Instruction Pipelining:
Hazard Resolution, Timing Constraints

Joel Emer
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Pipeline Diagram – Ideal Pipelining

Over long term, i.e., in steady state, what is...

CPI? IPC? Inst exec time?

Num inst in flight?
Reminder: Pipelined MIPS Datapath
without jumps

Pipelining increases clock frequency,
but instruction dependences may increase CPI
How instructions can interact with each other in a pipeline

• An instruction in the pipeline may need a resource being used by another instruction in the pipeline \(\rightarrow\) \textit{structural hazard}

• An instruction may \textbf{depend} on a value produced by an earlier instruction
  
  – Dependence may be for a data calculation \(\rightarrow\) \textit{data hazard}

  – Dependence may be for calculating the next PC \(\rightarrow\) \textit{control hazard (branches, interrupts)}
Data Hazards

\[ r_1 \leftarrow r_0 + 10 \]
\[ r_3 \leftarrow r_1 + 12 \]

\[ r_1 \text{ is stale. Oops!} \]
Pipeline Diagram – Hazard

\[
\begin{align*}
t & \quad t0 & \quad t1 & \quad t2 & \quad t3 & \quad t4 & \quad t5 & \quad t6 & \quad t7 & \quad \ldots \ldots \\
\text{time} & & & & & & & & & & \\
(I_1) \ r1 & \leftarrow & r0 + 10 & & & & & & & & \\
(I_2) \ r3 & \leftarrow & r1 + 12 & & & & & & & & \\
& & IF_1 & & ID_1 & & EX_1 & & MA_1 & & WB_1 & & \\
& & IF_2 & & ID_2 & & EX_2 & & MA_2 & & WB_2 & \\
\end{align*}
\]

\textcolor{red}{r1 is stale. Oops!}
Resolving Data Hazards

Strategy 1: *Wait for the result to be available by freezing earlier pipeline stages* → *stall*

Strategy 2: *Route data as soon as possible after it is calculated to the earlier pipeline stage* → *bypass*

Strategy 3: *Speculate on the dependence*

*Two cases:*
  - Guessed correctly → no special action required
  - Guessed incorrectly → kill and restart
Resolving Data Hazards

Strategy 1: *Wait for the result to be available by freezing earlier pipeline stages* → *stall*
Reminder: Stall Control Logic

ignoring jumps & branches

Stall DEC & IF when instruction in DEC reads a register that is written by any earlier in-flight instruction (in EXE, MEM, or WB)
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>immediate16</th>
</tr>
</thead>
</table>

**J-type:**

<table>
<thead>
<tr>
<th>op</th>
<th>immediate26</th>
</tr>
</thead>
</table>

**source(s) destination**

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

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

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

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

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

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

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

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

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

\[ C_{\text{dest}} \]

\[
ws = \text{Case opcode} \\
\text{ALU} \implies \text{rd} \\
\text{ALU}, \text{LW} \implies \text{rt} \\
\text{JAL}, \text{JALR} \implies \text{R31}
\]

\[
we = \text{Case opcode} \\
\text{ALU, ALU}, \text{LW} \implies (ws \neq 0) \\
\text{JAL, JALR} \implies \text{on} \\
... \implies \text{off}
\]

\[ C_{\text{re}} \]

\[
re1 = \text{Case opcode} \\
\text{ALU, ALU}, \text{LW, SW, BZ, JR, JALR} \implies \text{on} \\
\text{J, JAL} \implies \text{off}
\]

\[
re2 = \text{Case opcode} \\
\text{ALU, SW} \implies \text{on} \\
... \implies \text{off}
\]

\[ C_{\text{stall}} \]

\[
\text{stall} = ((rs_D = ws_E) \cdot we_E + \\
(rs_D = ws_M) \cdot we_M + \\
(rs_D = ws_W) \cdot we_W) \cdot re1_D + \\
((rt_D = ws_E) \cdot we_E + \\
(rt_D = ws_M) \cdot we_M + \\
(rt_D = ws_W) \cdot we_W) \cdot re2_D
\]

This is not the full story!
Is there any possible data hazard in this instruction sequence?

---

```
M[(r1)+7] ← (r2)
r4 ← M[(r3)+5]
```
Load & Store Hazards

However, the hazard is avoided because our memory system completes writes in one cycle!

Load/Store hazards are sometimes resolved in the pipeline and sometimes in the memory system itself.

More on this later in the course.
Resolving Data Hazards (2)

Strategy 2:

Route data as soon as possible after it is calculated to the earlier pipeline stage \(\rightarrow \text{bypass}\)
Bypassing

Each *stall* or *kill* introduces a bubble \( \Rightarrow CPI > 1 \)

*When is data actually available?*

A new datapath, i.e., *a bypass*, can get the data from the output of the ALU to its input
Adding a Bypass

When does this bypass help?

(I_1) \[ r_1 \leftarrow r_0 + 10 \]

(I_2) \[ r_2 \leftarrow r_1 + 12 \]

r_1 \leftarrow M[r_0 + 10]

r_2 \leftarrow r_1 + 12

JAL 500

r_2 \leftarrow r_31 + 12

September 22, 2021
The Bypass Signal
Deriving it from the Stall Signal

\[
\text{stall} = ((\text{rs}_D=\text{ws}_E) \cdot \text{we}_E + (\text{rs}_D=\text{ws}_M) \cdot \text{we}_M + (\text{rs}_D=\text{ws}_W) \cdot \text{we}_W) \cdot \text{re}_{1D}
\]
\[
+ ((\text{rt}_D=\text{ws}_E) \cdot \text{we}_E + (\text{rt}_D=\text{ws}_M) \cdot \text{we}_M + (\text{rt}_D=\text{ws}_W) \cdot \text{we}_W) \cdot \text{re}_{2D}
\]

\[
\text{we} = \text{Case opcode}
\]
\[
\begin{align*}
\text{ALU} & \Rightarrow \text{rd} \\
\text{ALUi, LW} & \Rightarrow \text{rt} \\
\text{JAL, JALR} & \Rightarrow R31
\end{align*}
\]

\[
\text{ws} = \text{Case opcode}
\]
\[
\begin{align*}
\text{ALU, ALUi, LW} & \Rightarrow (\text{ws} \neq 0) \\
\text{JAL, JALR} & \Rightarrow \text{on} \\
\ldots & \Rightarrow \text{off}
\end{align*}
\]

\[
\text{ASrc} = (\text{rs}_D=\text{ws}_E) \cdot \text{we}_E \cdot \text{re}_{1D}
\]

Is this correct?

How might we address this?
Bypass and Stall Signals

Split \( w_{E} \) into two components: \( w_{-bypass} \), \( w_{-stall} \)

\[
\text{we-bypass}_E = \text{Case opcode}_E \\
\text{ALU, ALUi} \quad \Rightarrow (\text{ws} \neq 0) \\
\text{...} \quad \Rightarrow \text{off}
\]

\[
\text{we-stall}_E = \text{Case opcode}_E \\
\text{LW} \quad \Rightarrow (\text{ws} \neq 0) \\
\text{JAL, JALR} \quad \Rightarrow \text{on} \\
\text{...} \quad \Rightarrow \text{off}
\]

\[
\text{ASrc} = (r_{S_D} = = w_{E}) \cdot \text{we-bypass}_E \cdot \text{re}_{1_D}
\]

\[
\text{stall} = ((r_{S_D} = = w_{E}) \cdot \text{we-stall}_E + \\
(r_{S_D} = = w_{M}) \cdot \text{we}_M + (r_{S_D} = = w_{W}) \cdot \text{we}_W) \cdot \text{re}_{1_D}
\]

\[
+ ((r_{T_D} = = w_{E}) \cdot \text{we}_E + (r_{T_D} = = w_{M}) \cdot \text{we}_M + (r_{T_D} = = w_{W}) \cdot \text{we}_W) \cdot \text{re}_{2_D}
\]
Is there still a need for the stall signal?
Resolving Data Hazards (3)

Strategy 3:

Speculate on the dependence. Two cases:

Guessed correctly → no special action required

Guessed incorrectly → kill and restart
Instruction to Instruction Dependence

- **What do we need to calculate next PC?**
  - For Jumps
    - For Jump Register
  - For Conditional Branches
  - For all others

- **In what stage do we know these?**
  - PC →
  - Opcode, offset →
  - Register value →
  - Branch condition ((rs) == 0) →
NextPC Calculation Bubbles

What’s a good guess for next PC?
Speculate NextPC is PC+4

What happens on mis-speculation, i.e., when next instruction is not PC+4?

How?
Pipelining Jumps

To kill a fetched instruction -- Insert a nop in IR

Any interaction between stall and jump?

IRS\_{\text{op}} = \text{Case opcode}_{\text{D}}
\begin{align*}
\text{J, JAL} & \Rightarrow \text{nop} \\
... & \Rightarrow \text{IM}
\end{align*}
Jump Pipeline Diagrams

Resource Usage

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>I₁</td>
<td>I₁</td>
<td>I₂</td>
<td>I₁</td>
<td>I₁</td>
</tr>
<tr>
<td>I₂</td>
<td>I₂</td>
<td>I₃</td>
<td>I₂</td>
<td>I₁</td>
</tr>
<tr>
<td>I₃</td>
<td>I₃</td>
<td>I₄</td>
<td>I₃</td>
<td>I₁</td>
</tr>
<tr>
<td>I₄</td>
<td>I₄</td>
<td>I₅</td>
<td>I₄</td>
<td>I₁</td>
</tr>
<tr>
<td>I₅</td>
<td>I₅</td>
<td>I₅</td>
<td>I₅</td>
<td>I₁</td>
</tr>
<tr>
<td>I₆</td>
<td>I₆</td>
<td>I₆</td>
<td>I₆</td>
<td>I₁</td>
</tr>
</tbody>
</table>

(time)

\[ t₀ \quad t₁ \quad t₂ \quad t₃ \quad t₄ \quad t₅ \quad t₆ \quad t₇ \quad \ldots \]

\[ \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{MA} \quad \text{WB} \]

\[ \begin{align*}
(I₁) & \quad 096: \text{ADD} \\
(I₂) & \quad 100: \text{J 200} \\
(I₃) & \quad 104: \text{ADD} \\
(I₄) & \quad 304: \text{ADD}
\end{align*} \]

\[ \text{nop} \ \Rightarrow \ \text{pipeline bubble} \]
Pipelining Conditional Branches

Branch condition is not known until the execute stage. What action should be taken in the decode stage?

September 22, 2021

MIT 6.5900 (ne 6.823) Fall 2022

L05-26
Pipelining Conditional Branches

If the branch is taken
- kill the two following instructions
- the instruction at the decode stage is not valid
⇒ stall signal is not valid
Pipelining Conditional Branches

If the branch is taken
- kill the two following instructions
- the instruction at the decode stage
  is not valid

⇒ stall signal is not valid
New Stall Signal

\[
stall = ( (rs_D == ws_E) \cdot we_E + (rs_D == ws_M) \cdot we_M + (rs_D == ws_W) \cdot we_W) \cdot re1_D \\
+ ((rt_D == ws_E) \cdot we_E + (rt_D == ws_M) \cdot we_M + (rt_D == ws_W) \cdot we_W) \cdot re2_D \\
\) \cdot !((opcode_E == BEQZ) \cdot z + (opcode_E == BNEZ) \cdot !z)
\]

Don’t stall if the branch is taken. Why?
Control Equations for PC and IR Muxes

\[
\text{IRSrc}_D = \text{Case opcode}_E \\
\quad \text{BEQZ}\cdot z, \text{BNEZ}\cdot !z \quad \Rightarrow \text{nop} \\
\quad \ldots \quad \Rightarrow \quad \text{Case opcode}_D \\
\quad \text{J, JAL, JR, JALR} \quad \Rightarrow \text{nop} \\
\quad \ldots \quad \Rightarrow \text{IM}
\]

\[
\text{IRSrc}_E = \text{Case opcode}_E \\
\quad \text{BEQZ}\cdot z, \text{BNEZ}\cdot !z \quad \Rightarrow \text{nop} \\
\quad \ldots \quad \Rightarrow \quad \text{stall}\cdot \text{nop} + \!\text{stall}\cdot \text{IR}_D
\]

\[
\text{PCSrc} = \text{Case opcode}_E \\
\quad \text{BEQZ}\cdot z, \text{BNEZ}\cdot !z \quad \Rightarrow \text{br} \\
\quad \ldots \quad \Rightarrow \quad \text{Case opcode}_D \\
\quad \text{J, JAL} \quad \Rightarrow \text{jabs} \quad \text{nop} \quad \Rightarrow \text{Kill} \\
\quad \text{JR, JALR} \quad \Rightarrow \text{rind} \\
\quad \ldots \quad \Rightarrow \quad \text{pc}+4 \\
\quad \text{br/jabs/rind} \quad \Rightarrow \text{Restart} \\
\quad \text{pc}+4 \quad \Rightarrow \text{Speculate}
\]

Give priority to the older instruction, i.e., execute stage instruction over decode stage instruction.

\text{pc}+4 \text{ is a speculative guess}
Branch Pipeline Diagrams (resolved in execute stage)

\[
\begin{array}{cccccccc}
\text{time} & t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 \\
(I_1) & 096: \text{ADD} & \text{IF}_1 & \text{ID}_1 & \text{EX}_1 & \text{MA}_1 & \text{WB}_1 \\
(I_2) & 100: \text{BEQZ 200} & \text{IF}_2 & \text{ID}_2 & \text{EX}_2 & \text{MA}_2 & \text{WB}_2 \\
(I_3) & 104: \text{ADD} & \text{IF}_3 & \text{ID}_3 & \text{EX}_3 & \text{MA}_3 & \text{WB}_3 \\
(I_4) & 108: & \text{IF}_4 & \text{ID}_4 & \text{EX}_4 & \text{MA}_4 & \text{WB}_4 \\
(I_5) & 304: \text{ADD} & \text{IF}_5 & \text{ID}_5 & \text{EX}_5 & \text{MA}_5 & \text{WB}_5 \\
\end{array}
\]

\text{Resource Usage}

\[
\begin{array}{cccccccc}
\text{IF} & I_1 & I_2 & I_3 & I_4 & I_5 \\
\text{ID} & I_1 & I_2 & I_3 & I_4 & I_5 \\
\text{EX} & I_1 & I_2 & I_3 & I_4 & I_5 \\
\text{MA} & I_1 & I_2 & I_3 & I_4 & I_5 \\
\text{WB} & I_1 & I_2 & I_3 & I_4 & I_5 \\
\end{array}
\]

\text{nop} \Rightarrow \text{pipeline bubble}
Reducing Branch Penalty (resolve in decode stage)

- One pipeline bubble can be removed if an extra comparator is used in the Decode stage.

Pipeline diagram now same as for jumps
Branch Delay Slots
(expose control hazard to software)

- Change the ISA semantics so that the instruction that follows a jump or branch is always executed
  - gives compiler the flexibility to put in a useful instruction where normally a pipeline bubble would have resulted.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I₁</td>
<td>096</td>
<td>ADD</td>
</tr>
<tr>
<td>I₂</td>
<td>100</td>
<td>BEQZ r1 200</td>
</tr>
<tr>
<td>I₃</td>
<td>104</td>
<td>ADD</td>
</tr>
<tr>
<td>I₄</td>
<td>304</td>
<td>ADD</td>
</tr>
</tbody>
</table>

- Other techniques include branch prediction, which can dramatically reduce the branch penalty... to come later
Handling Control Hazards due to Exceptions

- Instructions may suffer exceptions in different pipeline stages
- Must prioritize exceptions from earlier instructions
• Typical strategy: Record exceptions, process the first one to reach commit point (i.e., the point where architectural state is modified)

  – Pros/cons vs handling exceptions eagerly, like branches?
Why an instruction may not be dispatched every cycle (CPI>1)

- Full bypassing may be too expensive to implement
  - Typically all frequently used paths are provided
  - Some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI

- Loads have two-cycle latency
  - Instruction after load cannot use load result
  - MIPS-I ISA defined load delay slots, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II.

- Conditional branches, jumps, and exceptions may cause bubbles
  - Kill instruction(s) following branch if no delay slots

*Machines with software-visible delay slots may execute significant number of NOP instructions inserted by the compiler.*
Next lecture: Superscalar & Scoreboarded Pipelines