Instruction Pipelining:
Hazard Resolution, Timing Constraints

Joel Emer
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Pipeline Diagram – Ideal Pipelining

\[
\begin{align*}
time & \quad t0 \quad t1 \quad t2 \quad t3 \quad t4 \quad t5 \quad t6 \quad t7 \quad \ldots \\
(I_1) & \quad r1 \leftarrow r0 + 10 \\
(I_2) & \quad r3 \leftarrow r2 + 12 \\
(I_3) & \quad r5 \leftarrow r4 + 14 \\
(I_4) & \quad r7 \leftarrow r6 + 16 \\
(I_5) & \quad r9 \leftarrow r8 + 18 \\
\end{align*}
\]

Over long term, i.e., in steady state, what is...

\[
T \quad L \quad N
\]

CPI? 1  IPC? 1  Inst exec time? 5

Num inst in flight? \[ T = \frac{N}{L} \rightarrow 5 \]
Reminder: Pipelined MIPS Datapath

without jumps

Pipelining increases clock frequency, but instruction dependences may increase CPI
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

... $r_1 \leftarrow r_0 + 10$

$\ldots$

$\ldots$

$r_3 \leftarrow r_1 + 12$

... $r_1$ is stale.Oops!

Cycle 0

MIT 6.5900 (ne 6.823) Fall 2022

September 22, 2021
Pipeline Diagram – Hazard

\[ \text{time} \]
\[(I_1) \ r1 \leftarrow r0 + 10\]
\[(I_2) \ r3 \leftarrow r1 + 12\]

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

\( \text{IF}_1 \quad \text{ID}_1 \quad \text{EX}_1 \quad \text{MA}_1 \quad \text{WB}_1 \)

\( \text{IF}_2 \quad \text{ID}_2 \quad \text{EX}_2 \quad \text{MA}_2 \quad \text{WB}_2 \)

\( r1 \text{ is stale. Oops!} \)
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
Resolving Data Hazards

Strategy 1: *Wait for the result to be available by freezing earlier pipeline stages → stall*
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-type:

\[
\text{op} \quad \text{rs} \quad \text{rt} \quad \text{rd} \quad \text{func}
\]

### I-type:

\[
\text{op} \quad \text{rs} \quad \text{rt} \quad \text{immediate16}
\]

### J-type:

\[
\text{op} \quad \text{immediate26}
\]

- **ALU**: \( \text{rd} \leftarrow (\text{rs}) \text{ func (rt)} \)
  - Destination: \( \text{rs, rt, rd} \)

- **ALUi**: \( \text{rt} \leftarrow (\text{rs}) \text{ op imm} \)
  - Destination: \( \text{rs, rt} \)

- **LW**: \( \text{rt} \leftarrow \text{M [(rs) + imm]} \)
  - Destination: \( \text{rs, rt} \)

- **SW**: \( \text{M [(rs) + imm]} \leftarrow (\text{rt}) \)
  - Destination: \( \text{rs, rt} \)

- **BZ**: \( \text{cond (rs)} \)
  - \text{true}:
    - \( \text{PC} \leftarrow (\text{PC}) + \text{imm} \)
    - Destination: \( \text{rs} \)
  - \text{false}:
    - \( \text{PC} \leftarrow (\text{PC}) + 4 \)
    - Destination: \( \text{rs} \)

- **J**: \( \text{PC} \leftarrow (\text{PC}) + \text{imm} \)

- **JAL**: \( \text{r31} \leftarrow (\text{PC}), \text{PC} \leftarrow (\text{PC}) + \text{imm} \)

- **JR**: \( \text{PC} \leftarrow (\text{rs}) \)
  - Destination: \( \text{rs} \)

- **JALR**: \( \text{r31} \leftarrow (\text{PC}), \text{PC} \leftarrow (\text{rs}) \)
  - Destination: \( \text{rs} \)
# Deriving the Stall Signal

\[ C_{dest} \]

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

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

\[ C_{re} \]

\[ \begin{align*}
re1 &= \text{Case opcode} \\
\text{ALU, ALUi, LW} &\implies \text{on} \\
\text{J, JAL} &\implies \text{off}
\end{align*} \]

\[ re2 = \text{Case opcode} \\
\text{ALU, SW} &\implies \text{on} \\
\ldots &\implies \text{off}
\]

\[ C_{stall} \]

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

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

Stall Condition

M[(r1)+7] ← (r2)
r4 ← M[(r3)+5]

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 $\rightarrow$ *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 ← r0 + 10
(I_2) r2 ← r1 + 12

<table>
<thead>
<tr>
<th></th>
<th>yes</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I_1) r1 ← r0 + 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(I_2) r2 ← r1 + 12</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

JAL 500 r2 ← r31 + 12

September 22, 2021

MIT 6.5900 (ne 6.823) Fall 2022

L05-16
The Bypass Signal

Deriving it from the Stall Signal

\[ \text{stall} = \left( (rs_D = ws_E) \cdot we_E + (rs_D = ws_M) \cdot we_M + (rs_D = ws_W) \cdot we_W \right) \cdot re_{1D} \]
\[ + \left( (rt_D = ws_E) \cdot we_E + (rt_D = ws_M) \cdot we_M + (rt_D = ws_W) \cdot we_W \right) \cdot re_{2D} \]

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

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

\[ \text{ASrc} = (rs_D = ws_E) \cdot we_E \cdot re_{1D} \]

Is this correct?

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

How might we address this?

Split \( we_E \) into two components: we-bypass, we-stall
Bypass and Stall Signals

Split $w_{E}$ into two components: $we_{-bypass}$, $we_{-stall}$

$we_{-bypass_{E}} = \text{Case opcode}_{E}$
- $\text{ALU, ALUi} \Rightarrow (ws \neq 0)$
- $\ldots \Rightarrow \text{off}$

$we_{-stall_{E}} = \text{Case opcode}_{E}$
- LW $\Rightarrow (ws \neq 0)$
- JAL, JALR $\Rightarrow \text{on}$
- $\ldots \Rightarrow \text{off}$

$ASrc = (rs_{D} == ws_{E}) \cdot we_{-bypass_{E}} \cdot re1_{D}$

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

$(rs_{D} == ws_{M}) \cdot we_{M} + (rs_{D} == ws_{W}) \cdot we_{W} \cdot re1_{D}$

$+((rt_{D} == ws_{E}) \cdot we_{E} + (rt_{D} == ws_{M}) \cdot we_{M} + (rt_{D} == ws_{W}) \cdot we_{W} \cdot re2_{D}$
Is there still a need for the stall signal?

\[
\text{stall} = (rs_D == ws_E) \cdot (\text{opcode}_E == \text{LW}_E) \cdot (ws_E \neq 0) \cdot re1_D \\
+ (rt_D == ws_E) \cdot (\text{opcode}_E == \text{LW}_E) \cdot (ws_E \neq 0) \cdot re2_D
\]
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 → Fetch
  - Opcode, offset → Decode (or Fetch?)
  - Register value → Decode
  - Branch condition ((rs)==0) → Execute (or Decode?)
**NextPC Calculation Bubbles**

<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></td>
<td>(I_1) r1 ← (r0) + 10</td>
<td>IF_1</td>
<td>ID_1</td>
<td>EX_1</td>
<td>MA_1</td>
<td>WB_1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(I_2) r3 ← (r2) + 17</td>
<td>IF_2</td>
<td>ID_2</td>
<td>EX_2</td>
<td>MA_2</td>
<td>WB_2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(I_3)</td>
<td>IF_3</td>
<td>ID_3</td>
<td>EX_3</td>
<td>MA_3</td>
<td>WB_3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(I_4)</td>
<td>IF_4</td>
<td>ID_4</td>
<td>EX_4</td>
<td>MA_4</td>
<td>WB_4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Resource Usage**

- **IF**: I_1 \( \rightleftharpoons \text{nop} \rightarrow I_1 \rightleftharpoons I_2 \rightleftharpoons I_3 \rightleftharpoons I_4 \rightleftharpoons \text{pipeline bubble}
- **ID**: I_1 \( \rightleftharpoons \text{nop} \rightarrow I_1 \rightleftharpoons I_2 \rightleftharpoons I_3 \rightleftharfpoons I_4 \rightleftharpoons \text{pipeline bubble}
- **EX**: I_1 \( \rightleftharpoons \text{nop} \rightarrow I_1 \rightleftharpoons I_2 \rightleftharpoons I_3 \rightleftharpoons I_4 \rightleftharpoons \text{pipeline bubble}
- **MA**: I_1 \( \rightleftharpoons \text{nop} \rightarrow I_1 \rightleftharpoons I_2 \rightleftharpoons I_3 \rightleftharpoons I_4 \rightleftharpoons \text{pipeline bubble}
- **WB**: I_1 \( \rightleftharpoons \text{nop} \rightarrow I_1 \rightleftharpoons I_2 \rightleftharpoons I_3 \rightleftharpoons I_4 \rightleftharpoons \text{pipeline bubble}

**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
I4 304 ADD
Jump Pipeline Diagrams

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

\[
\begin{align*}
& \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{MA} \quad \text{WB} \\
(I_1) & \quad \text{096: ADD} \\
(I_2) & \quad \text{100: J 200} \\
(I_3) & \quad \text{104: ADD} \\
(I_4) & \quad \text{304: ADD}
\end{align*}
\]

\[
\begin{align*}
& \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{MA} \quad \text{WB} \\
(I_1) & \quad \text{096: ADD} \\
(I_2) & \quad \text{100: J 200} \\
(I_3) & \quad \text{104: ADD} \\
(I_4) & \quad \text{304: ADD}
\end{align*}
\]

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

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

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

\[
\begin{align*}
& \text{nop} \quad \Rightarrow \quad \text{pipeline bubble}
\end{align*}
\]
Pipelining Conditional Branches

Branch condition is not known until the execute stage

what action should be taken in the decode stage?

September 22, 2021

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
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

\[
\text{stall} = \left( \left( (rs_{D}==ws_{E})\cdot we_{E} + (rs_{D}==ws_{M})\cdot we_{M} + (rs_{D}==ws_{W})\cdot we_{W})\cdot re_{1D} \\
+ ((rt_{D}==ws_{E})\cdot we_{E} + (rt_{D}==ws_{M})\cdot we_{M} + (rt_{D}==ws_{W})\cdot we_{W})\cdot re_{2D} \right) \cdot !((\text{opcode}_{E}==\text{BEQZ})\cdot z + (\text{opcode}_{E}==\text{BNEZ})\cdot \neg z) \right)
\]

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

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

IRSrc_D = Case opcode_E
BEQZ·z, BNEZ·!z ⇒ nop
...
⇒ Case opcode_D
J, JAL, JR, JALR ⇒ nop
...
⇒ IM

IRSrc_E = Case opcode_E
BEQZ·z, BNEZ·!z ⇒ nop
...
⇒ stall·nop + !stall·IR_D

PCSrc = Case opcode_E
BEQZ·z, BNEZ·!z ⇒ br
...
⇒ Case opcode_D
J, JAL ⇒ jabs
JR, JALR ⇒ rind
...
⇒ pc+4

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

pc+4 is a speculative guess

nop ⇒ Kill
br/jabs/rind ⇒ Restart
pc+4 ⇒ Speculate
Branch Pipeline Diagrams
(resolved in execute stage)

\[\begin{array}{cccccccc}
\text{time} & t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 & \ldots \ldots \\
\hline
(\text{I}_1) 096: \text{ADD} & \text{IF}_1 & \text{ID}_1 & \text{EX}_1 & \text{MA}_1 & \text{WB}_1 & \\
(\text{I}_2) 100: \text{BEQZ} 200 & \text{IF}_2 & \text{ID}_2 & \text{EX}_2 & \text{MA}_2 & \text{WB}_2 & \\
(\text{I}_3) 104: \text{ADD} & \text{IF}_3 & \text{ID}_3 & \text{MA}_3 & \text{WB}_3 & \\
(\text{I}_4) 108: \text{ADD} & \text{IF}_4 & \text{ID}_4 & \text{MA}_4 & \text{WB}_4 & \\
(\text{I}_5) 304: \text{ADD} & \text{IF}_5 & \text{ID}_5 & \text{EX}_5 & \text{MA}_5 & \text{WB}_5 & \\
\end{array}\]

**Resource Usage**

- IF: I_1, I_2, I_3, I_4, I_5, I_6
- ID: I_1, I_2, I_3, I_4, I_5
- EX: I_1, I_2, I_3, I_4, I_5
- MA: I_1, I_2, I_3, I_4, I_5
- WB: I_1, I_2, I_3, I_4, I_5

* 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.
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.

<table>
<thead>
<tr>
<th>I_1</th>
<th>096</th>
<th>ADD</th>
</tr>
</thead>
<tbody>
<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*
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