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* \(\rightarrow\) **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 → stall (interlocks)*
Resolving Data Hazards by Stalling

Stall Condition

... r1 ← r0 + 10
r4 ← 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.
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>ALU</th>
<th>rd ← (rs) func (rt)</th>
<th>rs, rt</th>
<th>rd</th>
</tr>
</thead>
<tbody>
<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>rs</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

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

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

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

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

\[ re2 = \text{Case opcode} \]
- ALU, SW \( \Rightarrow \) on
- ... \( \Rightarrow \) 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!
Hazards due to Loads & Stores

**Stall Condition**

Is there any possible data hazard in this instruction sequence?

... $M[(r1)+7] \leftarrow (r2)$

$\quad r4 \leftarrow M[(r3)+5]$

...
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 → bypass
Bypassing

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

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 ← r0 + 10  yes  r1 ← M[r0 + 10]  no  JAL 500  no
(I_2)  r4 ← r1 + 17  yes  r4 ← r1 + 17  no  r4 ← r31 + 17  no

The Bypass Signal
Deriving it from the Stall Signal

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

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

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

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

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

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

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

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

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

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

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

$(rs_D = ws_M) \cdot we_M + (rs_D = ws_W) \cdot we_W) \cdot re1_D$

$+ (\text{rt}_D = ws_E) \cdot we_E + (\text{rt}_D = ws_M) \cdot we_M + (\text{rt}_D = ws_W) \cdot we_W) \cdot re2_D$
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 $\rightarrow$ 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 $\rightarrow$ Fetch
  - Opcode, offset $\rightarrow$ Decode (or Fetch?)
  - Register value $\rightarrow$ Decode
  - Branch condition ($(rs)==0) \rightarrow$ Execute (or Decode?)
NextPC Calculation Bubbles

\[
time \\
\begin{array}{cccccccc}
  t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 & \ldots \\
\hline
  I_1 & & & & & & & \\
\end{array}
\]

*(I_1)* \( r_1 \leftarrow (r_0) + 10 \) \( \text{IF}_1 \)

*(I_2)* \( r_3 \leftarrow (r_2) + 17 \) \( \text{IF}_2 \)

*(I_3)*

*(I_4)*

\[
\begin{array}{cccccccc}
  t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 & \ldots \\
\hline
  \text{IF} & I_1 & \text{nop} & I_2 & \text{nop} & I_3 & \text{nop} & I_4 \\
  \text{ID} & I_1 & \text{nop} & I_2 & \text{nop} & I_3 & \text{nop} & I_4 \\
  \text{EX} & I_1 & \text{nop} & I_2 & \text{nop} & I_3 & \text{nop} & I_4 \\
  \text{MA} & I_1 & \text{nop} & I_2 & \text{nop} & I_3 & \text{nop} & I_4 \\
  \text{WB} & I_1 & \text{nop} & I_2 & \text{nop} & I_3 & \text{nop} & I_4 \\
\end{array}
\]

*Resource Usage*

\[
\text{nop} \Rightarrow \text{pipeline bubble}
\]

What’s a good guess for next PC? \( \text{PC}+4 \)
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
Jump Pipeline Diagrams

(time
(t0  t1  t2  t3  t4  t5  t6  t7  ....
(IF1  ID1  EX1  MA1  WB1
(IF2  ID2  EX2  MA2  WB2
(IF3  nop  nop  nop  nop
(IF4  ID4  EX4  MA4  WB4

(I1) 096: ADD
(I2) 100: J 200
(I3) 104: ADD
(I4) 304: ADD

Resource Usage

IF  I1  I2  I3  I4  I5  I6  I7  ....
ID  I1  I2  I3  I4  I5  I6  I7  ....
EX  I1  I2  I3  I4  I5  I6  I7  ....
MA  I1  I2  I3  I4  I5  I6  I7  ....
WB  I1  I2  I3  I4  I5  I6  I7  ....

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
New Stall Signal

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

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

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

**IRSr** = \(Case \text{ opcode}_E\)

- \(\text{BEQZ} \cdot z, \text{BNEZ} \cdot !z\) \(\Rightarrow\) nop
- ... \(\Rightarrow\) \(Case \text{ opcode}_D\)
- \(\text{J}, \text{JAL}, \text{JR}, \text{JALR}\) \(\Rightarrow\) nop
- ... \(\Rightarrow\) IM

**IRSr** = \(Case \text{ opcode}_E\)

- \(\text{BEQZ} \cdot z, \text{BNEZ} \cdot !z\) \(\Rightarrow\) nop
- ... \(\Rightarrow\) stall \cdot nop + !stall \cdot IR_D

**PCSr** = \(Case \text{ opcode}_E\)

- \(\text{BEQZ} \cdot z, \text{BNEZ} \cdot !z\) \(\Rightarrow\) br
- ... \(\Rightarrow\) \(Case \text{ opcode}_D\)
- \(\text{J}, \text{JAL}\) \(\Rightarrow\) jabs
- \(\text{JR}, \text{JALR}\) \(\Rightarrow\) 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)

(time)

t0 t1 t2 t3 t4 t5 t6 t7 . . . .

(I_1) 096: ADD

IF_1 ID_1 EX_1 MA_1 WB_1

(I_2) 100: BEQZ 200

IF_2 ID_2 EX_2 MA_2 WB_2

(I_3) 104: ADD

ID_3 nop nop nop

(I_4) 108:

IF_4 ID_4 EX_4 MA_4 WB_4

(I_5) 304: ADD

IF_5 ID_5 EX_5 MA_5 WB_5

Resource Usage

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.

\[ I_1 \quad 096 \quad \text{ADD} \]
\[ I_2 \quad 100 \quad \text{BEQZ r1 200} \quad \text{Delay slot instruction} \]
\[ I_3 \quad 104 \quad \text{ADD} \quad \text{executed regardless of branch outcome} \]
\[ I_4 \quad 304 \quad \text{ADD} \]

- 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.