Computer Science 104: Pipelining

Alvin R. Lebeck

Slides based on those from Randy Bryant, G. Kedem

Administrative

Homework #5

Due March 23, 11:59pm
Please use Word or whatever and submit one document

Midterm II, Monday March 28 in class, closed book/notes

Reading 4.4 (4.5 if you want)
SEQ Problems

- Stages occur in sequence
- One operation in process at a time
- Very slow, but works!

Understanding Performance

- Execution time = Inst/prog * cycles/inst * seconds/cycle
- Inst/prog: \(\sim O(n)\)...take algorithms
- Cycles/inst: for SEQ = 1
- Seconds/cycle: for SEQ assume 5ns
- **Problem with SEQ: every instruction takes full 5ns**
  - E.g., ADD doesn’t need to use data memory
Overview

General Principles of Pipelining
- Goal
- Difficulties

Creating a Pipelined Y86 Processor
- Rearranging SEQ
- Inserting pipeline registers
- Problems with data and control hazards

Real-World Pipelines: Car Washes

Sequential

Parallel

Idea
- Divide process into independent stages
- Move objects through stages in sequence
- At any given time, multiple objects being processed
### Computational Example

- **Delay = 320 ps**
- **Throughput = 3.12 GOPS**

#### System
- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Can must have clock cycle of at least 320 ps

---

### 3-Way Pipelined Version

- **Delay = 360 ps**
- **Throughput = 8.33 GOPS**

#### System
- Divide combinational logic into 3 blocks of 100 ps each
- Can begin new operation as soon as previous one passes through stage A.
  - Begin new operation every 120 ps
- Overall latency increases
  - 360 ps from start to finish
Pipeline Diagrams

Unpipelined

- Cannot start new operation until previous one completes

3-Way Pipelined

- Up to 3 operations in process simultaneously

Operating a Pipeline
Limitations: Nonuniform Delays

- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Delay = 510 ps
Throughput = 5.88 GOPS

Limitations: Register Overhead

- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

Delay = 420 ps, Throughput = 14.29 GOPS
SEQ Hardware

- Stages occur in sequence
- One operation in process at a time

SEQ+ Hardware

- Still sequential implementation
- Reorder PC stage to put at beginning

PC Stage

- Task is to select PC for current instruction
- Based on results computed by previous instruction

Processor State

- PC is no longer stored in register
- But, can determine PC based on other stored information
Adding Pipeline Registers

Pipeline Stages

**Fetch**
- Select current PC
- Read instruction
- Compute incremented PC

**Decode**
- Read program registers

**Execute**
- Operate ALU

**Memory**
- Read or write data memory

**Write Back**
- Update register file
The Five Stages of mrmovl

Ifetch: Instruction Fetch
- Fetch the instruction from the Instruction Memory

Reg/Dec: Registers Fetch and Instruction Decode

Exec: Calculate the memory address

Mem: Read the data from the Data Memory

WrB: Write the data back to the register file

Key Ideas Behind Instruction Execution Pipelining

Overlap execution of instructions

The mrmovl instruction has 5 stages: I-fetch, Reg-Fetch / I-Decode, Execute, Memory-Access, Register Write-Back.

- Five independent functional units to work on each stage
  - Each functional unit is used only once

- The 2nd mrmovl can start as soon as the 1st finishes its Ifetch stage

- Each mrmovl still takes five cycles to complete. Latency is still 5 cycles

- The throughput is much higher; CPI is 1 with ~1/5 cycle time.

- Instructions start before the previous ones are completed.
Pipelining the mrmovl Instruction

The five independent functional units in the pipeline datapath are:
- Instruction Memory for the Ifetch stage
- Register File’s Read ports for the Reg/Dec stage
- ALU for the Exec stage
- Data Memory for the Mem stage
- Register File’s Write port for the WrB stage

One instruction enters the pipeline every cycle
- One instruction comes out of the pipeline (completed) every cycle
- The “Effective” Cycles per Instruction (CPI) is 1; ~1/5 cycle time

The Four Stages of Int Op

Ifetch: Instruction Fetch
- Fetch the instruction from the Instruction Memory

Reg/Dec: Register access and Instruction Decode

Exec: ALU operates on the two register operands

WrB: Write the ALU output back to the register file
Pipelining the IOP and mrmovl Instructions

We have a problem called a pipeline conflict or resource hazard:
- Two instructions try to write to the register file at the same time!

OOPS! We have a problem!

Important Observation

Each functional unit can only be used once per instruction
Each functional unit must be used in the same stage for all instructions:
- Mrmovl uses Register File’s Write Port during its 5th stage

<table>
<thead>
<tr>
<th>Load</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>WrB</td>
<td></td>
</tr>
</tbody>
</table>

- IOP uses Register File’s Write Port during its 4th stage

<table>
<thead>
<tr>
<th>R-type</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>WrB</td>
<td></td>
</tr>
</tbody>
</table>

° How to solve this pipeline hazard?
**Solution: Delay IOP’s Write by One Cycle**

Delay R-type’s register write by one cycle:
- Now IOP instructions also use Reg File’s write port at Stage 5
- Mem stage is a NO-OP stage: nothing is being done. **Effective CPI?**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
<th>Cycle 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOP</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wrb</td>
<td>Mem</td>
<td>Wrb</td>
<td>Mem</td>
</tr>
</tbody>
</table>

**Data Dependencies**

- Each operation depends on result from preceding one

**System**

- Each operation depends on result from preceding one
Data Dependencies in Programs

- Result from one instruction used as operand for another
  - Read-after-write (RAW) dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

Example:

```c
1. irmovl $50, %eax
2. addl %eax, %ebx
3. mrmovl 100( %ebx), %edx
```

Data Hazards

- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system
- How can we avoid data hazards (i.e., ensure we get correct answer)?
Solving Data Hazards

```assembly
imovl $50, %eax
addl %eax, %ebx  # depends on imovl
mrmovl 100(%ebx), %edx  # depends on addl
```

- Insert nops, or
- Stall (i.e., hold addl in Fetch until imovl is finished), or
- Modify datapath to pass/forward value

---

What else can go wrong!

```assembly
imovl $50, %eax
addl %eax, %ebx
jne foo
mrmovl 100(%ebx), %edx
addl %ebx, %edx
```

```
foo:
imovl $220, %edx
subl %ebx, %ecx
```
Control Dependency

```plaintext
irmovl $50, %eax
addl %eax, %ebx
jne foo
```
```
mrmovl 100(%ebx), %edx  # control dependent on jne
addl %ebx, %edx
```
```
foo:
```
```
irmovl $220, %edx
```
```
subl %ebx, %ecx
```

- We don’t know next PC!
- What should we do?

Predicting the PC

- Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
- Guess which instruction will follow
  - Recover if prediction was incorrect
One Prediction Strategy

Instructions that Don’t Transfer Control
- Predict next PC to be valP
- Always reliable

Call and Unconditional Jumps
- Predict next PC to be valC (destination)
- Always reliable

Conditional Jumps
- Predict next PC to be valC (destination)
- Only correct if branch is taken
  - Typically right 60% of time

Return Instruction
- Don’t try to predict

What if we guess wrong?

Recovering from Branch Misprediction

Clock
- Cycle 1
- Cycle 2
- Cycle 3
- Cycle 4
- Cycle 5
- Cycle 6
- Cycle 7
- Cycle 8
- Cycle 9

irmovl  Ifetch  Reg/Dec  Exec  Mem  Wr
addl   Ifetch  Reg/Dec  Exec  Mem  Wr
jne    Ifetch  Reg/Dec  Exec  Mem  Wr
irmovl  Ifetch  Reg/Dec  Exec  Mem  Wr
addl   Ifetch  Reg/Dec  Exec  Mem  Wr
mrmovl Ifetch  Reg/Dec  Exec  Mem  W
Pipeline Summary

Concept
- Break instruction execution into 5 stages
- Run instructions through in pipelined mode
- Latency of one instruction is ~ same, but throughput increases

Limitations
- Can’t handle dependencies between instructions when instructions follow too closely
- Data dependencies
  - One instruction writes register, later one reads it
- Control dependency
  - Instruction sets PC in way that pipeline did not predict correctly
  - Mispredicted branch and return