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

\begin{align*}
\text{I: } & \text{add } r1, r2, r3 \\
\text{J: } & \text{sub } r4, r1, r3
\end{align*}

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

Data Hazard on R1

Time (clock cycles)

\begin{align*}
\text{Instr. Order} & \quad \text{IF} \quad \text{ID/RF} \quad \text{EX} \quad \text{MEM} \quad \text{WB} \\
\text{add } r1, r2, r3 & \quad \text{IF} \quad \text{ID/RF} \quad \text{EX} \quad \text{MEM} \quad \text{WB} \\
\text{sub } r4, r1, r3 & \quad \text{IF} \quad \text{ID/RF} \quad \text{EX} \quad \text{MEM} \quad \text{WB} \\
\text{and } r6, r1, r7 & \quad \text{IF} \quad \text{ID/RF} \quad \text{EX} \quad \text{MEM} \quad \text{WB} \\
\text{or } r8, r1, r9 & \quad \text{IF} \quad \text{ID/RF} \quad \text{EX} \quad \text{MEM} \quad \text{WB} \\
\text{xor } r10, r1, r11 & \quad \text{IF} \quad \text{ID/RF} \quad \text{EX} \quad \text{MEM} \quad \text{WB}
\end{align*}
Three Generic Data Hazards - WAR

♦ Write After Read (WAR)
  Instr\textsubscript{J} writes operand \textbf{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 name “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)

Three Generic Data Hazards - WAW

♦ Write After Write (WAW)
  Instr\textsubscript{J} writes operand \textbf{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

<table>
<thead>
<tr>
<th>R-type:</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-type:</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate 16</td>
<td></td>
</tr>
<tr>
<td>J-type:</td>
<td>op</td>
<td>immediate 26</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source(s)  Destination

ALU\[rd \leftarrow (rs) \text{ func (rt)}\] \(rs, rt\) rd
ALUi\[rt \leftarrow (rs) \op \text{ imm}\] \(rs\) rt
LW\[rt \leftarrow \text{M }[(rs) + \text{ imm}]\] \(rs\) rt
SW\[\text{M }[(rs) + \text{ imm}] \leftarrow (rt)\] \(rs, rt\)
BZ\[\text{ cond (rs)}\]
  \(\text{true: PC }\leftarrow (PC) + \text{ imm}\) rs
  \(\text{false: PC }\leftarrow (PC) + 4\) rs
J\[\text{PC }\leftarrow (PC) + \text{ imm}\]
JAL\[r31 \leftarrow (PC), \text{PC }\leftarrow (PC) + \text{ imm}\] 31
JR\[\text{PC }\leftarrow (rs)\] rs
JALR\[r31 \leftarrow (PC), \text{PC }\leftarrow (rs)\] rs 31

Load & Store Hazards

\[\text{M}[(r1)+7] \leftarrow (r2)\]
\[r4 \leftarrow \text{M}[(r3)+5]\]
\[(r1)+7 = (r3)+5 \Rightarrow \text{data hazard}\]

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 \( r_1, r_2, r_3 \)

sub \( r_4, r_1, r_3 \)

and \( r_6, r_1, r_7 \)

or \( r_8, r_1, r_9 \)

xor \( r_{10}, r_1, r_{11} \)

HW Change for Forwarding

NextPC → Registers → ID/EX → EX/MEM → MEM/WR

Immediate → ID/EX → EX/MEM → MEM/WR

Data Memory

Copyright UCB & Morgan Kaufmann
**Data Hazard Even with Forwarding**

**Time (clock cycles)**

```
Instr. Order
lw r1, 0(r2)                          Ifetch Reg DMem Reg
sub r4, r1, r6                          Ifetch Reg DMem Reg
and r6, r1, r7                          Ifetch Reg DMem Reg
or r8, r1, r9                          Ifetch Reg DMem Reg
```

Copyright UCB & Morgan Kaufmann

*Adapted from UCB and other sources*
Detecting pipeline data hazards

<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

Copyright UCB & Morgan Kaufmann ECE568/Koren Part 3.15 Adapted from UCB and other sources Copyright UCB & Morgan Kaufmann
MIPS R4000 - RAW Hazard

Pipelining with stalls

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

\[ \text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Average stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}} \times \frac{\text{CPI}_{\text{ideal}}}{\text{CPI} + \text{Avg. stall CPI}} \]

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.

\[
\begin{align*}
\text{Speedup}_{\text{w/forwarding}} &= \frac{5}{1 + 1 \times .3 \times .5} \times \frac{1}{1.05} = 4.14 \\
\text{Speedup}_{\text{w/o forward}} &= \frac{5}{1+.3(\text{XX})} \times \frac{1}{1.05} = 3.43
\end{align*}
\]

Speedup due to forwarding = $\frac{4.14}{3.43} = 1.2$

**Benefits of forwarding (All instructions)**

Assume: Remaining 70% Not-Load instr., 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.

\[
\text{Speedup}_{\text{w/o forwarding}} = \frac{5}{1+.39+} \times \frac{1}{1.05} = 2.63
\]

\[
\text{Speedup due to forwarding} = \frac{4.14}{2.63} = 1.57
\]
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:

Fast code:

```
LW Rb,b
LW Rc,c
ADD Ra,Rb,Rc
SW a,Ra
LW Re,e
ADD Ra,Rb,Rc
LW Re,e
SUB Rd,Re,Rf
SW a,Ra
SW d,Rd
SUB Rd,Re,Rf
SW d,Rd
```

Multi-cycle operation - Basics

Floating-point operations can not complete in 1 cycle

Assume:

<table>
<thead>
<tr>
<th>OP</th>
<th>#cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int.op</td>
<td>1</td>
</tr>
<tr>
<td>FP.Add</td>
<td>4</td>
</tr>
<tr>
<td>FP.Mult</td>
<td>7</td>
</tr>
<tr>
<td>FP.Div</td>
<td>24</td>
</tr>
</tbody>
</table>
Multi-cycle operation - w/o & w/pipelining

![Diagram of multi-cycle operation with integer and FP units]

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</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</td>
<td>IF stall stall stall stall stall ID EX stall stall MEM</td>
</tr>
</tbody>
</table>

Clock cycle number

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1  2  3  4  5  6  7  8  9  10  11</th>
</tr>
</thead>
<tbody>
<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</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
Hazards in multi-cycle operation

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

I1: F1 ← F2 + F3
I2: F2 ← F4 × F5
I3: F3 ← F3 + F4
I4: F6 ← F6 × 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>

Finish:

trap 0
Software Static Scheduling - Multi-cycle

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

<table>
<thead>
<tr>
<th>In-order 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>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 \leftarrow (Y) )</td>
<td></td>
<td>F</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>
</tr>
<tr>
<td>( R_2 \leftarrow (Z) )</td>
<td></td>
<td>F</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>
</tr>
<tr>
<td>( R_3 \leftarrow (R_1) + (R_2) )</td>
<td></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>
</tr>
<tr>
<td>( X \leftarrow (R_3) )</td>
<td></td>
<td>F</td>
<td>*</td>
<td>*</td>
<td>D</td>
<td>*</td>
<td>*</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td>( R_4 \leftarrow (B) )</td>
<td></td>
<td>F</td>
<td>*</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>
</tr>
<tr>
<td>( R_5 \leftarrow (C) )</td>
<td></td>
<td>F</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>
</tr>
<tr>
<td>( R_6 \leftarrow (R_4) \times (R_5) )</td>
<td></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>
</tr>
<tr>
<td>( A \leftarrow (R_6) )</td>
<td></td>
<td>F</td>
<td>*</td>
<td>*</td>
<td>D</td>
<td>*</td>
<td>*</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
</tr>
</tbody>
</table>

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>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 \leftarrow (Y) )</td>
<td></td>
<td>F</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>
</tr>
<tr>
<td>( R_2 \leftarrow (Z) )</td>
<td></td>
<td>F</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>
</tr>
<tr>
<td>( R_3 \leftarrow (B) )</td>
<td></td>
<td>F</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>
</tr>
<tr>
<td>( R_5 \leftarrow (C) )</td>
<td></td>
<td>F</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>
</tr>
<tr>
<td>( R_4 \leftarrow (R_1) + (R_2) )</td>
<td></td>
<td>F</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>
</tr>
<tr>
<td>( R_6 \leftarrow (R_4) \times (R_5) )</td>
<td></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>
</tr>
<tr>
<td>( X \leftarrow (R_3) )</td>
<td></td>
<td>F</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>
</tr>
<tr>
<td>( A \leftarrow (R_6) )</td>
<td></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>
</tr>
</tbody>
</table>

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