Instruction Pipelining:
Hazard Resolution, Timing Constraints

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

\[\begin{array}{cccccccccc}
& t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 & \ldots \\
(I_1) & a1 & \leftarrow & a0 & + & 10 & IF_1 & ID_1 & EX_1 & MA_1 & WB_1 \\
(I_2) & a3 & \leftarrow & a2 & + & 12 & IF_2 & ID_2 & EX_2 & MA_2 & WB_2 \\
(I_3) & a5 & \leftarrow & a4 & + & 14 & IF_3 & ID_3 & EX_3 & MA_3 & WB_3 \\
(I_4) & a7 & \leftarrow & a6 & + & 16 & IF_4 & ID_4 & EX_4 & MA_4 & WB_4 \\
(I_5) & s1 & \leftarrow & s0 & + & 18 & IF_5 & ID_5 & EX_5 & MA_5 & WB_5 \\
\end{array}\]

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

Throughput (T)  \quad Latency (L)

CPI?  \quad 1  \quad IPC?  \quad 1  \quad Inst. exec. time?  \quad 5

\[\bar{T} = \frac{N}{L} \quad \rightarrow \quad 5\]

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 → 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

Cycle N+1

a1 is stale. Oops!

... a1 ← a0 + 10
a3 ← a1 + 12 ...

September 20, 2023
Pipeline Diagram – Hazard

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

time

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

\[ IF_1 \ \ \ ID_1 \ \ \ EX_1 \ \ \ MA_1 \ \ \ WB_1 \]
\[ IF_2 \ \ \ ID_2 \ \ \ EX_2 \ \ \ MA_2 \ \ \ WB_2 \]

\[ a_1 \text{ is stale. Oops!} \]
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)
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?
not every instruction writes a register ⇒ we (write enable)
not every instruction reads a register ⇒ re (read enable)
Source & Destination Registers

\[\text{R-type:} \quad \begin{array}{cccccc}
\text{funct7} & \text{rs2} & \text{rs1} & \text{funct3} & \text{rd} & \text{opcode}
\end{array}\]

\[\text{I-type:} \quad \begin{array}{cccccc}
\text{imm}[11:0] & \text{rs1} & \text{funct3} & \text{rd} & \text{opcode}
\end{array}\]

\[\text{imm}[11:5] \quad \text{rs2} \quad \text{rs1} \quad \text{funct3} \quad \text{imm}[4:0] \quad \text{opcode}\]

**source(s)** \hspace{1cm} **destination**

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

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

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

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

BEQ \quad \text{cond } (\text{rs1})==(\text{rs2})

\quad \text{true: } \text{PC} \leftarrow (\text{PC}) + \text{SXT(imm)} \quad \text{rs1, rs2}

\quad \text{false: } \text{PC} \leftarrow (\text{PC}) + 4 \quad \text{rs1, rs2}

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

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

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

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

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

\[
\text{re1} = \begin{cases} 
\text{Case opcode} \\
\text{ALU, ALUi, LW, SW, BEQ, JR, JALR} 
\end{cases} \\
\Rightarrow \text{on}
\]

\[
\text{re2} = \begin{cases} 
\text{Case opcode} \\
\text{ALU, SW, BEQ} 
\end{cases} \\
\Rightarrow \text{on} \\
\ldots \Rightarrow \text{off}
\]

\[ C_{\text{stall}} \]

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

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

Stall Condition

Is there any possible data hazard in this instruction sequence?

What if \((a1) + 7 = (a3) + 5\) ?

\[
M[(a1)+7] \leftrightarrow (a2) \\
a4 \leftrightarrow M[(a3)+5]
\]
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 \textit{stall} or \textit{kill} introduces a bubble $\Rightarrow CPI > 1$

\textbf{When is data actually available?} \quad \textbf{At Execute}

A new datapath, i.e., a \textit{bypass}, can get the data from the output of the ALU to its input
Adding a Bypass

When does this bypass help?

(I_1) \( a1 \leftarrow a0 + 10 \)
(I_2) \( a3 \leftarrow a1 + 12 \)

\( JAL \ ra \ 500 \)
\( t3 \leftarrow ra + 12 \)

yes  no  no

September 20, 2023

MIT 6.5900 Fall 2023
The Bypass Signal
Deriving it from the Stall Signal

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

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

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

Is this correct?
No, only ALU and ALUi instructions can benefit from this bypass. Memory and JAL* instructions cannot.

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: \( w_{\text{bypass}} \), \( w_{\text{stall}} \)

\[
\begin{align*}
\text{we-bypass}_E &= \text{Case opcode}_E \\
&\quad \text{ALU, ALUi} \quad \Rightarrow (\text{rd} \neq 0) \\
&\quad \ldots \quad \Rightarrow \text{off}
\end{align*}
\]

\[
\begin{align*}
\text{we-stall}_E &= \text{Case opcode}_E \\
&\quad \text{LW, JAL, JALR} \quad \Rightarrow (\text{rd} \neq 0) \\
&\quad \ldots \quad \Rightarrow \text{off}
\end{align*}
\]

\[
\begin{align*}
\text{ASrc} &= (\text{rs1}_D \text{== rd}_E) \cdot \text{re1}_D \cdot \text{we-bypass}_E \\
\text{stall} &= ((\text{rs1}_D \text{== rd}_E) \cdot \text{we-stall}_E + \\
&\quad (\text{rs1}_D \text{== rd}_M) \cdot \text{we}_M + (\text{rs1}_D \text{== rd}_W) \cdot \text{we}_W) \cdot \text{re1}_D \\
&\quad + ((\text{rs2}_D \text{== rd}_E) \cdot \text{we}_E + (\text{rs2}_D \text{== rd}_M) \cdot \text{we}_M + (\text{rs2}_D \text{== rd}_W) \cdot \text{we}_W) \cdot \text{re2}_D
\end{align*}
\]
Is there still a need for the stall signal?

\[
\text{stall} = (rs1_D == rd_E) \cdot (\text{opcode}_E == \text{LW}_E) \cdot (rd_E \neq 0) \cdot \text{re1}_D \\
+ (rs2_D == rd_E) \cdot (\text{opcode}_E == \text{LW}_E) \cdot (rd_E \neq 0) \cdot \text{re2}_D
\]
Full Bypass Datapath

Challenge: How to scale bypass datapaths for more pipeline stages? More complex ISAs?
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 (JAL)
    • Opcode, offset, and PC
  – For Jump Register (JALR)
    • Opcode, offset, register value
  – For Conditional Branches (e.g., BEQ)
    • Opcode, offset, PC, and register (for condition)
  – For all others (e.g., arithmetic insts)
    • 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 \((\text{rs1} == \text{rs2})\) \rightarrow Execute (or Decode?)
NextPC Calculation Bubbles

Time sequence:

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

Steps:

(I_1) \( a_1 \leftarrow (a_0) + 10 \) \( \text{IF}_1 \)

(I_2) \( a_3 \leftarrow (a_2) + 17 \) \( \text{IF}_2 \)

(I_3) \( \text{ID} \)

(I_4) \( \text{EX} \)

(I_5) \( \text{MA} \)

(I_6) \( \text{WB} \)

Resource Usage Diagram:

\text{IF} \quad \text{I}_1 \quad \text{nop} \quad \text{I}_2 \quad \text{nop} \quad \text{I}_3 \quad \text{nop} \quad \text{I}_4 \quad \ldots

\text{ID} \quad \text{I}_1 \quad \text{nop} \quad \text{I}_2 \quad \text{nop} \quad \text{I}_3 \quad \text{nop} \quad \text{I}_4 \quad \ldots

\text{EX} \quad \text{I}_1 \quad \text{nop} \quad \text{I}_2 \quad \text{nop} \quad \text{I}_3 \quad \text{nop} \quad \text{I}_4 \quad \ldots

\text{MA} \quad \text{I}_1 \quad \text{nop} \quad \text{I}_2 \quad \text{nop} \quad \text{I}_3 \quad \text{nop} \quad \text{I}_4 \quad \ldots

\text{WB} \quad \text{I}_1 \quad \text{nop} \quad \text{I}_2 \quad \text{nop} \quad \text{I}_3 \quad \text{nop} \quad \text{I}_4 \quad \ldots

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

What’s a good guess for next PC? \( \text{PC}+4 \)
Speculate NextPC is PC+4

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

stall

Jump?
Speculate \textbf{NextPC is PC+4}

\begin{itemize}
  \item \textbf{I} \text{1} 096 \text{ADD}
  \item \textbf{I} \text{2} 100 \text{JAL ra 200}
  \item \textbf{I} \text{3} 104 \text{ADD} \textit{kill}
  \item \textbf{I} \text{4} 304 \text{ADD}
\end{itemize}

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

\textbf{How?}
Pipelining Jumps

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

IRSrcD = Case opcodeD
JAL, JALR ⇒ nop
... ⇒ IM

I1 096 ADD
I2 100 JAL ra 200
I3 104 ADD  **kill**
I4 304 ADD
Pipelining Jumps

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

Any interaction between stall and jump? No for now

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

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

Time

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

\((I_1)\) 096: ADD
\((I_2)\) 100: JAL ra 200
\((I_3)\) 104: ADD
\((I_4)\) 304: ADD

Resource Usage Diagram

<table>
<thead>
<tr>
<th>Time</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_2</td>
<td>I_3</td>
<td>I_4</td>
</tr>
<tr>
<td>t1</td>
<td>I_2</td>
<td>I_1</td>
<td>I_2</td>
<td>I_4</td>
<td>I_5</td>
</tr>
<tr>
<td>t2</td>
<td>I_3</td>
<td>I_2</td>
<td>I_2</td>
<td>I_4</td>
<td>I_5</td>
</tr>
<tr>
<td>t3</td>
<td>I_4</td>
<td>I_2</td>
<td>I_2</td>
<td>I_4</td>
<td>I_5</td>
</tr>
<tr>
<td>t4</td>
<td>I_5</td>
<td>I_2</td>
<td>I_2</td>
<td>I_4</td>
<td>I_5</td>
</tr>
<tr>
<td>t5</td>
<td>I_6</td>
<td>I_2</td>
<td>I_2</td>
<td>I_4</td>
<td>I_5</td>
</tr>
<tr>
<td>t6</td>
<td>I_7</td>
<td>I_2</td>
<td>I_2</td>
<td>I_4</td>
<td>I_5</td>
</tr>
<tr>
<td>t7</td>
<td>I_8</td>
<td>I_2</td>
<td>I_2</td>
<td>I_4</td>
<td>I_5</td>
</tr>
</tbody>
</table>

\( \text{nop} \Rightarrow \text{pipeline bubble} \)
Pipelining Conditional Branches

Branch condition is not known until the execute stage. *What action should be taken in the decode stage?*

Continue speculating. Decode I2 and fetch PC+4
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

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

Instruction at the decode stage is invalid.

I₁  096  ADD
I₂  100  BEQ a1 a2 200
I₃  104  ADD
I₄  304  ADD
Control Equations for PC and IR Muxes

IRSrc\(_D = \text{Case opcode}_E\)

\[
\begin{align*}
\text{BEQ}\cdot z & \Rightarrow \text{nop} \\
\ldots & \Rightarrow \text{Case opcode}_D \\
\text{JAL, JALR} & \Rightarrow \text{nop} \\
\ldots & \Rightarrow \text{IM}
\end{align*}
\]

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

IRSrc\(_E = \text{Case opcode}_E\)

\[
\begin{align*}
\text{BEQ}\cdot z & \Rightarrow \text{nop} \\
\ldots & \Rightarrow \text{stall}\cdot\text{nop} + \neg\text{stall}\cdot\text{IR}_D
\end{align*}
\]

PCSrc = \text{Case opcode}_E

\[
\begin{align*}
\text{BEQ}\cdot z & \Rightarrow \text{pc} + \text{imm} \\
\ldots & \Rightarrow \text{Case opcode}_D \\
\text{JAL} & \Rightarrow \text{pc} + \text{imm} \\
\text{JALR} & \Rightarrow \text{rs1} + \text{imm} \\
\ldots & \Rightarrow \text{pc} + 4 \\
\end{align*}
\]

\text{pc}+4 is a speculative guess

copc+imm / rs1+imm \Rightarrow \text{Restart}

nop \Rightarrow \text{Kill}

pc+4 \Rightarrow \text{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: BEQ a1 a2 200 IF_2  ID_2  EX_2  MA_2  WB_2
(I_3) 104: ADD  IF_3  ID_3  nop  nop  nop
(I_4) 108:  
(I_5) 304: ADD  IF_5  ID_5  EX_5  MA_5  WB_5

Resource Usage Diagram
(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.

PCS\textsubscript{src} (pc+4 / evec / rs1+imm / pc+imm)

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.

  $\begin{array}{l}
  I_1 \quad 096 \quad \text{ADD} \\
  I_2 \quad 100 \quad \text{BEQ} \ a1\ a2\ 200 \\
  I_3 \quad 104 \quad \text{ADD} \\
  I_4 \quad 304 \quad \text{ADD}
  \end{array}$

  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?

September 20, 2023
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