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**

...  
\[ r1 \leftarrow r0 + 10 \]  
\[ r4 \leftarrow r1 + 17 \]  
...

How do we know when to stall?
Stall Control Logic

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>

#### source(s) destination

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

\[ C_{dest} \]
ws = Case opcode
- ALU \( \Rightarrow \) rd
- ALUi, LW \( \Rightarrow \) rt
- JAL, JALR \( \Rightarrow \) R31

we = Case opcode
- ALU, ALUi, LW \( \Rightarrow (ws \neq 0) \)
- JAL, JALR \( \Rightarrow \) on
- ... \( \Rightarrow \) off

\[ C_{re} \]
re1 = Case opcode
- ALU, ALUi, LW \( \Rightarrow \) on
- JR, JALR \( \Rightarrow \) off
- J, JAL \( \Rightarrow \) off

re2 = Case opcode
- ALU, SW \( \Rightarrow \) on
- ... \( \Rightarrow \) off

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

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

Stall Condition

Is there any possible data hazard in this instruction sequence?

What if \((r1)+7 = (r3)+5\)?
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 → bypass
Bypassing

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

When is data actually available? **At Execute**

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

\[ r4 \leftarrow r1 + 17 \]  
no

\[ JAL \ 500 \]  
no

\[ r4 \leftarrow r31 + 17 \]  
no

http://www.csg.csail.mit.edu/6.823

Sanchez & Emer
The Bypass Signal
Deriving it from the Stall Signal

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

\[
\text{ws} = \text{Case opcode}
\]

ALU \implies \text{rd}

ALUi, LW \implies \text{rt}

JAL, JALR \implies \text{R31}

\[
\text{we} = \text{Case opcode}
\]

ALU, ALUi, LW \implies (\text{ws} \neq 0)

JAL, JALR \implies \text{on}

... \implies \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 $w_{E}$ into two components: we-bypass, we-stall

\[
\text{we-bypass}_{E} = \text{Case opcode}_{E}
\]
\[
\text{ALU, ALUi} \Rightarrow (ws \neq 0)
\]
\[
\ldots \Rightarrow \text{off}
\]

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

\[
\text{ASrc} = (rs_{D} = ws_{E}) \cdot \text{we-bypass}_{E} \cdot \text{re}_{1_{D}}
\]

\[
\text{stall} = ((rs_{D} = ws_{E}) \cdot \text{we-stall}_{E} +
\]
\[
(rs_{D} = ws_{M}) \cdot \text{we}_{M} + (rs_{D} = ws_{W}) \cdot \text{we}_{W} \cdot \text{re}_{1_{D}}
\]
\[
+((rt_{D} = ws_{E}) \cdot \text{we}_{E} + (rt_{D} = ws_{M}) \cdot \text{we}_{M} + (rt_{D} = ws_{W}) \cdot \text{we}_{W}) \cdot \text{re}_{2_{D}}
\]
Fully Bypassed Datapath

Is there still a need for the stall signal?

\[
\text{stall} = (r_{D} = w_{E}) \cdot (o_{D} = \text{LW}) \cdot (w_{E} \neq 0) \cdot r_{1D} + (r_{D} = w_{E}) \cdot (o_{D} = \text{LW}) \cdot (w_{E} \neq 0) \cdot r_{2D}
\]
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
    • 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

`time`

(t0) r1 ← (r0) + 10  (I1)  r1 ← (r0) + 10  (I1)  (I1)  (I1)

(t1) IF1  ID1  EX1  MA1  WB1  IF2  ID2  EX2  MA2  WB2  IF3  ID3  EX3  MA3  WB3  IF4  ID4  EX4  MA4  WB4

(t2)

(t3)

(t4)

(t5)

(t6)

(t7)

. . . .

(I2) r3 ← (r2) + 17  (I2)  (I2)  (I2)  (I2)

(I3)  (I3)  (I3)  (I3)  (I3)

(I4)  (I4)  (I4)  (I4)  (I4)

Resource Usage

IF  I1  nop  I2  nop  I3  nop  I4  nop  I4

ID  I1  nop  I2  nop  I3  nop  I4  nop  I4

EX  I1  nop  I2  nop  I3  nop  I4  nop  I4

MA  I1  nop  I2  nop  I3  nop  I4  nop  I4

WB  I1  nop  I2  nop  I3  nop  I4  nop  I4

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

nop  ⇒  pipeline bubble
Speculate NextPC is PC+4

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

I1  096  ADD
I2  100  J   200
I3  104  ADD
I4  304  ADD

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
I4 304 ADD

kill
Jump Pipeline Diagrams

\[ \text{time} \]
\[ \begin{array}{cccccccc}
  t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 \\
  \text{IF}_1 & \text{ID}_1 & \text{EX}_1 & \text{MA}_1 & \text{WB}_1 & \text{IF}_2 & \text{ID}_2 & \text{EX}_2 & \text{MA}_2 & \text{WB}_2 & \text{IF}_3 & \text{nop} & \text{nop} & \text{nop} & \text{nop} & \text{IF}_4 & \text{ID}_4 & \text{EX}_4 & \text{MA}_4 & \text{WB}_4
\end{array} \]

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

\( \text{nop} \Rightarrow \text{pipeline bubble} \)
Pipelining Conditional Branches

Branch condition is not known until the execute stage.

what action should be taken in the decode stage?

I_1 096 ADD
I_2 100 BEQZ r1 200
I_3 104 ADD
I_4 304 ADD
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

\[
stall = ( (rs_D = ws_E) \cdot we_E + (rs_D = ws_M) \cdot we_M + (rs_D = ws_W) \cdot we_W ) \cdot re_{1D}
+ ( (rt_D = ws_E) \cdot we_E + (rt_D = ws_M) \cdot we_M + (rt_D = ws_W) \cdot we_W ) \cdot re_{2D}
\cdot !((opcode_E = BEQZ) \cdot z + (opcode_E = BNEZ) \cdot !z)
\]

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

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

**IRSrc\textsubscript{D} = Case opcode\textsubscript{E}**
- BEQZ\cdot z, BNEZ\cdot !z \implies \text{nop}
- ... \implies \text{Case opcode}\textsubscript{D}
  - J, JAL, JR, JALR \implies \text{nop}
- ... \implies \text{IM}

**IRSrc\textsubscript{E} = Case opcode\textsubscript{E}**
- BEQZ\cdot z, BNEZ\cdot !z \implies \text{nop}
- ... \implies \text{stall}\cdot \text{nop} + \text{!stall}\cdot \text{IR}\textsubscript{D}

**PCSrc = Case opcode\textsubscript{E}**
- BEQZ\cdot z, BNEZ\cdot !z \implies \text{br}
- ... \implies \text{Case opcode}\textsubscript{D}
  - J, JAL \implies \text{jabs}
  - JR, JALR \implies \text{rind}
  - ... \implies \text{pc}+4

- \text{nop} \implies \text{Kill}
- br/jabs/rind \implies \text{Restart}
- pc+4 \implies \text{Speculate}

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

*pc+4 is a speculative guess*
Branch Pipeline Diagrams
(resolved in execute stage)

\[
time \\
t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \quad t_7 \quad \ldots
\]

\begin{align*}
(I_1) \ 096: \text{ADD} & \quad \text{IF}_1 \quad \text{ID}_1 \quad \text{EX}_1 \quad \text{MA}_1 \quad \text{WB}_1 \\
(I_2) \ 100: \text{BEQZ} 200 & \quad \text{IF}_2 \quad \text{ID}_2 \quad \text{EX}_2 \quad \text{MA}_2 \quad \text{WB}_2 \\
(I_3) \ 104: \text{ADD} & \quad \text{IF}_3 \quad \text{ID}_3 \quad \text{nop} \quad \text{nop} \quad \text{nop} \\
(I_4) \ 108: & \quad \text{IF}_4 \quad \text{ID}_4 \quad \text{nop} \quad \text{nop} \quad \text{nop} \quad \text{nop} \\
(I_5) \ 304: \text{ADD} & \quad \text{IF}_5 \quad \text{ID}_5 \quad \text{EX}_5 \quad \text{MA}_5 \quad \text{WB}_5
\end{align*}

Resource Usage

\[
\begin{array}{c}
\text{IF} \\
\text{ID} \\
\text{EX} \\
\text{MA} \\
\text{WB}
\end{array}
\]

\[
\begin{align*}
time \\
t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \quad t_7 \quad \ldots
\end{align*}
\]

\[
\begin{align*}
\text{IF} & \quad I_1 \quad I_2 \quad I_3 \quad I_4 \quad I_5 \\
\text{ID} & \quad I_1 \quad I_2 \quad I_3 \quad \text{nop} \quad I_5 \\
\text{EX} & \quad I_1 \quad I_2 \quad \text{nop} \quad \text{nop} \quad I_5 \\
\text{MA} & \quad I_1 \quad I_2 \quad \text{nop} \quad \text{nop} \quad I_5 \\
\text{WB} & \quad I_1 \quad I_2 \quad \text{nop} \quad \text{nop} \quad I_5
\end{align*}
\]

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

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