Instruction Pipelining: Hazard Resolution, Timing Constraints

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory M.I.T.
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

\[
\begin{align*}
r1 &\leftarrow r0 + 10 \\
r4 &\leftarrow r1 + 17 \\
\end{align*}
\]
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
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

<table>
<thead>
<tr>
<th>R-type:</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-type:</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate16</td>
<td></td>
</tr>
<tr>
<td>J-type:</td>
<td>op</td>
<td>immediate26</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

source(s) destination

ALU  rd ← (rs) func (rt)     rs, rt    rd
ALUi rt ← (rs) op imm        rs        rt
LW   rt ← M [(rs) + imm]     rs        rt
SW   M [(rs) + imm] ← (rt)   rs, rt
BZ   cond (rs)
     true: PC ← (PC) + imm    rs
     false: PC ← (PC) + 4     rs
J    PC ← (PC) + imm
JAL  r31 ← (PC), PC ← (PC) + imm  31
JR   PC ← (rs)               rs
JALR r31 ← (PC), PC ← (rs)   rs    31
### Deriving the Stall Signal

#### $C_{\text{dest}}$

<table>
<thead>
<tr>
<th>Case</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ALU</code></td>
<td><code>rd</code></td>
</tr>
<tr>
<td><code>ALUi</code>, <code>LW</code></td>
<td><code>rt</code></td>
</tr>
<tr>
<td><code>JAL</code>, <code>JALR</code></td>
<td><code>R31</code></td>
</tr>
</tbody>
</table>

#### $w_e = Case$ opcode

- `ALU`, `ALUi`, `LW` $\Rightarrow (w_e \neq 0)$
- `JAL`, `JALR` $\Rightarrow$ on
- `...` $\Rightarrow$ off

#### $C_{\text{re}}$

<table>
<thead>
<tr>
<th>Case</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ALU</code>, <code>ALUi</code>, <code>LW</code>, <code>SW</code>, <code>BZ</code>, <code>JR</code>, <code>JALR</code></td>
<td>$\Rightarrow$ on</td>
</tr>
<tr>
<td><code>J</code>, <code>JAL</code></td>
<td>$\Rightarrow$ off</td>
</tr>
</tbody>
</table>

#### $re1 = Case$ opcode

```
$re1 = Case$ opcode
   `ALU, ALUi, LW` $\Rightarrow$ on
   `JAL, JALR` $\Rightarrow$ off
```

#### $re2 = Case$ opcode

```
$re2 = Case$ opcode
   `ALU, SW` $\Rightarrow$ on
```

#### $C_{\text{stall}}$

\[
\text{stall} = ((r_{D} = = ws_{E}) \cdot w_{E} + \\
(r_{D} = = ws_{M}) \cdot w_{M} + \\
(r_{D} = = ws_{W}) \cdot w_{W}) \cdot re_{1_{D}} + \\
((r_{T} = = ws_{E}) \cdot w_{E} + \\
(r_{T} = = ws_{M}) \cdot w_{M} + \\
(r_{T} = = ws_{W}) \cdot w_{W}) \cdot re_{2_{D}}
\]
... $M[(r1)+7] \leftarrow (r2)$

$r4 \leftarrow M[(r3)+5]$

... Is there any possible data hazard in this instruction sequence?
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?

(I_1) \( r_1 \leftarrow r_0 + 10 \)

(I_2) \( r_4 \leftarrow r_1 + 17 \)

yes

JAL 500

JAL 500

no
The Bypass Signal
Deriving it from the Stall Signal

\[
\text{stall} = ((rs_D = ws_E) \cdot we_E + (rs_D = ws_M) \cdot we_M + (rs_D = ws_W) \cdot we_W) \cdot \text{re1}_D
\]
\[
+ ((rt_D = ws_E) \cdot we_E + (rt_D = ws_M) \cdot we_M + (rt_D = ws_W) \cdot we_W) \cdot \text{re2}_D
\]

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

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

\[
\text{ASrc} = (rs_D = ws_E) \cdot we_E \cdot \text{re1}_D
\]

Is this correct?

How might we address this?
Bypass and Stall Signals

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

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

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

\[
\text{ASrc} = (\text{rs}_D == \text{ws}_E) \cdot \text{we-bypass}_E \cdot \text{re}_1_D
\]

\[
\text{stall} = ((\text{rs}_D == \text{ws}_E) \cdot \text{we-stall}_E + \\
(\text{rs}_D == \text{ws}_M) \cdot \text{we}_M + (\text{rs}_D == \text{ws}_W) \cdot \text{we}_W) \cdot \text{re}_1_D \\
+((\text{rt}_D == \text{ws}_E) \cdot \text{we}_E + (\text{rt}_D == \text{ws}_M) \cdot \text{we}_M + (\text{rt}_D == \text{ws}_W) \cdot \text{we}_W) \cdot \text{re}_2_D
\]
Is there still a need for the stall signal?
Resolving Data Hazards (3)

**Strategy 3:**

*Speculate on the dependence. Two cases:*

*Guessed correctly* → no special action required

*Guessed incorrectly* → kill and restart
Instruction to Instruction Dependence

• What do we need to calculate next PC?
  – For Jumps
  – For Jump Register
  – For Conditional Branches
  – For all others

• In what stage do we know these?
NextPC Calculation Bubbles

What’s a good guess for next PC?
Speculate NextPC is PC+4

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

I₀  096  ADD
I₁  100  J  200
I₂  104  ADD  **kill**
I₃  304  ADD

How?
Pipelining Jumps

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

Any interaction between stall and jump?

IRSr_{D} = Case opcode_{D}
J, JAL \Rightarrow \text{nop}
...
\Rightarrow \text{IM}

I_1 \quad 096 \quad \text{ADD}
I_2 \quad 100 \quad \text{J} \quad 200
I_3 \quad 104 \quad \text{ADD}
I_4 \quad 304 \quad \text{ADD}
Jump Pipeline Diagrams

(time)

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

IF \( I_1 \) 096: ADD
ID \( I_2 \) 100: J 200
EX \( I_3 \) 104: ADD
MA \( I_4 \) 304: ADD
WB

Resource Usage

\( I_1 \) \( I_2 \) \( I_3 \) \( I_4 \) \( I_5 \)
IF \nID \nEX \nMA \nWB

\( \text{nop} \Rightarrow \text{pipeline bubble} \)
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
If the branch is taken
- kill the two following instructions
- the instruction at the decode stage is not valid
⇒ \textit{stall signal is not valid}
New Stall Signal

\[
stall = ( ((rs_D==ws_E) \cdot we_E + (rs_D==ws_M) \cdot we_M + (rs_D==ws_W) \cdot we_W) \cdot re_1D \\
+ ((rt_D==ws_E) \cdot we_E + (rt_D==ws_M) \cdot we_M + (rt_D==ws_W) \cdot we_W) \cdot re_2D \\
) \cdot !((\text{opcode}_E==\text{BEQZ}) \cdot z + (\text{opcode}_E==\text{BNEZ}) \cdot !z)
\]

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

**IRSrc**

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

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

**IRSrc**

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

**PCSrc**

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

\[
\text{nop} \quad \Rightarrow \text{Kill} \\
\text{br/jabs/rind} \Rightarrow \text{Restart} \\
\text{pc+4} \quad \Rightarrow \text{Speculate}
\]

pc+4 is a speculative guess
Branch Pipeline Diagrams
(resolved in execute stage)

(time
t0  t1  t2  t3  t4  t5  t6  t7  . . . .

(I_1) 096: ADD
(I_2) 100: BEQZ 200
(I_3) 104: ADD
(I_4) 108: ADD
(I_5) 304: ADD

Resource Usage

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

| I₁  | 096  | ADD   |
| I₂  | 100  | BEQZ  r1 200 |
| I₃  | 104  | ADD   |
| I₄  | 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