text{div r1, r4, r3} \\
  J: & \text{add r1, r2, r3} \\
  K: & \text{mul r6, r1, r7}
\end{align*}
\]

- Called an “output dependence” by compiler writers. This also results from the reuse of name “r1”.

- Can not happen in MIPS 5-stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5

- Will see WAR and WAW in later more complicated pipes (out-of-order execution)
Dealing with data hazards

♦ Must first detect
♦ 1. Introduce stalls (bubbles)
   • Capability to stall instructions and then release when it becomes safe to do so
♦ 2. Internal data forwarding
♦ 3. Reschedule instructions
   • Static rescheduling - by the compiler
   • Dynamic rescheduling - by the hardware
♦ A combination of the above

Interlock Control Logic

Compare source registers of instruction in ID to destination register of previous instructions.
Should we always stall if the rs field matches some rd?
not every instruction writes a register ⇒ we or reads ⇒ re
## Source & Destination Registers

**R-type:**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>func</th>
</tr>
</thead>
</table>

**I-type:**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate 16</th>
</tr>
</thead>
</table>

**J-type:**

<table>
<thead>
<tr>
<th>op</th>
<th>immediate 26</th>
<th>source(s)</th>
<th>destination</th>
</tr>
</thead>
</table>

- **ALU**  
  \[ \text{rd} \leftarrow (\text{rs}) \text{ func (rt)} \]  
  rs, rt rd

- **ALUi**  
  \[ \text{rs} \leftarrow (\text{rs}) \text{ op imm} \]  
  rs rt

- **LW**  
  \[ \text{rt} \leftarrow \text{M}[(\text{rs}) + \text{imm}] \]  
  rs rt

- **SW**  
  \[ \text{M}[(\text{rs}) + \text{imm}] \leftarrow (\text{rt}) \]  
  rs rt

- **BZ**  
  \[ \text{cond (rs)} \]  
  \[ \begin{align*} 
  \text{true: } & \text{PC} \leftarrow (\text{PC}) + \text{imm} \text{ rs} \\
  \text{false: } & \text{PC} \leftarrow (\text{PC}) + 4 \text{ rs} 
  \end{align*} \]

- **J**  
  \[ \text{PC} \leftarrow (\text{PC}) + \text{imm} \]

- **JAL**  
  \[ r31 \leftarrow (\text{PC}), \text{PC} \leftarrow (\text{PC}) + \text{imm} \text{ 31} \]

- **JR**  
  \[ \text{PC} \leftarrow (\text{rs}) \text{ rs} \]

- **JALR**  
  \[ r31 \leftarrow (\text{PC}), \text{PC} \leftarrow (\text{rs}) \text{ rs} \text{ 31} \]

## Load & Store Hazards

However, the hazard is avoided because our memory system completes writes in one cycle.
Forwarding to Avoid RAW Data Hazard

Time (clock cycles)

Instr. Order

add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11

HW Change for Forwarding
Data Hazard Even with Forwarding

Time (clock cycles)

Instr. Order

lw r1, 0(r2)  
add r4, r1, r5  
sub r6, r1, r7  
or r8, r1, r9

Copyright UCB & Morgan Kaufmann ECE568/Koren Part 3.13  Adapte from UCB and other sources  Copyright UCB & Morgan Kaufmann
# Detecting pipeline data hazards after lw

<table>
<thead>
<tr>
<th>Situation</th>
<th>Example code sequence</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>No dependence</td>
<td>LW R1, 45 (R2) ADD R5, R6, R7 SUB 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>LW R1, 45 (R2) ADD R5, R1, R7 SUB R8, R6, R7 OR R9, R6, R7</td>
<td>Comparators detect the use of R1 in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX.</td>
</tr>
<tr>
<td>Dependence overcome by forwarding</td>
<td>LW R1, 45 (R2) ADD R5, R6, R7 SUB R8, R1, R7 OR R9, R6, R7</td>
<td>Comparators detect use of R1 in SUB and forward result of load to ALU in time for SUB to begin EX.</td>
</tr>
<tr>
<td>Dependence with accesses in order</td>
<td>LW R1, 45 (R2) ADD R5, R6, R7 SUB 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>

---

## MIPS R4000

![MIPS R4000 Diagram]

- Instruction cache
- Instruction fetch
- Instruction address translation
- Instruction decode
- Read register file
- Fetch register file
- Decode
- ALU operation
- Data address check
- Data cache access
- Write to register file
- Data tag check
- Execution
- Instruction fetch
- Instruction second
- RS: Instruction second
- RF: Register file
- EX: Execution
- TC: Tag check
- WB: Write back
Pipelining with stalls

\[ CPI_{\text{pipelined}} = CPI_{\text{ideal}} + \text{Average Stall cycles per Inst} \]

\[ \text{Speedup} = \text{Pipeline depth} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}} \times \frac{CPI_{\text{ideal}}}{CPI + \text{Avg. stall CPI}} \]

For simple RISC pipeline, \( CPI_{\text{ideal}} = 1 \):

\[ \text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Average stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}} \]
Benefits of forwarding (LOAD only)

\[
\text{Speedup} = \frac{\text{Pipeline_depth}}{1 + \text{Stall_CPI}} \times \frac{T_{\text{unpipelined}}}{T_{\text{pipelined}}}
\]

Assume: 30% Load instr., 50% of the time the following instr. needs the data, 30% of the time only the 2nd instr. needs the data; w/o forwarding the 1st instr. requires 2 stall cycles, the 2nd requires 1 stall cycle.

\[
\text{Speedup}_{\text{w/forwarding}} = \frac{5}{1 + 1 \times 0.3 \times 0.5} \times \frac{1}{1.05} = 4.14
\]

\[
\text{Speedup}_{\text{w/o forwarding}} = \frac{5}{1 + 0.3(\text{something})} \times \frac{1}{1.05} = 3.43
\]

Speedup due to forwarding = 4.14/3.43 = 1.2

Benefits of forwarding (All instructions)

Assume: Remaining 50% Not-Load instr. (not BR), 20% of the time the instr. following it needs the data, 10% of the time only the 2nd instr. needs the data; w/o forwarding the 1st instr. requires 2 stall cycles, the 2nd requires 1 stall cycle.

\[
\text{Speedup}_{\text{w/o forwarding}} = \frac{5}{1 + 0.39} \times \frac{1}{1.05} = 2.9
\]

Speedup due to forwarding = 4.14/2.9 = 1.43
Software Scheduling to Avoid Load Hazards

Try producing fast code for

\[ a = b + c; \]
\[ d = e - f; \]

assuming \( a, b, c, d, e, \) and \( f \) in memory.

Slow code:

\[
\begin{align*}
\text{LW} & \quad Rb, b \\
\text{LW} & \quad Rc, c \\
\text{ADD} & \quad Ra, Rb, Rc \\
\text{SW} & \quad a, Ra \\
\text{LW} & \quad Re, e \\
\text{ADD} & \quad Ra, Rb, Rc \\
\text{LW} & \quad Rf, f \\
\text{SW} & \quad a, Ra \\
\text{SW} & \quad d, Rd
\end{align*}
\]

Fast code:

\[
\begin{align*}
\text{LW} & \quad Rb, b \\
\text{LW} & \quad Rc, c \\
\text{LW} & \quad Re, e \\
\text{ADD} & \quad Ra, Rb, Rc \\
\text{LW} & \quad Rf, f \\
\text{SW} & \quad a, Ra \\
\text{SW} & \quad d, Rd
\end{align*}
\]

Floating-point operations cannot complete in 1 cycle

Assume:

\begin{align*}
\text{OP} & \quad \#\text{cycles} \\
\text{Int. op} & \quad 1 \\
\text{FP. Add} & \quad 4 \\
\text{FP. Mult} & \quad 7 \\
\text{FP. Div} & \quad 24
\end{align*}
Multi-cycle operation - w/o & w/pipelining

Multi-cycle - Hazards

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

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

FP adder & unipipelined multiplier taking 1 & 3 cycles, respectively

I1: F1 ← F2 + F3
I2: F2 ← F4 x F5
I3: F3 ← F3 + F4
I4: F6 ← F6 x F6
I5: F1 ← F3 + F5
I6: F2 ← F3 + F4

Dependency graph

T(seq_ex) = ?
T(Ideal) = 10
Use WinDLX (flt1.s)

Floating-point stages configuration:

<table>
<thead>
<tr>
<th>Unit</th>
<th>No.</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adders</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Mpiers</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Software Static Scheduling - Multi-cycle

\[ X = Y + Z \quad \& \quad A = B \times C \]

| In-order instruction issuing | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 |
| \( R_1 \leftarrow (Y) \)    | F | D | I | E | E | E | E | W |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| \( R_2 \leftarrow (Z) \)    | F | D | I | E | E | E | E | W |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| \( R_3 \leftarrow (R_1) + (R_2) \) | F | D | * | * | I | E | E | E | W |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| \( X \leftarrow (R_3) \)    | F | * | * | D | * | * | I | E | E | E | W |   |   |   |   |   |   |   |   |   |   |   |   |
| \( R_4 \leftarrow (B) \)    |   | F | * | * | D | I | E | E | E | W |   |   |   |   |   |   |   |   |   |   |   |   |   |
| \( R_5 \leftarrow (C) \)    |   | F | D | I | E | E | E | W |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| \( R_6 \leftarrow (R_4) \times (R_5) \) | F | D | * | * | I | E | E | E | W |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| \( A \leftarrow (R_6) \)    |   | F | * | * | D | * | * | I | E | E | E | W |   |   |   |   |   |   |   |   |   |   |   |

Static Scheduling - Optimizing Compiler

<table>
<thead>
<tr>
<th>Reordered instruction issuing</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>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 \leftarrow (Y) )</td>
<td>F</td>
<td>D</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( R_2 \leftarrow (Z) )</td>
<td>F</td>
<td>D</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( R_4 \leftarrow (B) )</td>
<td>F</td>
<td>D</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( R_5 \leftarrow (C) )</td>
<td>F</td>
<td>D</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( R_3 \leftarrow (R_1) + (R_2) )</td>
<td>F</td>
<td>D</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( R_6 \leftarrow (R_4) \times (R_5) )</td>
<td>F</td>
<td>D</td>
<td>*</td>
<td>*</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( X \leftarrow (R_3) )</td>
<td>F</td>
<td>*</td>
<td>D</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( A \leftarrow (R_6) )</td>
<td>F</td>
<td>D</td>
<td>*</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 16 vs. 22 cycles
- Cheap to implement
- May need NOPs