Instruction Pipelining

Hyun Ryong (Ryan) Lee
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Announcements

• Lab 1 released later today
  – Designing 3 different cache models using Pin
  – Due Sept. 29

• Please view the Pin tutorial video posted on Piazza

• Contact Nikola (in charge of labs) if you cannot get access to lab machines
Reminder: Harvard-Style Single-Cycle Datapath for RISC-V
We will assume
• Clock period is sufficiently long for all of the following steps to be “completed”:

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. data fetch if required
5. register write-back setup time

\[ t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB} \]

• At the rising edge of the following clock, the PC, the register file and the memory are updated
Datapath for Memory Instructions

Should program and data memory be separate?

*Harvard style: separate* (Aiken and Mark 1 influence)
- read-only program memory
- read/write data memory

- Note:
  There must be a way to load the program memory

*Princeton style: the same* (von Neumann’s influence)
- single read/write memory for program and data

- Note:
  Executing a Load or Store instruction requires accessing the memory more than once
Princeton challenge

• What problem arises if instructions and data reside in the same memory?
Princeton Microarchitecture
Datapath & Control

Fetch phase

PCen

IR

clk

on

IRen

0x4

Add

AddrSrc = PC

data

AddrSrc

MemWrite

WBSrc

MemWrite

off

RegWrite

off

ALU

ALUFunc

BSrc

ALUFunct

AddrSrc = PC

IRen

IR

clk

IRen

IR

irr

AddrSrc

Fetch phase
Two-State Controller: Princeton Architecture

fetch phase

execute phase

A flipflop can be used to remember the phase
Hardwired Controller:
Princeton Architecture

old combinational logic (Harvard)

new combinational logic

1-bit Toggle FF
I-fetch / Execute
Is it possible to design a controller for the Princeton architecture with CPI < 2?

\[ t_{C-Princeton} > \max \{ t_M, t_{RF} + t_{ALU} + t_M + t_{WB} \} \]
\[ t_{C-Princeton} > t_{RF} + t_{ALU} + t_M + t_{WB} \]
\[ t_{C-Harvard} > t_M + t_{RF} + t_{ALU} + t_M + t_{WB} \]

Suppose \( t_M >> t_{RF} + t_{ALU} + t_{WB} \)

\[ t_{C-Princeton} = 0.5 \times t_{C-Harvard} \]

\( CPI_{Princeton} = 2 \)
\( CPI_{Harvard} = 1 \)

No difference in performance!

\[ CPI = \text{Clock cycles Per Instruction} \]
Princeton Microarchitecture
(redrawn)

Only one of the phases is active in any cycle
⇒ a lot of datapath not used at any given time
Can we overlap instruction fetch and execute?

Yes, unless IR contains a Load or Store

Which action should be prioritized?  
Execute

What do we do with Fetch?  
Stall it

How?
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
# Hardwired Control Table

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Stall</th>
<th>Imm Sel</th>
<th>BSrc</th>
<th>ALU</th>
<th>MemW</th>
<th>WBSrc</th>
<th>RegWr</th>
<th>PC Src1</th>
<th>PC Src2</th>
<th>IRSrc</th>
<th>MAddr Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>no</td>
<td>*</td>
<td>Reg</td>
<td>funct</td>
<td>no</td>
<td>ALU</td>
<td>yes</td>
<td>pc+4</td>
<td>npc</td>
<td>mem</td>
<td>pc</td>
</tr>
<tr>
<td>ALUi</td>
<td>no</td>
<td>SXT(imm [11:0])</td>
<td>Imm</td>
<td>funct</td>
<td>no</td>
<td>ALU</td>
<td>yes</td>
<td>pc+4</td>
<td>npc</td>
<td>mem</td>
<td>pc</td>
</tr>
<tr>
<td>LW</td>
<td>yes</td>
<td>SXT(imm [11:0])</td>
<td>Imm</td>
<td>+</td>
<td>no</td>
<td>Mem</td>
<td>yes</td>
<td>pc+4</td>
<td>pc</td>
<td>nop</td>
<td>ALU</td>
</tr>
<tr>
<td>SW</td>
<td>yes</td>
<td>SXT(imm [11:0])</td>
<td>Imm</td>
<td>+</td>
<td>yes</td>
<td>*</td>
<td>no</td>
<td>pc+4</td>
<td>pc</td>
<td>nop</td>
<td>ALU</td>
</tr>
<tr>
<td>BRtaken</td>
<td>yes</td>
<td>ImmB</td>
<td>Reg</td>
<td>funct</td>
<td>no</td>
<td>*</td>
<td>no</td>
<td>br</td>
<td>npc</td>
<td>nop</td>
<td>*</td>
</tr>
<tr>
<td>BRnotTaken</td>
<td>no</td>
<td>ImmB</td>
<td>Reg</td>
<td>funct</td>
<td>no</td>
<td>*</td>
<td>no</td>
<td>pc+4</td>
<td>npc</td>
<td>mem</td>
<td>pc</td>
</tr>
<tr>
<td>JALR</td>
<td>yes</td>
<td>ImmI</td>
<td>Imm</td>
<td>+</td>
<td>no</td>
<td>pc+4</td>
<td>yes</td>
<td>jt</td>
<td>npc</td>
<td>nop</td>
<td>*</td>
</tr>
<tr>
<td>JAL</td>
<td>yes</td>
<td>ImmU</td>
<td>Imm</td>
<td>*</td>
<td>no</td>
<td>pc+4</td>
<td>yes</td>
<td>br</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>*</td>
<td>no</td>
<td>pc+4</td>
<td>npc</td>
<td>mem</td>
<td>pc</td>
</tr>
</tbody>
</table>

BSrc = Reg / Imm; IRSrc = nop/mem; MAddrSrc = pc/ALU; WBSrc = ALU / Mem / pc+4
PCSrc1 = pc+4 / br / rind; PCSrc2 = npc/pc;
Pipelined Princeton Architecture

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

**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.
How to divide datapath into stages

Suppose memory is significantly slower than other stages. For example, suppose

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

Since the slowest stage determines the clock, it may be possible to combine some stages without any loss of performance.
Alternative Pipelining

Write-back stage takes much less time than other stages. Suppose we combined it with the memory phase.

\[ t_C > \max \{t_{IM}, t_{RF} + t_{ALU}, t_{DM} + t_{RW}\} = t_{DM} + t_{RW} \]

⇒ increase the critical path by 10%
# Maximum Speedup by Pipelining

## Assumptions

<table>
<thead>
<tr>
<th></th>
<th>Unpipelined $t_C$</th>
<th>Pipelined $t_C$</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$t_{IM} = t_{DM} = 10$, $t_{ALU} = 5$, $t_{RF} = t_{RW} = 1$</td>
<td>27</td>
<td>10</td>
</tr>
<tr>
<td>2.</td>
<td>$t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>3.</td>
<td>$t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$</td>
<td>25</td>
<td>5</td>
</tr>
</tbody>
</table>

### What seems to be the message here?

*One can achieve higher speedup with more pipeline stages*
5-Stage Pipelined Execution

Instruction Flow Diagram

I-Fetch (IF)

Decode, Reg. Fetch (ID)

Execute (EX)

Memory (MA)

Write-Back (WB)

[Diagram showing the pipeline stages with time stamps and instructions]

t0 t1 t2 t3 t4 t5 t6 t7 . . .

0x4 Add

PC

addr p

Inst. Memory

IR

rdata

Dec. Reg. Fetch (ID)

Execute (EX)

Memory (MA)

Write-Back (WB)

[Box labeled with various stages and operations: IF, ID, EX, MA, WB]
5-Stage Pipelined Execution

Resource Usage Diagram

<table>
<thead>
<tr>
<th>Resources</th>
<th>time</th>
<th>Resources</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>t0 I_1</td>
<td>IF</td>
<td>t1 I_2</td>
</tr>
<tr>
<td>ID</td>
<td>t2 I_3</td>
<td>ID</td>
<td>t3 I_4</td>
</tr>
<tr>
<td>EX</td>
<td>t4 I_5</td>
<td>EX</td>
<td>t5 I_6</td>
</tr>
<tr>
<td>MA</td>
<td>t6 I_7</td>
<td>MA</td>
<td>t7 I_8</td>
</tr>
<tr>
<td>WB</td>
<td>... I_9</td>
<td>WB</td>
<td>... I_10</td>
</tr>
</tbody>
</table>

September 15, 2023
Pipelined Execution

ALU Instructions

Not quite correct!

We need an Instruction Reg (IR) for each stage
Pipelined RISC-V Datapath

without jumps

What else is needed?

Control Points Need to Be Connected
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

*Use strategy from Princeton Pipeline:*

*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 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
...

September 15, 2023
Stalled Stages and Pipeline Bubbles

\[ (I_1) \text{ } r_1 \leftarrow (r_0) + 10 \]
\[ (I_2) \text{ } r_4 \leftarrow (r_1) + 17 \]

\[ \text{Resource Usage} \]

\[ \text{time} \]
\[ t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \quad t_7 \quad \ldots \]

\[ \text{IF} \quad I_1 \quad I_2 \quad I_3 \quad I_3 \quad I_3 \quad I_4 \quad I_5 \]
\[ \text{ID} \quad I_1 \quad I_2 \quad I_2 \quad I_2 \quad I_3 \quad I_4 \quad I_5 \]
\[ \text{EX} \quad I_1 \quad \text{nop} \quad \text{nop} \quad \text{nop} \quad I_2 \quad I_3 \quad I_4 \quad I_5 \]
\[ \text{MA} \quad I_1 \quad \text{nop} \quad \text{nop} \quad \text{nop} \quad I_2 \quad I_3 \quad I_4 \quad I_5 \]
\[ \text{WB} \quad I_1 \quad \text{nop} \quad \text{nop} \quad \text{nop} \quad I_2 \quad I_3 \quad I_4 \quad I_5 \]

\[ \text{pipeline bubble} \]
Compare the *source registers* of the instruction in the decode stage with the *destination register* of the *uncommitted instructions*.
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
not every instruction reads a register ⇒ re
Thank you!