Pipeline Hazards

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Based on the material prepared by
Arvind and Krste Asanovic
Technology Assumptions

- A small amount of very fast memory (caches) backed up by a large, slower memory
- Fast ALU (at least for integers)
- Multiported Register files (slower!)

It makes the following timing assumption valid

\[ t_{IM} \approx t_{RF} \approx t_{ALU} \approx t_{DM} \approx t_{RW} \]

A 5-stage pipelined Harvard architecture will be the focus of our detailed design
5-Stage Pipelined Execution

- **I-Fetch (IF)**
  - Instruction Memory
  - PC
  - Addr
  - Rdata
  - Inst.
- **Decode, Reg. Fetch (ID)**
  - IR
  - Imm Ext
  - Rs1
  - Rs2
  - Rd1
  - We
  - Ws
  - Rd2
  - GPRs
- **Execute (EX)**
  - ALU
  - Rd1
  - Rdata
  - Ext
  - Imm
  - Rs1
  - Rs2
  - Ws
  - Wdrd2
- **Memory (MA)**
  - Data Memory
  - Addr
  - Rdata
- **Write-Back (WB)**

### Time
- t0
- t1
- t2
- t3
- t4
- t5
- t6
- t7

### Instructions
- Instruction1
- Instruction2
- Instruction3
- Instruction4
- Instruction5

### Stages
- **IF**: I-Fetch
- **ID**: Decode, Register Fetch
- **EX**: Execute
- **MA**: Memory
- **WB**: Write-Back
5-Stage Pipelined Execution

Resource Usage Diagram

- **I-Fetch (IF)**
  - PC
  - Addr
  - IR
  - Inst. Memory

- **Decode, Reg. Fetch (ID)**
  - IF
  - Rs1
  - Rs2
  - Rd1
  - We
  - Ws
  - Wd
  - Ddata
  - GPRs

- **Execute (EX)**
  - ALU
  - Imm Ext
  - Addr
  - Rdata
  - Inst. Memory
  - Rd

- **Memory (MA)**
  - ADDR
  - RD
  - WDATA

- **Write-Back (WB)**

**Resources**

- **time**
  - t0
  - t1
  - t2
  - t3
  - t4
  - t5
  - t6
  - t7

- **IF**
  - I1
  - I2
  - I3
  - I4
  - I5

- **ID**
  - I1
  - I2
  - I3
  - I4
  - I5

- **EX**
  - I1
  - I2
  - I3
  - I4
  - I5

- **MA**
  - I1
  - I2
  - I3
  - I4
  - I5

- **WB**
  - I1
  - I2
  - I3
  - I4
  - I5

September 28, 2005
Pipelined Execution: ALU Instructions

Not quite correct!

We need an Instruction Reg (IR) for each stage
IR’s and Control points

Are control points connected properly?
- ALU instructions
- Load/Store instructions
- Write back
Pipelined MIPS Datapath
without jumps

September 28, 2005
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 produce data that is needed by a later instruction
  - *data hazard*

- In the extreme case, an instruction may determine the next instruction to be executed
  - *control hazard* (branches, interrupts, ...)

September 28, 2005
Data Hazards

...  
\[ r1 \leftarrow r0 + 10 \]
\[ r4 \leftarrow r1 + 17 \]
...

\[ r1 \text{ is stale. Oops!} \]
Resolving Data Hazards

*Freeze earlier pipeline stages* until the data becomes available ⇒ *interlocks*

If data is available somewhere in the datapath provide a *bypass* to get it to the right stage

*Speculate* about the hazard resolution and *kill* the instruction later if the speculation is wrong.
Feedback to Resolve Hazards

- Detect a hazard and provide feedback to previous stages to *stall or kill instructions*

- Controlling a pipeline in this manner works provided the instruction at stage $i+1$ can complete without any interference from instructions in stages 1 to $i$ (otherwise deadlocks may occur)
Interlocks to resolve Data Hazards

Stall Condition

...r1 ← r0 + 10
r4 ← r1 + 17
...

September 28, 2005
Stalled Stages and Pipeline Bubbles

\[
\begin{align*}
(I_1) & \quad r1 \leftarrow (r0) + 10 & \text{IF}_1 \\
(I_2) & \quad r4 \leftarrow (r1) + 17 & \text{IF}_2 \\
(I_3) & \quad & \\
(I_4) & \quad & \\
(I_5) & \quad & \\
\end{align*}
\]

\text{time} \quad t0 \quad t1 \quad t2 \quad t3 \quad t4 \quad t5 \quad t6 \quad t7 \ldots .

\begin{align*}
\text{IF} & \quad I_1 \quad I_2 \quad I_3 \quad I_3 \quad I_3 \quad I_4 \quad I_5 \\
\text{ID} & \quad I_1 \quad I_2 \quad I_2 \quad I_2 \quad I_2 \quad I_4 \quad I_5 \\
\text{EX} & \quad \text{nop} \quad \text{nop} \quad \text{nop} \quad \text{nop} \quad \text{nop} \quad \text{nop} \quad \text{nop} \\
\text{MA} & \quad I_1 \quad I_3 \quad I_3 \quad I_3 \quad I_4 \quad I_5 \quad I_5 \\
\text{WB} & \quad I_1 \quad I_3 \quad I_3 \quad I_3 \quad I_4 \quad I_5 \quad I_5 \\
\end{align*}

\text{stalled stages}

\text{nop} \Rightarrow \text{pipeline bubble}
Compare the source registers of the instruction in the decode stage with the destination register of the uncommitted instructions.

September 28, 2005
Should we always stall if the rs field matches some rd?

not every instruction writes a register ⇒ we
not every instruction reads a register ⇒ re
Source & Destination Registers

**R-type:**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>func</th>
</tr>
</thead>
</table>

**I-type:**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate16</th>
</tr>
</thead>
</table>

**J-type:**

<table>
<thead>
<tr>
<th>op</th>
<th>immediate26</th>
</tr>
</thead>
</table>

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

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

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

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

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

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

- **JAL**
  - \( r31 \leftarrow (PC), PC \leftarrow (PC) + \text{ imm} \)
    - Destination: \( 31 \)

- **JR**
  - \( PC \leftarrow (rs) \)
    - Source(s): \( rs \)

- **JALR**
  - \( r31 \leftarrow (PC), PC \leftarrow (rs) \)
    - Source(s): \( rs \)
    - Destination: \( 31 \)
Deriving the Stall Signal

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

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

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

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

\[ \begin{align*}
\text{re1} &= \text{Case opcode} \\
\text{ALU, ALUi, LW, SW, BZ, JR, JALR} &\Rightarrow \text{on} \\
\text{J, JAL} &\Rightarrow \text{off}
\end{align*} \]

\[ \begin{align*}
\text{re2} &= \text{Case opcode} \\
\text{ALU, SW} &\Rightarrow \text{on} \\
\text{...} &\Rightarrow \text{off}
\end{align*} \]

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

\[ \begin{align*}
stall &= ((rs_D = ws_E).we_E + \\
& (rs_D = ws_M).we_M + \\
& (rs_D = ws_W).we_W) \cdot \text{re1}_D + \\
& ((rt_D = ws_E).we_E + \\
& (rt_D = ws_M).we_M + \\
& (rt_D = ws_W).we_W) \cdot \text{re2}_D
\end{align*} \]

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

Stall Condition

Is there any possible data hazard in this instruction sequence?

M[(r1)+7] ← (r2)
r4 ← M[(r3)+5]

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

September 28, 2005
Load & Store Hazards

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

(r1)+7 = (r3)+5 ⇒ data hazard

However, the hazard is avoided because our memory system completes writes in one cycle!

Load/Store hazards, even when they do exist, are often resolved in the memory system itself.

More on this later in the course.
Five-minute break to stretch your legs
Complications due to Jumps

A jump instruction kills (not stalls) the following instruction

How?
Pipelining Jumps

To kill a fetched instruction -- Insert a mux before IR

Any interaction between stall and jump?

IRSrd = Case opcodesD
J, JAL ⇒ nop
... ⇒ IM

September 28, 2005
Jump Pipeline Diagrams

\textit{time}
\begin{align*}
\text{t0} & \quad \text{t1} & \quad \text{t2} & \quad \text{t3} & \quad \text{t4} & \quad \text{t5} & \quad \text{t6} & \quad \text{t7} & \quad \ldots \ldots \\
\text{IF}_1 & \quad \text{ID}_1 & \quad \text{EX}_1 & \quad \text{MA}_1 & \quad \text{WB}_1 & \quad & \quad & \quad & \quad \\
\text{IF}_2 & \quad \text{ID}_2 & \quad \text{EX}_2 & \quad \text{MA}_2 & \quad \text{WB}_2 & \quad & \quad & \quad & \quad \\
\text{IF}_3 & \quad \text{nop} & \quad \text{nop} & \quad \text{nop} & \quad \text{nop} & \quad & \quad & \quad & \quad \\
\text{IF}_4 & \quad \text{ID}_4 & \quad \text{EX}_4 & \quad \text{MA}_4 & \quad \text{WB}_4 & \quad & \quad & \quad & \quad \\
\end{align*}

\textbf{Resource Usage}
\begin{align*}
\text{IF} & \quad I_1 & \quad I_2 & \quad I_3 & \quad I_4 & \quad I_5 & \quad & \quad & \quad \\
\text{ID} & \quad I_1 & \quad I_2 & \quad & \quad & \quad & \quad & \quad & \quad \\
\text{EX} & \quad I_1 & \quad I_2 & \quad I_4 & \quad I_4 & \quad I_5 & \quad & \quad & \quad \\
\text{MA} & \quad I_1 & \quad I_2 & \quad I_4 & \quad I_4 & \quad I_5 & \quad & \quad & \quad \\
\text{WB} & \quad I_1 & \quad I_2 & \quad & \quad & \quad & \quad & \quad & \quad \\
\end{align*}

\textit{nop} \Rightarrow \textit{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
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

$\begin{array}{cccc}
I_1 & 096 & \text{ADD} \\
I_2 & 100 & \text{BEQZ} \ r1 \ 200 \\
I_3 & 104 & \text{ADD} \\
I_4 & 304 & \text{ADD} \\
\end{array}$
New Stall Signal

\[
\text{stall} = ( (\text{rs}_D = \text{ws}_E).\text{we}_E + (\text{rs}_D = \text{ws}_M).\text{we}_M + (\text{rs}_D = \text{ws}_W).\text{we}_W).\text{re}_1D \\
  + (\text{rt}_D = \text{ws}_E).\text{we}_E + (\text{rt}_D = \text{ws}_M).\text{we}_M + (\text{rt}_D = \text{ws}_W).\text{we}_W).\text{re}_2D \\
) . !((\text{opcode}_E = \text{BEQZ}).z + (\text{opcode}_E = \text{BNEZ}).!z)
\]

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

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

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

\[\text{IRSrc}_D = \begin{cases} \text{Case opcode}_E \\ \quad \text{BEQZ.z, BNEZ.!z} \Rightarrow \text{nop} \\ \quad \ldots \Rightarrow \quad \text{Case opcode}_D \\ \quad \quad \quad \text{J, JAL, JR, JALR} \Rightarrow \text{nop} \\ \quad \ldots \quad \Rightarrow \quad \text{IM} \end{cases}\]

\[\text{IRSrc}_E = \begin{cases} \text{Case opcode}_E \\ \quad \text{BEQZ.z, BNEZ.!z} \Rightarrow \text{nop} \\ \quad \ldots \Rightarrow \quad \text{stall}.\text{nop} + \text{!stall}.\text{IR}_D \end{cases}\]

Give priority to the older instruction, i.e., execute stage instruction over decode stage instruction.
Branch Pipeline Diagrams
(resolved in execute stage)

\[\begin{array}{cccccccc}
\text{time} & t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 \\
\hline
(I_1) 096: ADD & IF_1 & ID_1 & EX_1 & MA_1 & WB_1 & \\
(I_2) 100: BEQZ 200 & IF_2 & ID_2 & EX_2 & MA_2 & WB_2 & \\
(I_3) 104: ADD & IF_3 & ID_3 & & & & \\
(I_4) 108: & & & & & \\
(I_5) 304: ADD & IF_5 & ID_5 & EX_5 & MA_5 & WB_5 & \\
\end{array}\]

Resource Usage

\[\begin{array}{cccccccc}
\text{time} & t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 \\
\hline
\text{IF} & I_1 & I_2 & I_3 & I_4 & I_5 & \\
\text{ID} & I_1 & I_2 & I_3 & \text{nop} & I_5 & \\
\text{EX} & I_1 & I_2 & \text{nop} & \text{nop} & I_5 & \\
\text{MA} & & I_1 & I_2 & \text{nop} & \text{nop} & I_5 & \\
\text{WB} & & I_1 & I_2 & \text{nop} & \text{nop} & I_5 & \text{nop} & I_5 & \\
\end{array}\]

\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

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.

<table>
<thead>
<tr>
<th>I₁</th>
<th>096</th>
<th>ADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>I₂</td>
<td>100</td>
<td>BEQZ r1 200</td>
</tr>
<tr>
<td>I₃</td>
<td>104</td>
<td>ADD</td>
</tr>
<tr>
<td>I₄</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
Bypassing

Each *stall or kill* introduces a bubble in the pipeline

⇒ *CPI* > 1

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

JAL 500  \( r4 \leftarrow r31 + 17 \)  no

September 28, 2005
The Bypass Signal
Deriving it from the Stall Signal

\[
\text{stall} = ((\text{rs}_D = \text{ws}_E).\text{we}_E + (\text{rs}_D = \text{ws}_M).\text{we}_M + (\text{rs}_D = \text{ws}_W).\text{we}_W).\text{re}_1^D \\
+((\text{rt}_D = \text{ws}_E).\text{we}_E + (\text{rt}_D = \text{ws}_M).\text{we}_M + (\text{rt}_D = \text{ws}_W).\text{we}_W).\text{re}_2^D)
\]

**ws = Case opcode**
- ALU \(\Rightarrow rd\)
- ALUi, LW \(\Rightarrow rt\)
- JAL, JALR \(\Rightarrow R31\)

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

\[
\text{ASrc} = (\text{rs}_D = \text{ws}_E).\text{we}_E.\text{re}_1^D
\]

Is this correct?

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

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

Split \( we_E \) into two components: \( we\text{-bypass} \), \( we\text{-stall} \)

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

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

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

\[\text{stall} = ((rs_D = ws_E) \cdot \text{we-stall}_E + \]

\[\quad (rs_D = ws_M) \cdot \text{we}_M + (rs_D = ws_W) \cdot \text{we}_W) \cdot \text{re}_1_D\]

\[\quad + ((rt_D = ws_E) \cdot \text{we}_E + (rt_D = ws_M) \cdot \text{we}_M + (rt_D = ws_W) \cdot \text{we}_W) \cdot \text{re}_2_D\]
Fully Bypassed Datapath

Is there still a need for the stall signal?

\[
\text{stall} = (\text{rs}_D = \text{ws}_E). (\text{opcode}_E = \text{LW}_E). (\text{ws}_E \neq 0). \text{re1}_D \\
+ (\text{rt}_D = \text{ws}_E). (\text{opcode}_E = \text{LW}_E). (\text{ws}_E \neq 0). \text{re2}_D
\]
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 may cause bubbles
  - kill following instruction(s) if no delay slots

*Machines with software-visible delay slots may execute significant number of NOP instructions inserted by the compiler.*
Thank you!