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
Stall DEC & IF when instruction in DEC reads a register that is written by any earlier in-flight instruction (in EXE, MEM, or WB)
Reminder: Load & Store Hazards

Stall Condition

M[(r1)+7] ← (r2)
r4 ← M[(r3)+5]

(r1)+7 = (r3)+5 ⇒ data hazard

These hazards do not need pipeline changes because our memory system completes writes in one cycle
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 $\Rightarrow CPI > 1$

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?

(I_1) \( r1 \leftarrow r0 + 10 \) yes

(I_2) \( r4 \leftarrow r1 + 17 \) yes

r1 \( \leftarrow M[r0 + 10] \) no

r1 \( \leftarrow r1 + 17 \) no

JAL 500 \( r4 \leftarrow r31 + 17 \) no
The Bypass Signal
Deriving it from the Stall Signal

\[
\text{stall} = \begin{align*}
&\left((\text{rs}_D = \text{ws}_E) \cdot \text{we}_E + (\text{rs}_D = \text{ws}_M) \cdot \text{we}_M + (\text{rs}_D = \text{ws}_W) \cdot \text{we}_W\right) \cdot \text{re}_{1D} \\
&+ \left((\text{rt}_D = \text{ws}_E) \cdot \text{we}_E + (\text{rt}_D = \text{ws}_M) \cdot \text{we}_M + (\text{rt}_D = \text{ws}_W) \cdot \text{we}_W\right) \cdot \text{re}_{2D}
\end{align*}
\]

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

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

\[
\text{ASrc} = (\text{rs}_D = \text{ws}_E) \cdot \text{we}_E \cdot \text{re}_{1D}
\]

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: \text{we}-bypass, \text{we}-stall
Bypass and Stall Signals

Split $w_{\text{E}}$ into two components: we-bypass, we-stall

\[
\text{we-bypass}_{\text{E}} = \text{Case } \text{opcode}_{\text{E}} \text{ ALU, ALUi } \Rightarrow (ws \neq 0) \\
\text{... } \Rightarrow \text{off}
\]

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

\[
\text{ASrc } = (rs_{\text{D}} == ws_{\text{E}}) \cdot \text{we-bypass}_{\text{E}} \cdot \text{re}_{\text{1D}}
\]

\[
\text{stall } = ((rs_{\text{D}} == ws_{\text{E}}) \cdot \text{we-stall}_{\text{E}} + \\
(rs_{\text{D}} == ws_{\text{M}}) \cdot \text{we}_{\text{M}} + (rs_{\text{D}} == ws_{\text{W}}) \cdot \text{we}_{\text{W}}) \cdot \text{re}_{\text{1D}}
\]

\[
+ ((rt_{\text{D}} == ws_{\text{E}}) \cdot \text{we}_{\text{E}} + (rt_{\text{D}} == ws_{\text{M}}) \cdot \text{we}_{\text{M}} + (rt_{\text{D}} == ws_{\text{W}}) \cdot \text{we}_{\text{W}}) \cdot \text{re}_{\text{2D}}
\]
Is there still a need for the stall signal?

\[
stall = (rs_D == ws_E) \cdot (opcode_E == LW_E) \cdot (ws_E \neq 0) \cdot re1_D + (rt_D == ws_E) \cdot (opcode_E == LW_E) \cdot (ws_E \neq 0) \cdot re2_D
\]
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

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  **kill**
I_4  304  ADD

How?
Pipelining Jumps

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

Any interaction between stall and jump?

\[ \text{IRSrc}_D = \text{Case opcode}_D \]
\[ \begin{align*}
\text{J, JAL} & \Rightarrow \text{nop} \\
\text{...} & \Rightarrow \text{IM}
\end{align*} \]

I_1 096 ADD
I_2 100 J 200
I_3 104 ADD
I_4 304 ADD

March 11, 2021
Jump Pipeline Diagrams

\begin{align*}
&\text{time} \\
&t0 \quad t1 \quad t2 \quad t3 \quad t4 \quad t5 \quad t6 \quad t7 \quad \ldots \ldots \\
&(I_1) \ 096: \text{ADD} \\
&(I_2) \ 100: \text{J \ 200} \\
&(I_3) \ 104: \text{ADD} \\
&(I_4) \ 304: \text{ADD}
\end{align*}

Resource Usage

\begin{align*}
&\text{IF} \\
&\quad I_1 \\
&\text{ID} \\
&\quad I_1 \\
&\text{EX} \\
&\quad I_1 \\
&\text{MA} \\
&\quad I_1 \\
&\text{WB} \\
&\quad I_1
\end{align*}

nop \Rightarrow \text{pipeline bubble}
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
Don’t stall if the branch is taken. Why?

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

**IRSrc**

\[ IR_{SrcD} = \text{Case opcode}_E \]
\[ \text{BEQZ} \cdot z, \text{BNEZ} \cdot !z \quad \Rightarrow \text{nop} \]
\[ \quad \Rightarrow \]
\[ \text{Case opcode}_D \]
\[ J, JAL, JR, JALR \quad \Rightarrow \text{nop} \]
\[ \quad \Rightarrow \text{IM} \]

**IRSrc**

\[ IR_{SrcE} = \text{Case opcode}_E \]
\[ \text{BEQZ} \cdot z, \text{BNEZ} \cdot !z \quad \Rightarrow \text{nop} \]
\[ \quad \Rightarrow \]
\[ \text{stall} \cdot \text{nop} + \text{!stall} \cdot IR_{D} \]

**PCSrc**

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

\[ PCSrc = \text{Case opcode}_E \]
\[ \text{BEQZ} \cdot z, \text{BNEZ} \cdot !z \quad \Rightarrow \text{br} \]
\[ \quad \Rightarrow \]
\[ \text{Case opcode}_D \]
\[ J, JAL \quad \Rightarrow \text{jabs} \]
\[ JR, JALR \quad \Rightarrow \text{rind} \]
\[ \quad \Rightarrow \text{pc}+4 \]

\[ \text{nop} \quad \Rightarrow \text{Kill} \]
\[ \text{br/jabs/rind} \quad \Rightarrow \text{Restart} \]
\[ \text{pc}+4 \quad \Rightarrow \text{Speculate} \]
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: BEQZ 200
    IF_2  ID_2  EX_2  MA_2  WB_2

(I_3) 104: ADD
    IF_3  ID_3  MA_2  WB_2

(I_4) 108:
    IF_4  ID_4  EX_5  MA_5  WB_5

(I_5) 304: ADD
    IF_5  ID_5  EX_5  MA_5  WB_5

Resource Usage

IF  I_1  I_2  I_3  I_4  I_5
ID  I_1  I_2  I_3  nop  I_5
EX  I_1  I_2  nop  nop  I_5
MA  I_1  I_2  nop  nop  I_5
WB  I_1  I_2  nop  nop  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

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