Chapter 4.5 Overview of Pipelining

Chansu Yu

Cleveland State University

Laundry Example

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 30 minutes
- "Folder" takes 30 minutes
- "Stasher" takes 30 minutes to put clothes into drawers
Sequential Laundry

Sequential laundry takes 8 hours for 4 loads
If they learned pipelining, how long would laundry take?

Faster Laundry - Pipelining

Faster laundry takes 3.5 hours for 4 loads!
Breaking down an instruction

- ISA definition of arithmetic:
  \[
  \text{Reg}[\text{Memory}[\text{PC}][15:11]] \leq \text{Reg}[\text{Memory}[\text{PC}][25:21]] \circ \text{op} \leq \text{Reg}[\text{Memory}[\text{PC}][20:16]]
  \]

- Could break down to:
  - \( \text{IR} \leq \text{Memory}[\text{PC}] \)
  - \( \text{A} \leq \text{Reg}[\text{IR}[25:21]] \)
  - \( \text{B} \leq \text{Reg}[\text{IR}[20:16]] \)
  - \( \text{ALUOut} \leq \text{A} \circ \text{op} \text{B} \)
  - \( \text{Reg}[\text{IR}[20:16]] \leq \text{ALUOut} \)

- We forgot an important part of the definition of arithmetic!
  - \( \text{PC} \leq \text{PC} + 4 \)

Comparison

- Read registers \((s0, s2)\)
- ALU operation \((\text{add})\)
- Write register \((s0)\)

- Read registers \((s0, s2)\)
- ALU operation \((\text{add})\)
- Write register \((s0)\)
Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Memory-read completion step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

\[
\begin{align*}
\text{IR} &= \text{Memory[PC]}; \\
\text{PC} &= \text{PC} + 4;
\end{align*}
\]

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?
- “ALU will be busy calculating something else in other cycles”
Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:
  
  \[
  A = \text{Reg}[\text{IR}[25-21]]; \\
  B = \text{Reg}[\text{IR}[20-16]]; \\
  \text{ALUOut} = \text{PC} + (\text{sign-extend}(\text{IR}[15-0]) \ll 2);
  \]

- We aren’t setting any control lines based on the instruction type
  (we are busy “decoding” it in our control logic)

Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type
- Memory Reference:
  
  \[
  \text{ALUOut} = A + \text{sign-extend}(\text{IR}[15-0]);
  \]

- R-type:
  
  \[
  \text{ALUOut} = A \text{ op } B;
  \]

- Branch:
  
  \[
  \text{if } (A==B) \text{ PC }= \text{ALUOut};
  \]
Step 4: R-type Completion or Memory Access

- Loads and stores access memory
  
  \[ \text{MDR} = \text{Memory[ALUOut]}; \]
  
  \[ \text{or} \]
  
  \[ \text{Memory[ALUOut]} = B; \]

- R-type instructions finish
  
  \[ \text{Reg[IR[15-11]]} = \text{ALUOut}; \]

The write actually takes place at the end of the cycle on the edge

Step 5: Memory Read Completion

- \[ \text{Reg[IR[20-16]]} = \text{MDR}; \]

What about all the other instructions?

Which operations do we need?

- Set “MemtoReg” to “1”
- Assert “RegWrite” signal
- Set “RegDst” to “0”
Summary:

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>IR = Memory[PC]</td>
<td>PC = PC + 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction decoder/register fetch</td>
<td>A = Reg[IR[25:21]]</td>
<td>B = Reg[IR[20:16]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALUOut = PC + (sign-extend(IR[15:0]) &lt;&lt; 2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/ jump completion</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A + sign-extend(IR[15:0])</td>
<td>if (A == B) then PC = ALUOut</td>
<td>PC = PC[31:28]</td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>Reg[IR[15:11]] = ALUOut</td>
<td>Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td>Load: Reg[IR[23:16]] = MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cleveland State University

13

c.yu91@csuohio.edu

---

Execution time for “add”

```
add $8, $17, $18
```

<table>
<thead>
<tr>
<th>000000</th>
<th>10001</th>
<th>10010</th>
<th>01000</th>
<th>00000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
</tbody>
</table>

![Diagram of instruction fetch, register read, ALU operation, register write, and total time]

- Instruction fetch: 2ns
- Register read: 1ns
- ALU operation: 2ns
- Register write: 1ns
- **Total**: 6ns

Cleveland State University

14

c.yu91@csuohio.edu
Execution time for “load”

```
lw  $1, 100($2)
```

```
110101  00010  00001  0000 0000 0110 0100
```

- Instruction fetch: 2ns
- Register read: 1ns
- ALU operation: 2ns
- Memory read: 2ns
- Register write: 1ns

Total: 8ns

---

5 Stages in MIPS

Instruction fetch | Reg. read | ALU operation | Reg. write
---|---|---|---
Instruction fetch | Reg. read | ALU operation | Reg. write
Instruction fetch | Reg. read | ALU operation | Reg. write
Instruction fetch | Reg. read | ALU operation | Reg. write
Instruction fetch | Reg. read | ALU operation | Reg. write

What’s changed?
Speedup

- Improve performance by increasing instruction throughput

![Diagram showing program execution order and ideal speedup]

**Ideal speedup is equal to what? Do we achieve this?**

Stage Length

- **Clock cycle**
  - Each of 5 stages takes the similar amount of time
  - And it must be small as much as possible

- **What makes it easy**
  - all instructions are the same length
  - just a few instruction formats
Stage Length

Stages' may require different amounts of time
Clock cycle time = maximum length of any stage.

Why we can write & read the same register values in the same cycle. Write during 1st half, read during 2nd half.

Lessons from Pipelined Laundry

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
- Multiple tasks operating simultaneously using different resources – any dependencies, any conflicts ??
Can pipelining get us into trouble?

If any two stages use the same resource, there must be a conflict.

Hazards

*Hazard* = when an instruction’s stage is unable to execute during the current cycle.

- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards

Instruction #2 stage 3 unable to continue.
Hazards

- Resource conflict
  - At any moment, 5 pipeline stages are all active doing something but with different instructions
  - A resource in each stage must not be used for other stages

- What makes it easy
  - memory operands appear only in loads and stores

- What makes it hard
  - structural hazards: suppose we had only one memory (<= we already remove this type of hazards)
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction

Structural Hazards

A needed functional unit is busy executing a previous instruction (Attempt to use the same resource two different ways at the same time)

Example:
- Our sample MIPS pipeline has none.
- What if PC+4 computation used main ALU instead of separate adder?
Resources used in 5 Stages

- Instruction Fetch (IF)
- Instruction Decode/Register Fetch (ID)
- Execute, Address computation, Brach/Jump completion (EX)
- Memory access or R-type completion (MEM)
- Memory read completion (WB)

Now, what are the problems? And what are the solutions?

Resource Conflicts

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type Instructions</th>
<th>Action for memory-reference Instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td></td>
<td>PC = PC + 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td></td>
<td>A = Reg[IR[5:2]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address</td>
<td>ALUOut = A = \text{sign-extend}(IR[15:0])</td>
<td>B = Reg[IR[20:16]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>computation, branch/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jump completion</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory access or R-type</td>
<td>Load: MDR = Memory[ALUOut]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td>or Store: Memory[ALUOut] = B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td>Load: Reg[IR[20:16]] = MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ALU conflict

Memory conflict

Register file conflict (read or write)
Control Hazards

While executing a previous branch, next instruction address might not yet be known.
(attempt to make a decision before condition is evaluated)

Performs branch test & sets PC to target.

Data Hazards

Needed data still being computed by previous instruction.
(attempt to use item before it is ready)