Instruction Pipelining:
Hazard Resolution, Timing Constraints

Mengjia Yan
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Pipeline Diagram – Ideal Pipelining

\[
\begin{align*}
\text{time} & \quad t0 \quad t1 \quad t2 \quad t3 \quad t4 \quad t5 \quad t6 \quad t7 \quad \ldots \\
(I_1) \quad a1 & \leftarrow a0 + 10 & \text{IF}_1 & \text{ID}_1 & \text{EX}_1 & \text{MA}_1 & \text{WB}_1 \\
(I_2) \quad a3 & \leftarrow a2 + 12 & \text{IF}_2 & \text{ID}_2 & \text{EX}_2 & \text{MA}_2 & \text{WB}_2 \\
(I_3) \quad a5 & \leftarrow a4 + 14 & \text{IF}_3 & \text{ID}_3 & \text{EX}_3 & \text{MA}_3 & \text{WB}_3 \\
(I_4) \quad a7 & \leftarrow a6 + 16 & \text{IF}_4 & \text{ID}_4 & \text{EX}_4 & \text{MA}_4 & \text{WB}_4 \\
(I_5) \quad s1 & \leftarrow s0 + 18 & \text{IF}_5 & \text{ID}_5 & \text{EX}_5 & \text{MA}_5 & \text{WB}_5 \\
\end{align*}
\]

Over long term, i.e., in steady state, what are the ...

Throughput (T) \quad Latency (L)

CPI? \quad IPC? \quad Inst exec time?

\[ N \]

Num inst in flight?
Reminder: Pipelined RISC-V 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 \(\rightarrow\) structural hazard

• An instruction may depend on a value produced by an earlier instruction
  
  – Dependence may be for a data calculation \(\rightarrow\) data hazard
  
  – Dependence may be for calculating the next PC \(\rightarrow\) control hazard (branches, interrupts)
Data Hazards

... a1 ← a0 + 10
a3 ← a1 + 12 ...

Cycle N+1

a1 is stale. Oops!
Pipeline Diagram – Hazard

\[ t(t) \]
\[ (I_1) \ a_1 \leftarrow a_0 + 10 \]
\[ (I_2) \ a_3 \leftarrow a_1 + 12 \]

\[ \text{IF}_1 \quad \text{ID}_1 \quad \text{EX}_1 \quad \text{MA}_1 \quad \text{WB}_1 \]
\[ \text{IF}_2 \quad \text{ID}_2 \quad \text{EX}_2 \quad \text{MA}_2 \quad \text{WB}_2 \]

\text{a1 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* \( \rightarrow \text{stall} \)

<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>ID</td>
<td>IF1</td>
<td>ID1</td>
<td>EX1</td>
<td>MA1</td>
<td>WB1</td>
<td>ID2</td>
<td>ID2</td>
<td>ID2</td>
<td>ID2</td>
</tr>
<tr>
<td>IF2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stall DEC & IF when instruction in DEC reads a register that is written by any earlier in-flight instruction (in EXE, MEM, or WB)
Compare the *source registers* of the instruction in the decode stage with the *destination register of the uncommitted instructions*. 
Reminder: Stall Control Logic

**ignoring jumps & branches**

Should we always stall if the rs field(s) matches some rd?
Source & Destination Registers

R-type:

\[
\begin{array}{ccccccc}
\text{funct7} & \text{rs2} & \text{rs1} & \text{funct3} & \text{rd} & \text{opcode} \\
\end{array}
\]

I-type:

\[
\begin{array}{ccccccc}
\text{imm}[11:0] & \text{rs1} & \text{funct3} & \text{rd} & \text{opcode} \\
\text{imm}[11:5] & \text{rs2} & \text{rs1} & \text{func3} & \text{imm}[4:0] & \text{opcode} \\
\end{array}
\]

source(s)  destination

ALU \quad rd \leftarrow (rs1) \text{ func } (rs2) \quad \text{rs1, rs2} \quad \text{rd}

ALUi \quad rd \leftarrow (rs1) \text{ funct SXT}(\text{imm}) \quad \text{rs1} \quad \text{rd}

LW \quad rd \leftarrow M \left[ (rs1) + \text{imm.disp} \right] \quad \text{rs1} \quad \text{rd}

SW \quad M \left[ (rs1) + \text{imm.disp} \right] \leftarrow (rs2) \quad \text{rs1, rs2}

BEQ \quad cond \ (rs1)==(rs2)

\begin{align*}
\text{true:} & \quad \text{PC} \leftarrow \text{(PC)} + \text{SXT}(\text{imm}) \quad \text{rs1, rs2} \\
\text{false:} & \quad \text{PC} \leftarrow \text{(PC)} + 4 \quad \text{rs1, rs2}
\end{align*}

JAL \quad rd \leftarrow (\text{PC+4}), \text{ PC} \leftarrow (\text{PC}) + \text{imm} \quad \text{rd}

JALR \quad rd \leftarrow (\text{PC+4}), \text{ PC} \leftarrow (rs1) + \text{imm}^* \quad \text{rs1} \quad \text{rd}

*More precise: \(pc \leftarrow (\text{reg}[rs1] + \text{imm})[31:1], 1'b0\)
Deriving the Stall Signal

\[
C_{\text{dest}} \quad \text{we} = \text{Case opcode} \\
\text{ALU, ALUi, LW, JAL, JALR} \quad \Rightarrow (\text{rd} \neq 0) \\
\ldots \quad \Rightarrow \text{off}
\]

\[
C_{\text{re}} \\
\text{re1} = \text{Case opcode} \\
\text{ALU, ALUi, LW, SW, BEQ, JR, JALR} \quad \Rightarrow \text{on} \\
\ldots \quad \Rightarrow \text{off}
\]

\[
C_{\text{stall}} \quad \text{stall} = ((\text{rs1}_D = \text{rd}_E) \cdot \text{we}_E + \\
(\text{rs1}_D = \text{rd}_M) \cdot \text{we}_M + \\
(\text{rs1}_D = \text{rd}_W) \cdot \text{we}_W) \cdot \text{re1}_D \\
+ \\
((\text{rs2}_D = \text{rd}_E) \cdot \text{we}_E + \\
(\text{rs2}_D = \text{rd}_M) \cdot \text{we}_M + \\
(\text{rs2}_D = \text{rd}_W) \cdot \text{we}_W) \cdot \text{re2}_D
\]

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

Stall Condition

... M[(a1)+7] ← (a2)
a4 ← M[(a3)+5]
...

Is there any possible data hazard in this instruction sequence?
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) \[ a₁ ← a₀ + 10 \]
(I₂) \[ a₃ ← a₁ + 12 \]

a₁ ← M[a₀ + 10]
a₃ ← a₁ + 12

JAL ra 500
t₃ ← ra + 12

September 20, 2023
The Bypass Signal
Deriving it from the Stall Signal

\[
\text{stall} = ((\text{rs}_1 = \text{rd}_E) \cdot \text{we}_E + (\text{rs}_1 = \text{rd}_M) \cdot \text{we}_M + (\text{rs}_1 = \text{rd}_W) \cdot \text{we}_W) \cdot \text{re}_1_D \\
+ ((\text{rs}_2 = \text{rd}_E) \cdot \text{we}_E + (\text{rs}_2 = \text{rd}_M) \cdot \text{we}_M + (\text{rs}_2 = \text{rd}_W) \cdot \text{we}_W) \cdot \text{re}_2_D
\]

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

\[
\text{ASrc} = (\text{rs}_1 = \text{rd}_E) \cdot \text{we}_E \cdot \text{re}_1_D
\]

Is this correct?

How might we address this?
Bypass and Stall Signals

Split \( wE \) into two components: \( wE\)-bypass, \( wE\)-stall

\[
\begin{align*}
\text{we-bypass}_E &= \text{Case opcode}_E \\
&= \begin{cases} 
\text{ALU, ALUi} & \Rightarrow (rd \neq 0) \\
... & \Rightarrow \text{off}
\end{cases} \\
\text{we-stall}_E &= \text{Case opcode}_E \\
&= \begin{cases} 
\text{ opcode } \in \{\text{LW, JAL, JALR}\} & \Rightarrow (rd \neq 0) \\
... & \Rightarrow \text{off}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{ASrc} &= (rs1_D == rd_E) \cdot \text{re1}_D \cdot \text{we-bypass}_E \\
\text{stall} &= ((rs1_D == rd_E) \cdot \text{we-stall}_E + \\
&\quad (rs1_D == rd_M) \cdot \text{we}_M + (rs1_D == rd_W) \cdot \text{we}_W) \cdot \text{re1}_D \\
&\quad + ((rs2_D == rd_E) \cdot \text{we}_E + (rs2_D == rd_M) \cdot \text{we}_M + (rs2_D == rd_W) \cdot \text{we}_W) \cdot \text{re2}_D
\end{align*}
\]
Is there still a need for the stall signal?

\[
\text{stall} = (\text{rs}_1^D = = \text{rd}_E \cdot (\text{opcode}_E = = \text{LW}_E \cdot (\text{rd}_E \not= 0) \cdot \text{re}_1^D ) + (\text{rs}_2^D = = \text{rd}_E \cdot (\text{opcode}_E = = \text{LW}_E \cdot (\text{rd}_E \not= 0) \cdot \text{re}_2^D )
\]
Full Bypass Datapath

Challenge: How to scale bypass datapaths for more pipeline stages? More complex ISAs?
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 (JAL)
  – For Jump Register (JALR)
  – For Conditional Branches (e.g., BEQ)
  – For all others (e.g., arithmetic insts)

• In what stage do we know these?
  – PC
  – Opcode, offset
  – Register value
  – Branch condition ((rs1)== (rs2))
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>(I_1) a_1 ← (a_0) + 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>
<td></td>
</tr>
<tr>
<td>(I_2) a_3 ← (a_2) + 17</td>
<td>IF_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(I_3)</td>
<td>IF_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(I_4)</td>
<td>IF_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Resource Usage Diagram**

<table>
<thead>
<tr>
<th>Resource</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>IF</td>
<td>I_1</td>
<td>nop</td>
<td>I_2</td>
<td>nop</td>
<td>I_3</td>
<td>nop</td>
<td>I_4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>I_1</td>
<td>nop</td>
<td>I_2</td>
<td>nop</td>
<td>I_3</td>
<td>nop</td>
<td>I_4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>I_1</td>
<td>nop</td>
<td>I_2</td>
<td>nop</td>
<td>I_3</td>
<td>nop</td>
<td>I_4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MA</td>
<td>I_1</td>
<td>nop</td>
<td>I_2</td>
<td>nop</td>
<td>I_3</td>
<td>nop</td>
<td>I_4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>I_1</td>
<td>nop</td>
<td>I_2</td>
<td>nop</td>
<td>I_3</td>
<td>nop</td>
<td>I_4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*nop ⇒ pipeline bubble*

What’s a good guess for next PC?

September 20, 2023
Speculate NextPC is PC+4

PCSrc (pc+4 / evec / rs1+imm / pc+imm)

stall

Jump?
Speculate NextPC is PC+4

I
1
096
ADD

I
2
100
JAL ra 200

I
3
104
ADD

I
4
304
ADD

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

How?
Pipelining Jumps

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

**PCS**

\[
\text{PCS} = \left( pc + 4 / \text{evc} / rs1 + \text{imm} / pc + \text{imm} \right)
\]

I

1

096

ADD

100

JAL

200

ra

I

2

104

ADD

I

3

104

kill

I

4

304

ADD

\[
\text{IR}_{D} = \text{Case} \quad \text{opcode}_{D}
\]

JAL, JALR \rightarrow \text{nop}

\[
\ldots \rightarrow \text{IM}
\]
Pipelining Jumps

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

Any interaction between stall and jump?

PCSr (pc+4 / evec / rs1+imm / pc+imm)

I_1  096  ADD
I_2  100  JAL ra 200
I_3  104  ADD  kill
I_4  304  ADD

IRSrc_D

IRSrc_D = Case opcode_D
JAL, JALR  ⇒ nop
...  ⇒ IM
Jump Pipeline Diagrams

```
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: JAL ra 200
  IF_2  ID_2  EX_2  MA_2  WB_2

(I_3) 104: ADD
  IF_3    nop    nop    nop    nop
  IF_4  ID_4  EX_4  MA_4  WB_4

(I_4) 304: ADD

Resource Usage Diagram

<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>time</td>
<td></td>
<td>I_1</td>
<td>I_2</td>
<td>I_3</td>
<td>I_4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I_1</td>
<td></td>
<td>I_2</td>
<td>I_4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I_1</td>
<td>I_1</td>
</tr>
</tbody>
</table>

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?*
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

I_1 096  ADD
I_2 100  BEQ a1 a2 200
I_3 104  ADD
I_4 304  ADD
Pipelining Conditional Branches

Don’t stall if the branch is taken. Why?
Control Equations for PC and IR Muxes

\[
\text{IRSrc}_D = \text{Case opcode}_E \\
\quad \text{BEQ} \cdot z \Rightarrow \text{nop} \Rightarrow \text{Case opcode}_D \\
\quad \ldots \Rightarrow \text{IM}
\]

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

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

\[
\text{PCSrc} = \text{Case opcode}_E \\
\quad \text{BEQ} \cdot z \Rightarrow \text{pc} + \text{imm} \\
\quad \ldots \Rightarrow \text{Case opcode}_D \\
\quad \text{JAL} \Rightarrow \text{pc} + \text{imm} \\
\quad \text{JALR} \Rightarrow \text{rs1} + \text{imm} \\
\quad \ldots \Rightarrow \text{pc}+4
\]

pc+4 is a speculative guess

\[
\text{nop} \Rightarrow \text{Kill} \\
\text{pc}+\text{imm} / \text{rs1}+\text{imm} \Rightarrow \text{Restart} \\
\text{pc}+4 \Rightarrow \text{Speculate}
\]
Branch Pipeline Diagrams
(resolved in execute stage)

\begin{align*}
\text{time} & \quad t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 & \ldots \\
(I_1) & 096: \text{ADD} & IF_1 & ID_1 & EX_1 & MA_1 & WB_1 \\
(I_2) & 100: \text{BEQ a1 a2 200} & ID_2 & EX_2 & MA_2 & WB_2 \\
(I_3) & 104: \text{ADD} & IF_3 & ID_3 & hop & hop & hop \\
(I_4) & 108: & IF_4 & ID_4 & hop & hop & hop \\
(I_5) & 304: \text{ADD} & IF_5 & ID_5 & EX_5 & MA_5 & WB_5 \\
\end{align*}

\text{Resource Usage Diagram}

\begin{align*}
\text{time} & \quad t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 & \ldots \\
\text{IF} & I_1 & I_2 & I_3 & I_4 & I_5 \\
\text{ID} & I_1 & I_2 & I_3 & \text{nop} & I_5 \\
\text{EX} & I_1 & I_2 & \text{nop} & \text{nop} & I_5 \\
\text{MA} & I_1 & I_2 & \text{nop} & \text{nop} & \text{nop} & I_5 \\
\text{WB} & I_1 & I_2 & \text{nop} & \text{nop} & \text{nop} & 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.

Comparison result generated on register file output

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.

\[
\begin{array}{ccc}
I_1 & 096 & ADD \\
I_2 & 100 & BEQ a1 a2 200 \\
I_3 & 104 & ADD \\
I_4 & 304 & ADD \\
\end{array}
\]

- 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