Instruction Pipelining and Hazards

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory M.I.T.
Reminder: Harvard-Style Single-Cycle Datapath for MIPS
Reminder: Princeton Microarchitecture

Datapath & Control for 2 cycles-per-instruction

March 4, 2021

MIT 6.823 Spring 2021
Princeton Microarchitecture
*(redrawn)*

Only one of the phases is active in any cycle
⇒ a lot of datapath not used at any given time
Princeton Microarchitecture
Overlapped execution

Can we overlap instruction fetch and execute?

Which action should be prioritized?

What do we do with Fetch?
When stall condition is indicated
- *don’t fetch a new instruction and don’t change the PC*
- *insert a nop in the IR*
- *set the Memory Address mux to ALU (not shown)*

*What if IR contains a jump or branch instruction?*
**Need to stall on branches**

*Princeton Microarchitecture*

When IR contains a jump or taken branch:

- *no “structural conflict” for the memory*
- *but we do not have the correct PC value in the PC*
- *memory cannot be used – Address Mux setting is irrelevant*
- *insert a nop in the IR*
- *insert the nextPC (branch-target) address in the PC*
Pipelined Princeton Microarchitecture
## Pipelined Princeton: Control Table

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Stall</th>
<th>Ext Sel</th>
<th>B Src</th>
<th>Op Sel</th>
<th>Mem W</th>
<th>Reg W</th>
<th>WB Src</th>
<th>Reg Dst</th>
<th>PC Src1</th>
<th>PC Src2</th>
<th>IR Src</th>
<th>MAaddr Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>no</td>
<td>*</td>
<td>Reg</td>
<td>Func</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rd</td>
<td>pc+4</td>
<td>npc</td>
<td>mem</td>
<td>pc</td>
</tr>
<tr>
<td>ALUi</td>
<td>no</td>
<td>sE&lt;sub&gt;16&lt;/sub&gt;</td>
<td>Imm</td>
<td>Op</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rt</td>
<td>pc+4</td>
<td>npc</td>
<td>mem</td>
<td>pc</td>
</tr>
<tr>
<td>ALUiu</td>
<td>no</td>
<td>uE&lt;sub&gt;16&lt;/sub&gt;</td>
<td>Imm</td>
<td>Op</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rt</td>
<td>pc+4</td>
<td>npc</td>
<td>mem</td>
<td>pc</td>
</tr>
<tr>
<td>LW</td>
<td>yes</td>
<td>sE&lt;sub&gt;16&lt;/sub&gt;</td>
<td>Imm</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>Mem</td>
<td>rt</td>
<td>pc+4</td>
<td>pc</td>
<td>nop</td>
<td>ALU</td>
</tr>
<tr>
<td>SW</td>
<td>yes</td>
<td>sE&lt;sub&gt;16&lt;/sub&gt;</td>
<td>Imm</td>
<td>*</td>
<td>yes</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
<td>pc</td>
<td>nop</td>
<td>ALU</td>
</tr>
<tr>
<td>BEQZ&lt;sub&gt;z=1&lt;/sub&gt;</td>
<td>yes</td>
<td>sE&lt;sub&gt;16&lt;/sub&gt;</td>
<td>*</td>
<td>0?</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>br</td>
<td>npc</td>
<td>nop</td>
<td>*</td>
</tr>
<tr>
<td>BEQZ&lt;sub&gt;z=0&lt;/sub&gt;</td>
<td>no</td>
<td>sE&lt;sub&gt;16&lt;/sub&gt;</td>
<td>*</td>
<td>0?</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
<td>npc</td>
<td>mem</td>
<td>pc</td>
</tr>
<tr>
<td>J</td>
<td>yes</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>jabs</td>
<td>npc</td>
<td>nop</td>
<td>*</td>
</tr>
<tr>
<td>JAL</td>
<td>yes</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>PC</td>
<td>R31</td>
<td>jabs</td>
<td>npc</td>
<td>nop</td>
<td>*</td>
</tr>
<tr>
<td>JR</td>
<td>yes</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>rind</td>
<td>npc</td>
<td>nop</td>
<td>*</td>
</tr>
<tr>
<td>JALR</td>
<td>yes</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>PC</td>
<td>R31</td>
<td>rind</td>
<td>npc</td>
<td>nop</td>
<td>*</td>
</tr>
<tr>
<td>NOP</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
<td>npc</td>
<td>mem</td>
<td>pc</td>
</tr>
</tbody>
</table>

BSrc = Reg / Imm ; WBSrc = ALU / Mem / PC; IRSrc = nop/mem; MAaddrSrc = pc/ALU
RegDst = rt / rd / R31; PCSrc1 = pc+4 / br / rind / jabs; PCSrc2 = pc/nPC

* stall & IRSrc columns are identical
Pipelined Princeton Architecture

Clock: \( t_{C-Princeton} > t_{RF} + t_{ALU} + t_{M} + t_{WB} \)

\( CPI: \) \( (1 - f) + 2f \) cycles per instruction
where \( f \) is the fraction of instructions that cause a stall

What is a likely value of \( f \)?
An Ideal Pipeline

- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

These conditions generally hold for industrial assembly lines.

But what about an instruction pipeline?
Clock period can be reduced by dividing the execution of an instruction into multiple cycles

\[ t_C > \max \{ t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW} \} \ (= t_{DM} \text{ probably}) \]

However, CPI will increase unless instructions are pipelined
Suppose memory is significantly slower than other stages. For example, suppose

\[
\begin{align*}
    t_{IM} &= 10 \text{ units} \\
    t_{DM} &= 10 \text{ units} \\
    t_{ALU} &= 5 \text{ units} \\
    t_{RF} &= 1 \text{ unit} \\
    t_{RW} &= 1 \text{ unit}
\end{align*}
\]

Since the slowest stage determines the clock, it may be possible to combine some stages without any loss of performance.
Write-back stage takes much less time than other stages. Suppose we combined it with the memory phase.
# Maximum Speedup by Pipelining

<table>
<thead>
<tr>
<th>Assumptions</th>
<th>Unpipelined $t_C$</th>
<th>Pipelined $t_C$</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $t_{IM} = t_{DM} = 10,$ $t_{ALU} = 5,$ $t_{RF} = t_{RW} = 1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-stage pipeline</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-stage pipeline</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5-stage pipeline</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What seems to be the message here?
5-Stage Pipelined Execution

Instruction Flow Diagram

I-Fetch (IF)

<table>
<thead>
<tr>
<th>time</th>
<th>instruction1</th>
<th>instruction2</th>
<th>instruction3</th>
<th>instruction4</th>
<th>instruction5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF&lt;sub&gt;1&lt;/sub&gt;</td>
<td>IF&lt;sub&gt;2&lt;/sub&gt;</td>
<td>IF&lt;sub&gt;3&lt;/sub&gt;</td>
<td>IF&lt;sub&gt;4&lt;/sub&gt;</td>
<td>IF&lt;sub&gt;5&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Decode, Reg. Fetch (ID)

Execute (EX)

Memory (MA)

Write-Back (WB)

Write Back (WB)
5-Stage Pipelined Execution

Resource Usage Diagram

I-Fetch (IF)
Decode, Reg. Fetch (ID)
Execute (EX)
Memory (MA)
Write-Back (WB)

Resources
- IF
- ID
- EX
- MA
- WB

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

Resources:
- PC
- Addr
- rdata
- IR
- Inst. Memory
- addr
- rdata
- IR
- Decode, Reg. Fetch (ID)
- Execute (EX)
- Memory (MA)
- Write-Back (WB)
- 0x4 Add
- ALU
- Imm Ext
- Memory
- Data
- IR
- PC
- R1
- R2
- R3
- R4
- R5
- R6
- R7
- R8
- R9
- R10
- R11
- R12
- R13
- R14
- R15
Pipelined Execution

ALU Instructions

Not quite correct!
Pipelined MIPS Datapath
without jumps

What else is needed?
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

... 

r1 ← r0 + 10
r4 ← r1 + 17
...

r1 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* → do nothing
  - *Guessed incorrectly* → kill and restart
Resolving Data Hazards (1)

Strategy 1:

Wait for the result to be available by freezing earlier pipeline stages → stall
Feedback to Resolve Hazards

Later stages provide dependence information to earlier stages which can 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).
Resolving Data Hazards by Stalling

Stall Condition

... r1 ← r0 + 10
r4 ← r1 + 17
...

March 4, 2021
Stalled Stages and Pipeline Bubbles

\[ \text{time} \]
\[
\begin{array}{cccccccc}
\text{t0} & \text{t1} & \text{t2} & \text{t3} & \text{t4} & \text{t5} & \text{t6} & \text{t7} & \ldots \\
\text{IF} & \text{IF} & \text{EX} & \text{MA} & \text{WB} & \text{IF} & \text{EX} & \text{MA} & \text{WB} \\
\text{ID} & \text{ID} & \text{ID} & \text{ID} & \text{ID} & \text{ID} & \text{EX} & \text{MA} & \text{WB} \\
\text{IF} & \text{IF} & \text{IF} & \text{IF} & \text{IF} & \text{IF} & \text{IF} & \text{IF} & \text{IF} \\
\end{array}
\]

(stalled stages)

\[(I_1) \ r1 \leftarrow (r0) + 10\]
\[(I_2) \ r4 \leftarrow (r1) + 17\]
\[(I_3)\]
\[(I_4)\]
\[(I_5)\]

Resource Usage

\[
\begin{array}{cccccccc}
\text{IF} & \text{ID} & \text{EX} & \text{MA} & \text{WB} \\
\text{I}_1 & \text{I}_1 & \text{I}_1 & \text{I}_1 & \text{I}_1 \\
\text{I}_2 & \text{I}_2 & \text{I}_2 & \text{I}_2 & \text{I}_2 \\
\text{I}_3 & \text{I}_3 & \text{I}_3 & \text{I}_3 & \text{I}_3 \\
\text{I}_4 & \text{I}_4 & \text{I}_4 & \text{I}_4 & \text{I}_4 \\
\text{I}_5 & \text{I}_5 & \text{I}_5 & \text{I}_5 & \text{I}_5 \\
\end{array}
\]

\[
\text{nop} \Rightarrow \text{pipeline bubble}
\]
Compare the source registers of the instruction in the decode stage with the destination register of the uncommitted instructions.
Should we always stall if the rs field matches some rd?
Source & Destination Registers

\[ \text{R-type:} \quad \begin{array}{cccc} \text{op} & \text{rs} & \text{rt} & \text{rd} \\ \text{func} & \end{array} \]

\[ \text{I-type:} \quad \begin{array}{cccc} \text{op} & \text{rs} & \text{rt} & \text{immediate16} \end{array} \]

\[ \text{J-type:} \quad \begin{array}{cccc} \text{op} \phantom{immediate26} & \text{immediate26} \end{array} \]

source(s) destination

\begin{align*}
\text{ALU} & \; \quad \text{rd} \leftarrow (\text{rs}) \text{ func (rt)} & \text{rs, rt} & \text{rd} \\
\text{ALUi} & \; \quad \text{rt} \leftarrow (\text{rs}) \text{ op imm} & \text{rs} & \text{rt} \\
\text{LW} & \; \quad \text{rt} \leftarrow M [(\text{rs}) + \text{imm}] & \text{rs} & \text{rt} \\
\text{SW} & \; \quad M [(\text{rs}) + \text{imm}] \leftarrow (\text{rt}) & \text{rs, rt} & \text{rt} \\
\text{BZ} & \; \begin{array}{c} \text{cond (rs)} \\ \text{true:} \quad \text{PC} \leftarrow (\text{PC}) + \text{imm} \\ \text{false:} \quad \text{PC} \leftarrow (\text{PC}) + 4 \\ \end{array} & \text{rs} & \text{rs} \\
\text{J} & \; \quad \text{PC} \leftarrow (\text{PC}) + \text{imm} & \text{rs} & \text{rt} \\
\text{JAL} & \; \quad r31 \leftarrow (\text{PC}), \text{PC} \leftarrow (\text{PC}) + \text{imm} & \text{rs} & \text{rt} \\
\text{JR} & \; \quad \text{PC} \leftarrow (\text{rs}) & \text{rs} & \text{rt} \\
\text{JALR} & \; \quad r31 \leftarrow (\text{PC}), \text{PC} \leftarrow (\text{rs}) & \text{rs} & \text{rt} \\
\end{align*}
Deriving the Stall Signal

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

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

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

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

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

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

Stall Condition

... M[(r1)+7] ← (r2)  
r4 ← M[(r3)+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.

*(r1)+7 = (r3)+5 \Rightarrow \text{data hazard}*

*More on this later in the course.*
Next lecture: Control Hazards, Bypassing, and Speculation