Today's class

• Pipelining
  – Speed up
  – How it is done
  – Hazards
Pipelining: Its Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes
Sequential laundry takes 6 hours for 4 loads
If they learned pipelining, how long would laundry take?
Pipelined Laundry
Start work ASAP

• Pipelined laundry takes 3.5 hours for 4 loads
Key Definitions

*Pipelining* is a key implementation technique used to build fast processors. It allows the execution of multiple instructions to overlap in time.

A pipeline within a processor is similar to a car assembly line. Each assembly station is called a *pipe stage* or a *pipe segment*.

The *throughput* of an instruction pipeline is the measure of how often an instruction exits the pipeline.
Speed up

• Speedup for 4 loads
  – Old_execution_time / New_execution_time
    = 4 (wash_time + dry_time + fold_time) / (wash_time + 4 dry_time + fold_time)

• Speedup for N loads
  – N (wash_time + dry_time + fold_time) / (wash_time + N dry_time + fold_time)

=> max_speedup = (wash_time + dry_time + fold_time) / dry_time

=> max_speedup = execution_time_of_one / slowest_stage
Pipeline Stages

We can divide the execution of an instruction into the following 5 “classic” stages:

**IF:** Instruction Fetch  
**ID:** Instruction Decode, register fetch  
**EX:** Execution  
**MEM:** Memory Access  
**WB:** Register write Back
Pipeline Throughput and Latency

Consider the pipeline above with the indicated delays. We want to know what is the pipeline throughput and the pipeline latency.

Pipeline throughput: instructions completed per second.

Pipeline latency: how long does it take to execute a single instruction in the pipeline.
Pipeline Throughput and Latency

Pipeline throughput: how often an instruction is completed.

\[ \frac{1 \text{ instr}}{\max \left[ \text{lat}(\text{IF}), \text{lat}(\text{ID}), \text{lat}(\text{EX}), \text{lat}(\text{MEM}), \text{lat}(\text{WB}) \right]} \]

\[ = \frac{1 \text{ instr}}{\max \left[ 5 \text{ ns}, 4 \text{ ns}, 5 \text{ ns}, 10 \text{ ns}, 4 \text{ ns} \right]} \]

\[ = \frac{1 \text{ instr}}{10 \text{ ns}} \quad (\text{ignoring pipeline register overhead}) \]

Pipeline latency: how long does it take to execute an instruction in the pipeline.

\[ L = \text{lat}(\text{IF}) + \text{lat}(\text{ID}) + \text{lat}(\text{EX}) + \text{lat}(\text{MEM}) + \text{lat}(\text{WB}) \]

\[ = 5 \text{ ns} + 4 \text{ ns} + 5 \text{ ns} + 10 \text{ ns} + 4 \text{ ns} = 28 \text{ ns} \]

Is this right?
Pipeline Throughput and Latency

Simply adding the latencies to compute the pipeline latency, only would work for an isolated instruction.

I1  IF  ID  EX  MEM  WB  \( L(I1) = 28\text{ns} \)
I2  IF  ID  EX  MEM  WB  \( L(I2) = 33\text{ns} \)
I3  IF  ID  EX  MEM  WB  \( L(I3) = 38\text{ns} \)
I4  IF  ID  EX

We are in trouble! The latency is not constant. This happens because this is an unbalanced pipeline. The solution is to make every state the same length as the longest one.

I5  IF  ID  EX  MEM  WB  \( L(I5) = 43\text{ns} \)
Pipelining Lessons

• Pipelining doesn’t help latency of single task, it helps throughput of entire workload
• Pipeline rate limited by slowest pipeline stage
• Multiple tasks operating simultaneously
• Potential speedup = Number pipe stages
• Unbalanced lengths of pipe stages reduces speedup
• Time to “fill” pipeline and time to “drain” it reduces speedup
Other Definitions

• Pipe stage or pipe segment
  – A decomposable unit of the fetch-decode-execute paradigm

• Pipeline depth
  – Number of stages in a pipeline

• Machine cycle
  – Clock cycle time

• Latch
  – Per phase/stage local information storage unit
Design Issues

• Balance the length of each pipeline stage

\[
\text{Throughput} = \frac{\text{Depth of the pipeline}}{\text{Time per instruction on unpipelined machine}}
\]

• Problems
  – Usually, stages are not balanced
  – Pipelining overhead
  – Hazards (conflicts)

• Performance (throughput → CPU performance equation)
  – Decrease of the CPI
  – Decrease of cycle time
## MIPS Instruction Formats

### I Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>0-4</td>
</tr>
<tr>
<td>Rs1</td>
<td>5-6</td>
</tr>
<tr>
<td>Rd</td>
<td>10-11</td>
</tr>
<tr>
<td>Immediate</td>
<td>15-31</td>
</tr>
</tbody>
</table>

### R Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>0-4</td>
</tr>
<tr>
<td>Rs1</td>
<td>5-6</td>
</tr>
<tr>
<td>Rs2</td>
<td>10-11</td>
</tr>
<tr>
<td>Rd</td>
<td>15-16</td>
</tr>
<tr>
<td>Shamts/Function</td>
<td>20-21</td>
</tr>
</tbody>
</table>

### J Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>0-4</td>
</tr>
<tr>
<td>Address</td>
<td>5-6</td>
</tr>
</tbody>
</table>

**Fixed-field decoding**
1st and 2nd Instruction cycles

- **Instruction fetch (IF)**
  
  IR $\leftarrow$ Mem[PC];
  NPC $\leftarrow$ PC + 4

- **Instruction decode & register fetch (ID)**
  
  A $\leftarrow$ Regs[IR_{6..10}];
  B $\leftarrow$ Regs[IR_{11..15}];
  Imm $\leftarrow$ ((IR_{16})^{16}$#$ IR_{16..31})
3rd Instruction cycle

• Execution & effective address (EX)
  – Memory reference
    • ALUOutput ← A + Imm
  – Register - Register ALU instruction
    • ALUOutput ← A \text{func} B
  – Register - Immediate ALU instruction
    • ALUOutput ← A \text{op} Imm
  – Branch
    • ALUOutput ← NPC + Imm; Cond (A \text{op} 0)
4th Instruction cycle

- Memory access & branch completion (MEM)
  - Memory reference
    - PC  NPC
    - LMD  Mem[ALUOutput]  (load)
    - Mem[ALUOutput]  B  (store)
  - Branch
    - if (cond) PC  ALUOutput; else PC  NPC
5th Instruction cycle

• Write-back (WB)
  – Register - register ALU instruction
    • Regs[IR_{16..20}] ← ALUOutput
  – Register - immediate ALU instruction
    • Regs[IR_{11..15}] ← ALUOutput
  – Load instruction
    • Regs[IR_{11..15}] ← LMD
5 Steps of MIPS Datapath

Instruction Fetch

Instr. Decode Reg. Fetch

Execute Addr. Calc

Memory Access

Write Back

Next PC

Next SEQ PC

WD Data

Address

Adder

Memory

Reg File

ALU

Data Memory

Sign Extend

Imm

RS1

RS2

RD

Zero?

LMD

MUX

MUX

MUX

MUX
5 Steps of MIPS Datapath

- Instruction Fetch
- Instr. Decode
- Reg. Fetch
- Execute
- Addr. Calc
- Memory Access
- Write Back

- Data stationary control
  - local decode for each instruction phase / pipeline stage
Control

Step 1

Step 2

RR ALU

Imm

Store

Load

Step 3

Step 4

Step 3

Step 4

Step 3

Step 4

Step 3

Step 4

Step 5
# Basic Pipeline

<table>
<thead>
<tr>
<th>Instr #</th>
<th>Clock number</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>1</td>
</tr>
<tr>
<td>$i + 1$</td>
<td>IF</td>
</tr>
<tr>
<td>$i + 2$</td>
<td>IF</td>
</tr>
<tr>
<td>$i + 3$</td>
<td>IF</td>
</tr>
<tr>
<td>$i + 4$</td>
<td>IF</td>
</tr>
</tbody>
</table>
Pipeline Resources
Pipelined Datapath

IF/ID → ID/EX → EX/MEM → MEM/WB

PC → Instr. Cache → Add Mux → 4

Regs → ALU Mux → Zero?

Sign extend → Mux

Data Cache → Mux
Performance limitations

• Imbalance among pipe stages
  – limits cycle time to slowest stage

• Pipelining overhead
  – Pipeline register delay
  – Clock skew

• Clock cycle > clock skew + latch overhead

• Hazards