Instruction Pipelining: Hazard Resolution, Timing Constraints

Daniela 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* \( \rightarrow \) *stall*

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

Strategy 3: *Speculate on the dependence*
*Two cases:*
- *Guessed correctly* \( \rightarrow \) no special action required
- *Guessed incorrectly* \( \rightarrow \) kill and restart
Resolving Data Hazards (1)

Strategy 1:

Wait for the result to be available by freezing earlier pipeline stages \( \rightarrow \text{stall} \) (interlocks)
Resolving Data Hazards by Stalling

Stall Condition

... r1 ← r0 + 10
r4 ← r1 + 17
...

How do we know when to stall?

March 1, 2017
http://www.csg.csail.mit.edu/6.823
Sanchez & Emer
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 ⇒ we
not every instruction reads a register ⇒ 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>rs, rt</th>
<th>rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>rd ← (rs) func (rt)</td>
<td>rs, rt</td>
<td>rd</td>
</tr>
<tr>
<td>ALUi</td>
<td>rt ← (rs) op imm</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>LW</td>
<td>rt ← M [(rs) + imm]</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>SW</td>
<td>M [(rs) + imm] ← (rt)</td>
<td>rs, rt</td>
<td></td>
</tr>
<tr>
<td>BZ</td>
<td>cond (rs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>true: PC ← (PC) + imm</td>
<td>rs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>false: PC ← (PC) + 4</td>
<td>rs</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>PC ← (PC) + imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAL</td>
<td>r31 ← (PC), PC ← (PC) + imm</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>JR</td>
<td>PC ← (rs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JALR</td>
<td>r31 ← (PC), PC ← (rs)</td>
<td>rs</td>
<td>31</td>
</tr>
</tbody>
</table>
## Deriving the Stall Signal

### \( C_{dest} \)

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

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

### \( C_{re} \)

\[
re1 = \text{Case opcode} \\
ALU, ALUi, LW \Rightarrow on \\
J, JAL \Rightarrow off
\]

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

### \( C_{stall} \)

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

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

Is there any possible data hazard in this instruction sequence?

M[(r1)+7] ← (r2)
r4 ← M[(r3)+5]

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?

(I₁)  r1 ← r0 + 10
(I₂)  r4 ← r1 + 17

yes

r1 ← M[r0 + 10]
r4 ← r1 + 17

no

JAL  500
r4 ← r31 + 17

no
The Bypass Signal

Deriving it from the Stall Signal

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

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

\[ \text{we} = \text{Case opcode} \]
- ALU, ALUi, LW \Rightarrow (\text{ws} \neq 0)
- JAL, JALR \Rightarrow \text{on}
- ... \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 $w_{E}$ into two components: $we$-bypass, $we$-stall

$we_{bypassE} = \begin{cases} 
\text{Case opcode}_{E} & \text{ALU, ALUi } \Rightarrow (ws \neq 0) \\
... & \Rightarrow \text{off}
\end{cases}$

$we_{stallE} = \begin{cases} 
\text{Case opcode}_{E} & \text{LW } \Rightarrow (ws \neq 0) \\
\text{JAL, JALR } & \Rightarrow \text{on} \\
... & \Rightarrow \text{off}
\end{cases}$

$ASrc = (rs_{D} = ws_{E}) \cdot we_{bypassE} \cdot re_{1D}$

$stall = ((rs_{D} = ws_{E}) \cdot we_{bypassE} +$

$(rs_{D} = ws_{M}) \cdot we_{M} + (rs_{D} = ws_{W}) \cdot we_{W}) \cdot re_{1D}$

$+ ((rt_{D} = ws_{E}) \cdot we_{E} + (rt_{D} = ws_{M}) \cdot we_{M} + (rt_{D} = ws_{W}) \cdot we_{W}) \cdot re_{2D}$
Is there still a need for the stall signal?

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

\[\begin{align*}
(I_1) & \quad r_1 \leftarrow (r_0) + 10 \\
(I_2) & \quad r_3 \leftarrow (r_2) + 17 \\
(I_3) & \\
(I_4) & \\
\end{align*}\]

Time steps:
\[\begin{align*}
\text{time} & \quad t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \quad t_7 \quad \ldots \\
\text{IF}_1 & \quad \text{ID}_1 \quad \text{EX}_1 \quad \text{MA}_1 \quad \text{WB}_1 \\
\text{IF}_2 & \quad \text{ID}_2 \quad \text{EX}_2 \quad \text{MA}_2 \quad \text{WB}_2 \\
\text{IF}_3 & \quad \text{ID}_3 \quad \text{EX}_3 \quad \text{MA}_3 \quad \text{WB}_3 \\
\text{IF}_4 & \quad \text{ID}_4 \quad \text{EX}_4 \quad \text{MA}_4 \quad \text{WB}_4 \\
\end{align*}\]

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

What’s a good guess for next PC? PC+4

nop \Rightarrow \text{pipeline bubble}
Speculate NextPC is PC+4

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

Kill

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?

\[
\text{IRSrc}_D = \begin{cases} 
\text{Case opcode}_D \\
J, JAL & \Rightarrow \text{nop} \\
\ldots & \Rightarrow \text{IM}
\end{cases}
\]
Jump Pipeline Diagrams

(time

I_{0} \quad t0 \quad t1 \quad t2 \quad t3 \quad t4 \quad t5 \quad t6 \quad t7 \quad \ldots

I_{1} \quad I_{1} \quad I_{2} \quad I_{3} \quad I_{4} \quad I_{5} \quad I_{6} \quad I_{7} \quad \ldots

IF_{1} \quad ID_{1} \quad EX_{1} \quad MA_{1} \quad WB_{1}
IF_{2} \quad ID_{2} \quad EX_{2} \quad MA_{2} \quad WB_{2}
IF_{3} \quad nop \quad nop \quad nop \quad nop
IF_{4} \quad ID_{4} \quad EX_{4} \quad MA_{4} \quad WB_{4}

Resource Usage

IF \quad I_{1} \quad I_{2} \quad I_{3} \quad I_{4} \quad I_{5} \quad I_{6} \quad I_{7} \quad \ldots
ID \quad I_{1} \quad I_{2} \quad I_{3} \quad I_{4} \quad I_{5} \quad I_{6} \quad I_{7} \quad \ldots
EX \quad I_{1} \quad I_{2} \quad I_{3} \quad I_{4} \quad I_{5} \quad I_{6} \quad I_{7} \quad \ldots
MA \quad I_{1} \quad I_{2} \quad I_{3} \quad I_{4} \quad I_{5} \quad I_{6} \quad I_{7} \quad \ldots
WB \quad I_{1} \quad I_{2} \quad I_{3} \quad I_{4} \quad I_{5} \quad I_{6} \quad I_{7} \quad \ldots

nop \Rightarrow 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

March 1, 2017  http://www.csg.csail.mit.edu/6.823  Sanchez & Emer
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?

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

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

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

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

\text{Give priority to the older instruction, i.e., execute stage instruction over decode stage instruction}

\text{pc} + 4 \text{ is a speculative guess}

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

\[ \text{time} \]
\[
\begin{array}{cccccccc}
  t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 & \ldots \\
\end{array}
\]

(I_1) 096: ADD
(I_2) 100: BEQZ 200
(I_3) 104: ADD
(I_4) 108:
(I_5) 304: ADD

Resource Usage

**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.

| I₁ | 096 | ADD             | Delay slot instruction |
| I₂ | 100 | BEQZ r1 200     | executed regardless of |
| I₃ | 104 | ADD             | branch outcome         |
| I₄ | 304 | ADD             |                         |

- 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.