Instruction Pipelining:
Hazard Resolution, Timing Constraints

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Reminder: Pipelined MIPS Datapath

without jumps

Pipelining increases clock frequency, but instruction dependences may increase CPI

September 22, 2021
How instructions can interact with each other in a pipeline

• An instruction in the pipeline may need a resource being used by another instruction in the pipeline → **structural hazard**

• An instruction may depend on a value produced by an earlier instruction
  
  – Dependence may be for a data calculation → **data hazard**

  – Dependence may be for calculating the next PC → **control hazard (branches, interrupts)**
Data Hazards

... r1 ← r0 + 10
r4 ← r1 + 17
...

r1 is stale. Oops!
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
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)
Source & Destination Registers

\[R\text{-type:} \quad \begin{array}{cccc} \text{op} & \text{rs} & \text{rt} & \text{rd} \\ \hline \end{array} \quad \text{func}\]

\[I\text{-type:} \quad \begin{array}{cccc} \text{op} & \text{rs} & \text{rt} & \text{immediate}_{16} \end{array}\]

\[J\text{-type:} \quad \begin{array}{cccc} \text{op} & \text{immediate}_{26} \end{array}\]

\[\text{source(s)} \quad \text{destination}\]

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

**C\text{dest}**

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

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

**C\text{re}**

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

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

**C\text{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

Stall Condition

Is there any possible data hazard in this instruction sequence?

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

\[
\begin{align*}
M[(r1)+7] & \leftarrow (r2) \\
r4 & \leftarrow M[(r3)+5]
\end{align*}
\]
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.

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

(r1)+7 = (r3)+5 ⇒ data hazard
Resolving Data Hazards (2)

Strategy 2:

Route data as soon as possible after it is calculated to the earlier pipeline stage $\rightarrow$ 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?

(\(I_1\))  \(r1 \leftarrow r0 + 10\)  yes  \(r1 \leftarrow M[r0 + 10]\)  \(r4 \leftarrow r1 + 17\)  \(JAL 500\)  no  \(r4 \leftarrow r31 + 17\)  no
The Bypass Signal
Deriving it from the Stall Signal

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

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

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

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

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

Is this correct?

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

How might we address this?

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

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

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

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

\[
\text{ASrc} = (rs_D == ws_E) \cdot \text{we-bypass}_E \cdot \text{re}_1_D
\]

\[
\text{stall} = ((rs_D == ws_E) \cdot \text{we-stall}_E + \\
\quad (rs_D == ws_M) \cdot \text{we}_M + (rs_D == ws_W) \cdot \text{we}_W) \cdot \text{re}_1_D \\
\quad + ((rt_D == ws_E) \cdot \text{we}_E + (rt_D == ws_M) \cdot \text{we}_M + (rt_D == ws_W) \cdot \text{we}_W) \cdot \text{re}_2_D
\]
Is there still a need for the stall signal?

\[
\text{stall} = (\text{rs}_D = \text{ws}_E) \cdot (\text{opcode}_E = \text{LW}_E) \cdot (\text{ws}_E \neq 0) \cdot \text{re1}_D \\
+ (\text{rt}_D = \text{ws}_E) \cdot (\text{opcode}_E = \text{LW}_E) \cdot (\text{ws}_E \neq 0) \cdot \text{re2}_D
\]
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

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?

I \(_1\)  096  ADD
I \(_2\)  100  J  200
I \(_3\)  104  ADD
I \(_4\)  304  ADD

How?
Pipelining Jumps

To kill a fetched instruction -- Insert a nop in IR

Any interaction between stall and jump?

IRS\textsubscript{D} = Case opcode\textsubscript{D}
- J, JAL ⇒ nop
- ... ⇒ IM

I\textsubscript{1} 096 ADD
I\textsubscript{2} 100 J 200
I\textsubscript{3} 104 ADD
I\textsubscript{4} 304 ADD

kill

September 22, 2021
Jump Pipeline Diagrams

time

\( t_0 \ t_1 \ t_2 \ t_3 \ t_4 \ t_5 \ t_6 \ t_7 \ \ldots \)

\( \text{(I}_1\text{)} \ 096: \text{ADD} \)
\( \text{(I}_2\text{)} \ 100: \text{J 200} \)
\( \text{(I}_3\text{)} \ 104: \text{ADD} \)
\( \text{(I}_4\text{)} \ 304: \text{ADD} \)

\( \text{IF}_1 \ \text{ID}_1 \ \text{EX}_1 \ \text{MA}_1 \ \text{WB}_1 \)
\( \text{IF}_2 \ \text{ID}_2 \ \text{EX}_2 \ \text{MA}_2 \ \text{WB}_2 \)
\( \text{IF}_3 \ \text{nop} \ \text{nop} \ \text{nop} \ \text{nop} \)
\( \text{IF}_4 \ \text{ID}_4 \ \text{EX}_4 \ \text{MA}_4 \ \text{WB}_4 \)

\( \text{IF} \ \text{ID} \ \text{EX} \ \text{MA} \ \text{WB} \)

\( \text{Resource Usage} \)

\( \text{nop} \ \Rightarrow \ \text{pipeline bubble} \)
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

September 22, 2021

MIT 6.823 Fall 2021
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

IRSrc\(_D\) = \text{Case opcode}_E
\begin{align*}
\text{BEQZ} \cdot z, \text{BNEZ} \cdot !z & \Rightarrow \text{nop} \\
... & \Rightarrow \\
\text{Case opcode}_D
\text{J, JAL, JR, JALR} & \Rightarrow \text{nop} \\
... & \Rightarrow \text{IM}
\end{align*}

IRSrc\(_E\) = \text{Case opcode}_E
\begin{align*}
\text{BEQZ} \cdot z, \text{BNEZ} \cdot !z & \Rightarrow \text{nop} \\
... & \Rightarrow \text{stall} \cdot \text{nop} + \text{!stall} \cdot IR_D
\end{align*}

PC Src = \text{Case opcode}_E
\begin{align*}
\text{BEQZ} \cdot z, \text{BNEZ} \cdot !z & \Rightarrow \text{br} \\
... & \Rightarrow \\
\text{Case opcode}_D
\text{J, JAL} & \Rightarrow \text{jabs} \\
\text{JR, JALR} & \Rightarrow \text{rind} \\
... & \Rightarrow \text{pc}+4
\end{align*}

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

pc+4 is a speculative guess

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

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I_1) 096: ADD</td>
<td>IF_1</td>
<td>ID_1</td>
<td>EX_1</td>
<td>MA_1</td>
<td>WB_1</td>
</tr>
<tr>
<td>(I_2) 100: BEQZ 200</td>
<td>IF_2</td>
<td>ID_2</td>
<td>EX_2</td>
<td>MA_2</td>
<td>WB_2</td>
</tr>
<tr>
<td>(I_3) 104: ADD</td>
<td>IF_3</td>
<td>ID_3</td>
<td>EX_3</td>
<td>MA_3</td>
<td>WB_3</td>
</tr>
<tr>
<td>(I_4) 108:</td>
<td>IF_4</td>
<td>ID_4</td>
<td>EX_4</td>
<td>MA_4</td>
<td>WB_4</td>
</tr>
<tr>
<td>(I_5) 304: ADD</td>
<td>IF_5</td>
<td>ID_5</td>
<td>EX_5</td>
<td>MA_5</td>
<td>WB_5</td>
</tr>
</tbody>
</table>

Resource Usage

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

**time**

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

\[ \text{(I_1)} \ 096: \text{ADD} \]
\[ \text{(I_2)} \ 100: \text{BEQZ} 200 \]
\[ \text{(I_3)} \ 104: \text{ADD} \]
\[ \text{(I_4)} \ 108: \]
\[ \text{(I_5)} \ 304: \text{ADD} \]

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

September 22, 2021
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 | 096  | ADD |
| I_2 | 100  | BEQZ r1 200 |
| I_3 | 104  | ADD |
| I_4 | 304  | ADD |

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