Instruction Pipelining:
Hazard Resolution, Timing Constraints

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Resolving Data Hazards

Strategy 1: Wait for the result to be available by freezing earlier pipeline stages \( \rightarrow \text{stall} \)

Strategy 2: Route data as soon as possible after it is calculated to the earlier pipeline stage \( \rightarrow \text{bypass} \)

Strategy 3: Speculate on the dependence
Two cases:
- Guessed correctly \( \rightarrow \) no special action required
- Guessed incorrectly \( \rightarrow \) kill and restart
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)
**Reminder: Load & Store Hazards**

**Stall Condition**

These hazards do not need pipeline changes because our memory system completes writes in one cycle.
Strategy 2:

Route data as soon as possible after it is calculated to the earlier pipeline stage → 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)  
\[ r1 \leftarrow r0 + 10 \]  yes

(I_2)  
\[ r4 \leftarrow r1 + 17 \]  yes

\[ r1 \leftarrow M[r0 + 10] \]  no

\[ r4 \leftarrow r1 + 17 \]

JAL 500  
\[ r4 \leftarrow r31 + 17 \]  no
The Bypass Signal

Deriving it from the Stall Signal

\[
\text{stall} = ((\text{rs}_D == \text{ws}_E) \cdot \text{we}_E + (\text{rs}_D == \text{ws}_M) \cdot \text{we}_M + (\text{rs}_D == \text{ws}_W) \cdot \text{we}_W) \cdot \text{re1}_D \\
+ ((\text{rt}_D == \text{ws}_E) \cdot \text{we}_E + (\text{rt}_D == \text{ws}_M) \cdot \text{we}_M + (\text{rt}_D == \text{ws}_W) \cdot \text{we}_W) \cdot \text{re2}_D
\]

ws = Case opcode

- ALU ⇒ rd
- ALUi, LW ⇒ rt
- JAL, JALR ⇒ R31

we = Case opcode

- ALU, ALUi, LW ⇒ (ws ≠ 0)
- JAL, JALR ⇒ on
- ... ⇒ off

\[
\text{ASrc} = (\text{rs}_D == \text{ws}_E) \cdot \text{we}_E \cdot \text{re1}_D
\]

Is this correct?

How might we address this?
Bypass and Stall Signals

Split $we_E$ into two components: we-bypass, we-stall

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

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

$$\text{ASrc} = (rs_D == ws_E) \cdot we\text{-bypass}_E \cdot re1_D$$

$$\text{stall} = ((rs_D == ws_E) \cdot we\text{-stall}_E +$$
$$\quad (rs_D == ws_M) \cdot we_M + (rs_D == ws_W) \cdot we_W) \cdot re1_D$$
$$\quad + ((rt_D == ws_E) \cdot we_E + (rt_D == ws_M) \cdot we_M + (rt_D == ws_W) \cdot we_W) \cdot re2_D$$
Fully Bypassed Datapath

Is there still a need for the stall signal?
Strategy 3:

Speculate on the dependence. Two cases:

- Guessed correctly $\rightarrow$ no special action required
- Guessed incorrectly $\rightarrow$ kill and restart
Instruction to Instruction Dependence

• What do we need to calculate next PC?
  – For Jumps
    • Opcode, offset, and PC
  – For Jump Register
    • Opcode and register value
  – For Conditional Branches
    • Opcode, offset, PC, and register (for condition)
  – For all others
    • Opcode and PC

• In what stage do we know these?
  – PC → Fetch
  – Opcode, offset → Decode (or Fetch?)
  – Register value → Decode
  – Branch condition ((rs)==0) → Execute (or Decode?)
**NextPC Calculation Bubbles**

\[ (I_1) \ r_1 \leftarrow (r_0) + 10 \quad \text{IF}_1 \quad \text{ID}_1 \quad \text{EX}_1 \quad \text{MA}_1 \quad \text{WB}_1 \]
\[ (I_2) \ r_3 \leftarrow (r_2) + 17 \quad \text{IF}_2 \quad \text{ID}_2 \quad \text{EX}_2 \quad \text{MA}_2 \quad \text{WB}_2 \]
\[ (I_3) \quad \text{IF}_3 \quad \text{ID}_3 \quad \text{EX}_3 \quad \text{MA}_3 \quad \text{WB}_3 \]
\[ (I_4) \quad \text{IF}_4 \quad \text{ID}_4 \quad \text{EX}_4 \quad \text{MA}_4 \quad \text{WB}_4 \]

**Resource Usage**

\[ \text{IF} \quad t_0 \quad \text{t}1 \quad \text{t}2 \quad \text{t}3 \quad \text{t}4 \quad \text{t}5 \quad \text{t}6 \quad \text{t}7 \quad \ldots \]
\[ \text{ID} \quad \text{ID}_1 \quad \text{ID}_2 \quad \text{ID}_3 \quad \text{ID}_4 \]
\[ \text{EX} \quad \text{EX}_1 \quad \text{EX}_2 \quad \text{EX}_3 \quad \text{EX}_4 \]
\[ \text{MA} \quad \text{MA}_1 \quad \text{MA}_2 \quad \text{MA}_3 \quad \text{MA}_4 \]
\[ \text{WB} \quad \text{WB}_1 \quad \text{WB}_2 \quad \text{WB}_3 \quad \text{WB}_4 \]

**What’s a good guess for next PC?**

nop \Rightarrow \text{pipeline bubble}
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?

IRSrc\textsubscript{D} = Case opcode\textsubscript{D}
  
  J, JAL \implies \text{nop}
  
  \ldots \implies \text{IM}

\begin{align*}
I_1 &\quad 096 \quad \text{ADD} \\
I_2 &\quad 100 \quad \text{J} \quad 200 \\
I_3 &\quad 104 \quad \text{ADD} \\
I_4 &\quad 304 \quad \text{ADD}
\end{align*}
Jump Pipeline Diagrams

(time) t0 t1 t2 t3 t4 t5 t6 t7 . . . .

(I_1) 096: ADD
(I_2) 100: J 200
(I_3) 104: ADD
(I_4) 304: ADD

Resource Usage

IF I_1 I_2 I_3 I_4 I_5 I_6 I_7 . . . .
ID I_1 I_2 I_3 I_4 I_5 I_6 I_7 . . . .
EX I_1 I_2 I_3 I_4 I_5 I_6 I_7 . . . .
MA I_1 I_2 I_3 I_4 I_5 I_6 I_7 . . . .
WB I_1 I_2 I_3 I_4 I_5 I_6 I_7 . . . .

nop ⇒ pipeline bubble
Pipelining Conditional Branches

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

March 11, 2021
MIT 6.823 Spring 2021
New Stall Signal

Don’t stall if the branch is taken. Why?

stall = ( ((rs_D==ws_E)·we_E + (rs_D==ws_M)·we_M + (rs_D==ws_W)·we_W)·re1_D + ((rt_D==ws_E)·we_E + (rt_D==ws_M)·we_M + (rt_D==ws_W)·we_W)·re2_D ) · !((opcode_E==BEQZ)·z + (opcode_E==BNEZ)·!z)
Control Equations for PC and IR Muxes

IRSrc\(_D\) = Case opcode\(_E\)
BEQZ·z, BNEZ·!z \(\Rightarrow\) nop
...
\(\Rightarrow\)
Case opcode\(_D\)
J, JAL, JR, JALR \(\Rightarrow\) nop
...
\(\Rightarrow\) IM

IRSrc\(_E\) = Case opcode\(_E\)
BEQZ·z, BNEZ·!z \(\Rightarrow\) nop
...
\(\Rightarrow\) stall·nop + !stall·IR\(_D\)

PCSrc = Case opcode\(_E\)
BEQZ·z, BNEZ·!z \(\Rightarrow\) br
...
\(\Rightarrow\)
Case opcode\(_D\)
J, JAL \(\Rightarrow\) jabs
JR, JALR \(\Rightarrow\) rind
...
\(\Rightarrow\) pc+4

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

\(pc+4\) is a speculative guess

nop \(\Rightarrow\) Kill
br/jabs/rind \(\Rightarrow\) Restart
pc+4 \(\Rightarrow\) Speculate
Branch Pipeline Diagrams
(resolved in execute stage)

- **IF**
- **ID**
- **EX**
- **MA**
- **WB**

<table>
<thead>
<tr>
<th>time</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(I_1)</strong> 096: <strong>ADD</strong></td>
<td>IF_1</td>
<td>ID_1</td>
<td>EX_1</td>
<td>MA_1</td>
<td>WB_1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>(I_2)</strong> 100: <strong>BEQZ 200</strong></td>
<td>IF_2</td>
<td>ID_2</td>
<td>EX_2</td>
<td>MA_2</td>
<td>WB_2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>(I_3)</strong> 104: <strong>ADD</strong></td>
<td>IF_3</td>
<td>ID_3</td>
<td>EX_3</td>
<td>MA_3</td>
<td>WB_3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>(I_4)</strong> 108:</td>
<td>IF_4</td>
<td>ID_4</td>
<td>EX_4</td>
<td>MA_4</td>
<td>WB_4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>(I_5)</strong> 304: <strong>ADD</strong></td>
<td>IF_5</td>
<td>ID_5</td>
<td>EX_5</td>
<td>MA_5</td>
<td>WB_5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Resource Usage**

- **IF**
- **ID**
- **EX**
- **MA**
- **WB**

<table>
<thead>
<tr>
<th>time</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IF</strong></td>
<td>I_1</td>
<td>I_2</td>
<td>I_3</td>
<td>I_4</td>
<td>I_5</td>
<td>I_5</td>
<td>I_5</td>
<td>I_5</td>
</tr>
<tr>
<td><strong>ID</strong></td>
<td>I_1</td>
<td>I_2</td>
<td>I_3</td>
<td>nop</td>
<td>I_5</td>
<td>I_5</td>
<td>I_5</td>
<td>I_5</td>
</tr>
<tr>
<td><strong>EX</strong></td>
<td>I_1</td>
<td>I_2</td>
<td>nop</td>
<td>nop</td>
<td>I_5</td>
<td>I_5</td>
<td>I_5</td>
<td>I_5</td>
</tr>
<tr>
<td><strong>MA</strong></td>
<td>I_1</td>
<td>I_2</td>
<td>nop</td>
<td>nop</td>
<td>I_5</td>
<td>I_5</td>
<td>I_5</td>
<td>I_5</td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td>I_1</td>
<td>I_2</td>
<td>nop</td>
<td>nop</td>
<td>I_5</td>
<td>I_5</td>
<td>I_5</td>
<td>I_5</td>
</tr>
</tbody>
</table>

**nop** ⇒ 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>$I_1$</td>
<td>096</td>
<td>ADD</td>
</tr>
<tr>
<td>$I_2$</td>
<td>100</td>
<td>BEQZ r1 200</td>
</tr>
<tr>
<td>$I_3$</td>
<td>104</td>
<td>ADD</td>
</tr>
<tr>
<td>$I_4$</td>
<td>304</td>
<td>ADD</td>
</tr>
</tbody>
</table>

  *Delay slot instruction executed regardless of branch outcome*

- 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
Handling Control Hazards due to Exceptions

- 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