Systems I

Pipelining I

Topics

- Pipelining principles
- Pipeline overheads
- Pipeline registers and stages
Overview

What’s wrong with the sequential (SEQ) Y86?
- It’s slow!
- Each piece of hardware is used only a small fraction of time
- We would like to find a way to get more performance with only a little more hardware

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

**Idea**
- Divide process into independent stages
- Move objects through stages in sequence
- At any given times, multiple objects being processed
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

Slide courtesy of D. Patterson
Sequential Laundry

Sequential laundry takes 8 hours for 4 loads

If they learned pipelining, how long would laundry take?

Slide courtesy of D. Patterson
Pipelined Laundry: Start ASAP

Pipelined laundry takes 3.5 hours for 4 loads!

Slide courtesy of D. Patterson
Pipelining Lessons

Pipelining doesn’t help latency of single task, it helps throughput of entire workload.

Multiple tasks operating simultaneously using different resources.

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.

Stall for Dependences.

Slide courtesy of D. Patterson
Latency and Throughput

Latency: time to complete an operation
Throughput: work completed per unit time

Consider plumbing

- Low latency: turn on faucet and water comes out
- High bandwidth: lots of water (e.g., to fill a pool)

What is “High speed Internet?”

- Low latency: needed to interactive gaming
- High bandwidth: needed for downloading large files
- Marketing departments like to conflate latency and bandwidth…
Relationship between Latency and Throughput

Latency and bandwidth only loosely coupled

- Henry Ford: assembly lines increase bandwidth without reducing latency

My factory takes 1 day to make a Model-T ford.

- But I can start building a new car every 10 minutes
- At 24 hrs/day, I can make $24 \times 6 = 144$ cars per day
- A special order for 1 green car, still takes 1 day
- Throughput is increased, but latency is not.

Latency reduction is difficult

Often, one can buy bandwidth

- E.g., more memory chips, more disks, more computers
- Big server farms (e.g., google) are high bandwidth
Computational Example

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

**System**

- Delay = 320 ps
- Throughput = 3.12 GOPS
3-Way Pipelined Version

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

Delay = 360 ps
Throughput = 8.33 GOPS
Pipeline Diagrams

Unpipelined

- Cannot start new operation until previous one completes

3-Way Pipelined

- Up to 3 operations in process simultaneously
Operating a Pipeline

Clock
OP1
OP2
OP3

Time

0 120 240 360 480 640

100 ps 20 ps 100 ps 20 ps 100 ps 20 ps

Comb. logic A Reg Comb. logic B Reg Comb. logic C Reg

Clock
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
CPU Performance Equation

3 components to execution time:

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

Factors affecting CPU execution time:

<table>
<thead>
<tr>
<th></th>
<th>Inst. Count</th>
<th>CPI</th>
<th>Clock Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>X</td>
<td>(X)</td>
<td></td>
</tr>
<tr>
<td>Inst. Set</td>
<td>X</td>
<td>X</td>
<td>(X)</td>
</tr>
<tr>
<td>Organization</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MicroArch</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Technology</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

- Consider all three elements when optimizing
- Workloads change!
Cycles Per Instruction (CPI)

**Depends on the instruction**

\[ CPI_i = \text{Execution time of instruction } i \times \text{Clock Rate} \]

**Average cycles per instruction**

\[
CPI = \sum_{i=1}^{n} CPI_i \times F_i \quad \text{where } F_i = \frac{IC_i}{IC_{tot}}
\]

**Example:**

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI(i)</th>
<th>%time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>0.5</td>
<td>33%</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>2</td>
<td>0.4</td>
<td>27%</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>2</td>
<td>0.2</td>
<td>13%</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>0.4</td>
<td>27%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CPI(total)</td>
<td>1.5</td>
</tr>
</tbody>
</table>
Comparing and Summarizing Performance

Fair way to summarize performance?

Capture in a single number?

Example: Which of the following machines is best?

<table>
<thead>
<tr>
<th></th>
<th>Computer A</th>
<th>Computer B</th>
<th>Computer C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program 1</td>
<td>1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Program 2</td>
<td>1000</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>Total Time</td>
<td>1001</td>
<td>110</td>
<td>40</td>
</tr>
</tbody>
</table>
Means

Arithmetic mean

\[ \frac{1}{n} \sum_{i=1}^{n} T_i \]

Can be weighted: \( a_i T_i \)

Represents total execution time

Should not be used for aggregating
normalized numbers

Geometric mean

\[ \left( \prod_{i=1}^{n} T_i \right)^{\frac{1}{n}} \]

Consistent independent of reference

Best for combining results

Best for normalized results

\[ \ln(Geo) = \frac{1}{n} \sum_{i=1}^{n} \ln(T_i) \]
What is the geometric mean of 2 and 8?

- A. 5
- B. 4
Is Speed the Last Word in Performance?

Depends on the application!

Cost
- Not just processor, but other components (ie. memory)

Power consumption
- Trade power for performance in many applications

Capacity
- Many database applications are I/O bound and disk bandwidth is the precious commodity
Revisiting the Performance Eqn

CPU time = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}

**Instruction Count:** No change

**Clock Cycle Time**
- Improves by factor of almost \( N \) for \( N \)-deep pipeline
- Not quite factor of \( N \) due to pipeline overheads

**Cycles Per Instruction**
- In ideal world, CPI would stay the same
- An individual instruction takes \( N \) cycles
- But we have \( N \) instructions in flight at a time
- So - average \( \text{CPI}_{\text{pipe}} = \text{CPI}_{\text{no\_pipe}} \times \frac{N}{N} \)

Thus performance can improve by up to factor of \( N \)
Data Dependencies

- 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

```
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
SEQ Hardware

- **Stages occur in sequence**
- **One operation in process at a time**
- **One stage for each logical pipeline operation**
  - Fetch (get next instruction from memory)
  - Decode (figure out what instruction does and get values from regfile)
  - Execute (compute)
  - Memory (access data memory if necessary)
  - Write back (write any instruction result to regfile)
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

- **Fetch**
  - PC
  - Instruction memory
  - PC increment
  - predPC

- **Decode**
  - icode, valC, ifun, rA, rB, valC
  - A, B Register file

- **Execute**
  - ALU
  - srcA, srcB, dstA, dstB
  - valuA, valuB
  - Bch
  - valuE

- **Memory**
  - Addr, Data
  - Data memory

- **Write back**
  - valE, valM
  - valE, valM
  - valE, valM
  - valE, valM

- **Instruction memory**
  - Instruction

- **PC increment**
  - PC
  - Increment

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
Summary

Today

- Pipelining principles (assembly line)
- Overheads due to imperfect pipelining
- Breaking instruction execution into sequence of stages

Next Time

- Pipelining hardware: registers and feedback paths
- Difficulties with pipelines: hazards
- Method of mitigating hazards