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* → **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 (1)

**Strategy 1:**

Wait for the result to be available by freezing earlier pipeline stages → stall (interlocks)
Resolving Data Hazards by Stalling

**Stall Condition**

How do we know when to stall?

... $r1 \leftarrow r0 + 10$

... $r4 \leftarrow r1 + 17$

...
Compare the source registers of the instruction in the decode stage with the destination register of the uncommitted instructions.
Stall Control Logic
ignoring jumps & branches

Should we always stall if the rs field matches some rd?
not every instruction writes a register \(\Rightarrow\) we
not every instruction reads a register \(\Rightarrow\) 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>immediate16</th>
</tr>
</thead>
</table>

### J-type:

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

### Instructions:

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

source(s) destination:

- rs, rt  \rightarrow rd
- rs  \rightarrow rt
- rs, rt  \rightarrow rs, rt
- rs  \rightarrow rs
- rs  \rightarrow rs
Deriving the Stall Signal

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

\[
w_{\text{dest}} = \begin{cases} \text{Case opcode} \\
\text{ALU} & \Rightarrow \text{rd} \\
\text{ALUi, LW} & \Rightarrow \text{rt} \\
\text{JAL, JALR} & \Rightarrow R31
\end{cases}
\]

\[
w_{\text{exec}} = \begin{cases} \text{Case opcode} \\
\text{ALU, ALUi, LW} & \Rightarrow (w_{\text{dest}} \neq 0) \\
\text{JAL, JALR} & \Rightarrow \text{on} \\
\ldots & \Rightarrow \text{off}
\end{cases}
\]

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

\[
\begin{align*}
\text{re1} &= \begin{cases} \text{Case opcode} \\
\text{ALU, ALUi, LW} & \Rightarrow \text{on} \\
\text{JR, JALR} & \Rightarrow \text{off}
\end{cases} \\
\text{re2} &= \begin{cases} \text{Case opcode} \\
\text{ALU, SW} & \Rightarrow \text{on} \\
\ldots & \Rightarrow \text{off}
\end{cases}
\end{align*}
\]

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

\[
\text{stall} = \left( (r_{D} = \text{ws}_{E}) \cdot \text{we}_{E} + (r_{D} = \text{ws}_{M}) \cdot \text{we}_{M} + (r_{D} = \text{ws}_{W}) \cdot \text{we}_{W} \right) \cdot \text{re1}_{D} + \left( (r_{D} = \text{ws}_{E}) \cdot \text{we}_{E} + (r_{D} = \text{ws}_{M}) \cdot \text{we}_{M} + (r_{D} = \text{ws}_{W}) \cdot \text{we}_{W} \right) \cdot \text{re2}_{D}
\]

This is not the full story!
Hazards due to Loads & Stores

Stall Condition

What if (r1)+7 = (r3)+5 ?

... M[(r1)+7] ← (r2) 
   r4 ← M[(r3)+5] ...

Is there any possible data hazard in this instruction sequence?
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.
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 \( \Rightarrow \text{CPI} > 1 \)

### When is data actually available?  

At Execute

<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>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage</td>
<td>IF1</td>
<td>ID1</td>
<td>EX1</td>
<td>MA1</td>
<td>WB1</td>
<td>ID2</td>
<td>ID2</td>
<td>ID2</td>
<td>ID2</td>
</tr>
<tr>
<td></td>
<td>IF2</td>
<td>ID2</td>
<td>EX2</td>
<td>MA2</td>
<td>WB2</td>
<td>ID3</td>
<td>ID3</td>
<td>ID3</td>
<td>ID3</td>
</tr>
<tr>
<td></td>
<td>IF3</td>
<td>ID3</td>
<td>EX3</td>
<td>MA3</td>
<td>WB3</td>
<td>IF4</td>
<td>ID4</td>
<td>EX4</td>
<td>MA4</td>
</tr>
<tr>
<td></td>
<td>IF5</td>
<td>ID5</td>
<td>EX5</td>
<td>MA5</td>
<td>WB5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 ← r0 + 10  yes
(I_2)  r4 ← r1 + 17

r1 ← M[r0 + 10]

r4 ← r1 + 17  no

JAL  500  no
r4 ← r31 + 17
The Bypass Signal
Deriving it from the Stall Signal

\[
\text{stall} = (\text{rs}_D = \text{ws}_E) \cdot \text{we}_E + (\text{rs}_D = \text{ws}_M) \cdot \text{we}_M + (\text{rs}_D = \text{ws}_W) \cdot \text{we}_W \cdot \text{re}_1_D \\
+ ((\text{rt}_D = \text{ws}_E) \cdot \text{we}_E + (\text{rt}_D = \text{ws}_M) \cdot \text{we}_M + (\text{rt}_D = \text{ws}_W) \cdot \text{we}_W) \cdot \text{re}_2_D
\]

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

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

\[
\text{ASrc} = (\text{rs}_D = \text{ws}_E) \cdot \text{we}_E \cdot \text{re}_1_D
\]

Is this correct?

No, because only ALU and ALUi instructions can benefit from this bypass

How might we address this?

Split \text{we}_E into two components: we-bypass, we-stall
Bypass and Stall Signals

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

$$we\text{-bypass}_E = \text{Case opcode}_E$$

- ALU, ALUi $\Rightarrow (ws \neq 0)$
- ... $\Rightarrow$ off

$$we\text{-stall}_E = \text{Case opcode}_E$$

- LW $\Rightarrow (ws \neq 0)$
- JAL, JALR $\Rightarrow$ on
- ... $\Rightarrow$ off

$$ASrc = (rs_D == ws_E) \cdot we\text{-bypass}_E \cdot re_{1D}$$

$$\text{stall} = ((rs_D == ws_E) \cdot we\text{-stall}_E +$$

$$\quad (rs_D == ws_M) \cdot we_M + (rs_D == ws_W) \cdot we_W) \cdot re_{1D}$$

$$+ ((rt_D == ws_E) \cdot we_E + (rt_D == ws_M) \cdot we_M + (rt_D == ws_W) \cdot we_W) \cdot re_{2D}$$
Fully Bypassed Datapath

Is there still a need for the stall signal?

\[
\text{stall} = (rs_D = ws_E) \cdot (\text{opcode}_E = \text{LW}_E) \cdot (ws_E \neq 0) \cdot \text{re1}_D \\
+ (rt_D = ws_E) \cdot (\text{opcode}_E = \text{LW}_E) \cdot (ws_E \neq 0) \cdot \text{re2}_D
\]
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

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

IRSrcD = Case opcodeD
J, JAL ⇒ nop
... ⇒ IM

I1  096  ADD
I2  100  J  200
I3  104  ADD  kill
I4  304  ADD
Jump Pipeline Diagrams

\[
\begin{align*}
\text{time} \\
t_0 & \quad t_1 & \quad t_2 & \quad t_3 & \quad t_4 & \quad t_5 & \quad t_6 & \quad t_7 & \quad \ldots \ldots \\
(I_1) & 096: \text{ADD} \\
(I_2) & 100: J 200 \\
(I_3) & 104: \text{ADD} \\
(I_4) & 304: \text{ADD}
\end{align*}
\]

\[
\begin{align*}
\text{Resource Usage} \\
\text{IF} & \quad I_1 \\
\text{ID} & \quad I_1 \\
\text{EX} & \quad I_1 \\
\text{MA} & \quad I_1 \\
\text{WB} & \quad I_1 \\
& \quad I_2 \\
& \quad I_2 \\
& \quad I_2 \\
& \quad I_2 \\
& \quad \ldots \ldots 
\end{align*}
\]

\[
\begin{align*}
\text{nop} \Rightarrow \text{pipeline bubble}
\end{align*}
\]
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
New Stall Signal

\[
\text{stall} = \left( \left( (\text{rs}_D = \text{ws}_E) \cdot \text{we}_E + (\text{rs}_D = \text{ws}_M) \cdot \text{we}_M + (\text{rs}_D = \text{ws}_W) \cdot \text{we}_W \right) \cdot \text{re}_1_D \\
+ \left( (\text{rt}_D = \text{ws}_E) \cdot \text{we}_E + (\text{rt}_D = \text{ws}_M) \cdot \text{we}_M + (\text{rt}_D = \text{ws}_W) \cdot \text{we}_W \right) \cdot \text{re}_2_D \\
\right) \cdot \left( ((\text{opcode}_E = \text{BEQZ}) \cdot \text{z} + (\text{opcode}_E = \text{BNEZ}) \cdot !\text{z}) \right)
\]

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

Instruction at the decode stage is invalid
Control Equations for PC and IR Muxes

\[
\text{IRSrc}_D = \text{Case opcode}_E
\]

\[
\begin{align*}
\text{BEQZ} \cdot z, \text{BNEZ} \cdot !z & \Rightarrow \text{nop} \\
\ldots & \Rightarrow \text{Case opcode}_D \\
\text{J, JAL, JR, JALR} & \Rightarrow \text{nop} \\
\ldots & \Rightarrow \text{IM}
\end{align*}
\]

\[
\text{IRSrc}_E = \text{Case opcode}_E
\]

\[
\begin{align*}
\text{BEQZ} \cdot z, \text{BNEZ} \cdot !z & \Rightarrow \text{nop} \\
\ldots & \Rightarrow \text{stall} \cdot \text{nop} + !\text{stall} \cdot \text{IR}_D
\end{align*}
\]

\[
\text{PCSrc} = \text{Case opcode}_E
\]

\[
\begin{align*}
\text{BEQZ} \cdot z, \text{BNEZ} \cdot !z & \Rightarrow \text{br} \\
\ldots & \Rightarrow \text{Case opcode}_D \\
\text{J, JAL} & \Rightarrow \text{jabs} \\
\text{JR, JALR} & \Rightarrow \text{rind} \\
\ldots & \Rightarrow \text{pc}+4 \\
\end{align*}
\]

\[
\begin{align*}
\text{nop} & \Rightarrow \text{Kill} \\
\text{br/jabs/rind} & \Rightarrow \text{Restart} \\
\text{pc}+4 & \Rightarrow \text{Speculate}
\end{align*}
\]

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)

```
| time  | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | ...
|-------|----|----|----|----|----|----|----|----|-----
| IF    | I_1| I_2| I_3| I_4| I_5| I_6| I_7| I_8|     
| ID    | I_1| I_2| I_3| I_4| I_5| I_6| I_7| I_8|     
| EX    | I_1| I_2| I_3| I_4| I_5| I_6| I_7| I_8|     
| MA    | I_1| I_2| I_3| I_4| I_5| I_6| I_7| I_8|     
| WB    | I_1| I_2| I_3| I_4| I_5| I_6| I_7| I_8|     
```

Resource Usage

```
| time  | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | ...
|-------|----|----|----|----|----|----|----|----|-----
| IF    | I_1| I_2| I_3| I_4| I_5| I_6| I_7| I_8|     
| ID    | I_1| I_2| I_3| I_4| I_5| I_6| I_7| I_8|     
| EX    | I_1| I_2| I_3| I_4| I_5| I_6| I_7| I_8|     
| MA    | I_1| I_2| I_3| I_4| I_5| I_6| I_7| I_8|     
| WB    | I_1| I_2| I_3| I_4| I_5| I_6| I_7| I_8|     
```

(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>

- Other techniques include branch prediction, which can dramatically reduce the branch penalty... *to come later*
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 may cause bubbles
  – Kill following instruction(s) 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