Performance and Pipelining

Prof. Hakim Weatherspoon
CS 3410, Spring 2015
Computer Science
Cornell University

See P&H Chapter: 1.6, 4.5-4.6
That's it. We surrender. Winter, you win. Key West anyone?

Due to this ridiculously stupid winter, Ithaca invites you to visit The Florida Keys this week. Please come back when things thaw out. Really, it's for the birds here now. (Still want to Visit Ithaca? Are you sure? Ok, click here.)
P.S. Send us a postcard.

VisitIthaca.com

The southernmost city in the continental USA.
Announcements

HW 1
   Quite long. Do not wait till the end.
Project 1 design doc
   Critical to do this, else Project 1 will be hard

HW 1 review session
   Wed (2/18) @ 7:30pm and Sun (2/22) @ 5:00pm
   Locations: Both in Upson B17

Prelim 1 review session
   Next Tue (2/24) and Sun(2/28). 7:30pm.
   Location: Olin 255 and Upson B17, respectively.
Goals for today

Performance

• What is performance?
• How to get it?

Pipelining
Performance

Complex question

• How fast is the processor?
• How fast your application runs?
• How quickly does it respond to you?
• How fast can you process a big batch of jobs?
• How much power does your machine use?
Measures of Performance

Clock speed

- 1 KHz, $10^3$ Hz: cycle is 1 millisecond, ms, $(10^{-6})$
- 1 MHz, $10^6$ Hz: cycle is 1 microsecond, us, $(10^{-6})$
- 1 Ghz, $10^9$ Hz: cycle is 1 nanosecond, ns, $(10^{-9})$
- 1 Thz, $10^{12}$ Hz: cycle is 1 picosecond, ps, $(10^{-12})$

Instruction/application performance

- MIPs (Millions of instructions per second)
- FLOPs (Floating point instructions per second)
  - GPUs: GeForce GTX Titan (2,688 cores, 4.5 Tera flops, 7.1 billion transistors, 42 Gigapixel/sec fill rate, 288 GB/sec)
- Benchmarks (SPEC)
Measures of Performance

Latency

- How long to finish my program
  - Response time, elapsed time, wall clock time
  - CPU time: user and system time

Throughput

- How much work finished per unit time

Ideal: Want high throughput, low latency

... also, low power, cheap ($$) etc.
How to make the computer faster?

Decrease latency

Critical Path

• Longest path determining the minimum time needed for an operation
  • Determines minimum length of clock cycle
    i.e. determines maximum clock frequency

Optimize for latency on the critical path
  – Parallelism (like carry look ahead adder)
  – Pipelining
  – Both
Latency: Optimize Delay on Critical Path

E.g. Adder performance

<table>
<thead>
<tr>
<th>32 Bit Adder Design</th>
<th>Space</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ripple Carry</td>
<td>≈ 300 gates</td>
<td>≈ 64 gate delays</td>
</tr>
<tr>
<td>2-Way Carry-Skip</td>
<td>≈ 360 gates</td>
<td>≈ 35 gate delays</td>
</tr>
<tr>
<td>3-Way Carry-Skip</td>
<td>≈ 500 gates</td>
<td>≈ 22 gate delays</td>
</tr>
<tr>
<td>4-Way Carry-Skip</td>
<td>≈ 600 gates</td>
<td>≈ 18 gate delays</td>
</tr>
<tr>
<td>2-Way Look-Ahead</td>
<td>≈ 550 gates</td>
<td>≈ 16 gate delays</td>
</tr>
<tr>
<td>Split Look-Ahead</td>
<td>≈ 800 gates</td>
<td>≈ 10 gate delays</td>
</tr>
<tr>
<td>Full Look-Ahead</td>
<td>≈ 1200 gates</td>
<td>≈ 5 gate delays</td>
</tr>
</tbody>
</table>
Multi-Cycle Instructions

But what to do when operations take diff. times?

E.g: Assume:

- load/store: 100 ns $\leftarrow$ 10 MHz $\quad$ ms = 10^{-3} second
  $\quad$ us = 10^{-6} seconds
  $\quad$ ns = 10^{-9} seconds
  $\quad$ ps = 10^{-12} seconds
- arithmetic: 50 ns $\leftarrow$ 20 MHz
- branches: 33 ns $\leftarrow$ 30 MHz

Single-Cycle CPU

10 MHz (100 ns cycle) with
  - 1 cycle per instruction
Multi-Cycle Instructions
Multiple cycles to complete a single instruction

E.g: Assume:

- load/store: 100 ns \( \leftarrow \) 10 MHz \( \text{ms} = 10^{-3} \text{ second} \)
- arithmetic: 50 ns \( \leftarrow \) 20 MHz \( \text{us} = 10^{-6} \text{ seconds} \)
- branches: 33 ns \( \leftarrow \) 30 MHz \( \text{ns} = 10^{-9} \text{ seconds} \)

\( \text{ps} = 10^{-12} \text{ seconds} \)

Which one is faster: Single- or Multi-Cycle CPU?

Single-Cycle CPU
10 MHz (100 ns cycle) with
– 1 cycle per instruction

Multi-Cycle CPU
30 MHz (33 ns cycle) with
– 3 cycles per load/store
– 2 cycles per arithmetic
– 1 cycle per branch
Cycles Per Instruction (CPI)

*Instruction mix* for some program P, assume:
- 25% load/store (3 cycles/instruction)
- 60% arithmetic (2 cycles/instruction)
- 15% branches (1 cycle/instruction)

Multi-Cycle performance for program P:

\[
3 \times 0.25 + 2 \times 0.60 + 1 \times 0.15 = 2.1
\]

Average *cycles per instruction* (CPI) = 2.1

Multi-Cycle @ 30 MHz \(\leftarrow\) 30M cycles/sec ÷ 2.1 cycles/instr \(\approx\) 15 MIPS

Single-Cycle @ 10 MHz

10 MIPS = 10M cycles/sec ÷ 1 cycle/instr

MIPS = millions of instructions per second
Total Time

CPU Time = # Instructions x CPI x Clock Cycle Time

= Instr x cycles/instr x seconds/cycle

E.g. Say for a program with 400k instructions, 30 MHz:
CPU [Execution] Time = ?
**Total Time**

\[
\text{CPU Time} = \# \text{Instructions} \times \text{CPI} \times \text{Clock Cycle Time}
\]

\[
= \text{Instr} \times \text{cycles/instr} \times \text{seconds/cycle}
\]

E.g. Say for a program with 400k instructions, 30 MHz:

\[
\text{CPU [Execution] Time} = 400k \times 2.1 \times 33 \text{ ns} = 27 \text{ ms}
\]
Total Time

CPU Time = # Instructions x CPI x Clock Cycle Time

= Instr x cycles/instr x seconds/cycle

E.g. Say for a program with 400k instructions, 30 MHz:
CPU [Execution] Time = 400k x 2.1 x 33 ns = 27 ms

How do we increase performance?

• Need to reduce CPU time
  ▪ Reduce #instructions
  ▪ Reduce CPI
  ▪ Reduce Clock Cycle Time
Example

Goal: Make Multi-Cycle @ 30 MHz CPU (15MIPS) run 2x faster by making arithmetic instructions faster

*Instruction mix (for P):*
  - 25% load/store,  CPI = 3
  - 60% arithmetic,  CPI = 2
  - 15% branches,  CPI = 1

\[
\text{CPI} = 0.25 \times 3 + 0.6 \times 2 + 0.15 \times 1
\]
\[
= 2.1
\]

Goal: Make processor run 2x faster,
  i.e. 30 MIPS instead of 15 MIPS
Example

Goal: Make Multi-Cycle @ 30 MHz CPU (15MIPS) run 2x faster by making arithmetic instructions faster

*Instruction mix (for P)*:
- 25% load/store, CPI = 3
- 60% arithmetic, CPI = 2
- 15% branches, CPI = 1

\[
\text{CPI} = 0.25 \times 3 + 0.6 \times 2 + 0.15 \times 1 = 1.5
\]

First lets try CPI of 1 for arithmetic.

Is that 2x faster overall? No

How much does it improve performance?
Example

Goal: Make Multi-Cycle @ 30 MHz CPU (15MIPS) run 2x faster by making arithmetic instructions faster

*Instruction mix (for P)*:
- 25% load/store, CPI = 3
- 60% arithmetic, CPI = 2X
- 15% branches, CPI = 1

\[
\text{CPI} = 1.05 = 0.25 \times 3 + 0.6 \times X + 0.15 \times 1
\]

\[
1.05 = 0.75 + 0.6X + 0.15
\]

\[
X = 0.25
\]

But, want to half our CPI from 2.1 to 1.05.

Let new arithmetic operation have a CPI of X. \(X = ?\)

Then, \(X = 0.25\), which is a significant improvement.
Example

Goal: Make Multi-Cycle @ 30 MHz CPU (15MIPS) run 2x faster by making arithmetic instructions faster

*Instruction mix (for P):*

- 25% load/store, CPI = 3
- 60% arithmetic, CPI = 2 0.25
- 15% branches, CPI = 1

To double performance CPI for arithmetic operations have to go from 2 to 0.25
Amdahl’s Law

Amdahl’s Law

\[
\text{Execution time after improvement} = \frac{\text{execution time affected by improvement}}{\text{amount of improvement}} + \text{execution time unaffected}
\]

Or: Speedup is limited by popularity of improved feature

Corollary: Make the common case fast

Caveat: Law of diminishing returns
Review: Single Cycle Processor
Review: Single Cycle Processor

Advantages

• Single cycle per instruction make logic and clock simple

Disadvantages

• Since instructions take different time to finish, memory and functional unit are not efficiently utilized

• Cycle time is the longest delay
  – Load instruction

• Best possible CPI is 1 (actually < 1 w parallelism)
  – However, lower MIPS and longer clock period (lower clock frequency); hence, lower performance
Review: Multi Cycle Processor

Advantages

• Better MIPS and smaller clock period (higher clock frequency)
• Hence, better performance than Single Cycle processor

Disadvantages

• Higher CPI than single cycle processor

Pipelining: Want better Performance

• want small CPI (close to 1) with high MIPS and short clock period (high clock frequency)
Improving Performance

Parallelism

Pipelining

Both!
Single Cycle vs Pipelined Processor

See: P&H Chapter 4.5
Alice

Bob

They don’t always get along...
The Bicycle
The Materials

Saw

Drill

Glue

Paint
The Instructions

N pieces, each built following same sequence:

Saw → Drill → Glue → Paint
Design 1: Sequential Schedule

Alice owns the room
Bob can enter when Alice is finished
Repeat for remaining tasks
No possibility for conflicts
Sequential Performance

Latency: 4 hours/task
Throughput: 1 task/4 hrs
Concurrency: 1

CPI = 4

Can we do better?
Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*

One person owns a stage at a time

4 stages

4 people working simultaneously

Everyone moves right in lockstep
Design 2: Pipelined Design
Partition room into *stages* of a *pipeline*

<table>
<thead>
<tr>
<th>Alice</th>
</tr>
</thead>
<tbody>
<tr>
<td>One person owns a stage at a time</td>
</tr>
<tr>
<td>4 stages</td>
</tr>
<tr>
<td>4 people working simultaneously</td>
</tr>
<tr>
<td>Everyone moves right in lockstep</td>
</tr>
<tr>
<td>It still takes all four stages for one job to complete</td>
</tr>
</tbody>
</table>
Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*

Bob    Alice

One person owns a stage at a time
4 stages
4 people working simultaneously
Everyone moves right in lockstep
It still takes all four stages for one job to complete
Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*

One person owns a stage at a time

4 stages

4 people working simultaneously

Everyone moves right in lockstep

It still takes all four stages for one job to complete
Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*

Dave  Carol  Bob  Alice

One person owns a stage at a time

4 stages

4 people working simultaneously

Everyone moves right in lockstep

It still takes all four stages for one job to complete
Design 2: Pipelined Design
Partition room into *stages* of a *pipeline*

One person owns a stage at a time
4 stages
4 people working simultaneously
Everyone moves right in lockstep
It still takes all four stages for one job to complete
What if drilling takes twice as long, but gluing and paint take \( \frac{1}{2} \) as long?

Latency: 

Throughput: 

CPI =
What if drilling takes twice as long, but gluing and paint take ½ as long?

Latency: 4 cycles/task
Throughput: 1 task/2 cycles      CPI = 2
Lessons

Principle:

Throughput increased by parallel execution
Balanced pipeline very important
Else slowest stage dominates performance

Pipelining:

- Identify *pipeline stages*
- Isolate stages from each other
- Resolve pipeline *hazards* (next lecture)
MIPs designed for pipelining

• Instructions same length
  • 32 bits, easy to fetch and then decode

• 3 types of instruction formats
  • Easy to route bits between stages
  • Can read a register source before even knowing what the instruction is

• Memory access through lw and sw only
  • Access memory after ALU
Basic Pipeline
Five stage “RISC” load-store architecture

1. Instruction fetch (IF)
   - get instruction from memory, increment PC
2. Instruction Decode (ID)
   - translate opcode into control signals and read registers
3. Execute (EX)
   - perform ALU operation, compute jump/branch targets
4. Memory (MEM)
   - access memory if needed
5. Writeback (WB)
   - update register file
A Processor

Review: Single cycle processor
Latency: 5 cycles
Throughput: 1 instr/cycle
Concurrency: 5 CPI = 1
Principles of Pipelined Implementation

Break instructions across multiple clock cycles (five, in this case)

Design a separate stage for the execution performed during each clock cycle

Add pipeline registers (flip-flops) to isolate signals between different stages
Pipelined Processor

- Instruction Fetch (IF/ID)
  - PC
  - New pc
  - Instruction Fetch

- Instruction Decode (ID/EX)
  - Control
  - Extend
  - Register File

- Execute (EX/MEM)
  - ALU
  - Compute jump/branch targets

- Memory (MEM/WB)
  - Addr
  - Memory

- Write-Back

Flowchart:
- Memory → IF/ID
- IF/ID → ID/EX
- ID/EX → EX/MEM
- EX/MEM → MEM/WB
- MEM/WB → IF/ID
- IF/ID → New pc
- New pc → Instruction Fetch
- Instruction Fetch → IF/ID
Stage 1: Instruction Fetch

Fetch a new instruction every cycle
- Current PC is index to instruction memory
- Increment the PC at end of cycle (assume no branches for now)

Write values of interest to pipeline register (IF/ID)
- Instruction bits (for later decoding)
- PC+4 (for later computing branch targets)
- PC+4
- pcrel (PC-relative);   e.g. BEQ, BNE
- pcabs (PC absolute);  e.g. J and JAL
  . (PC+4)_{31..28} • target • 00
- pcreg (PC registers);  e.g. JR
IF

instruction memory

addr mc

00 = read word

PC

PC+4

Rest of pipeline

inst

new pc

pcsel

pcreg pc

pcrel

pcabs

IF/ID
Stage 2: Instruction Decode

On every cycle:

- Read IF/ID pipeline register to get instruction bits
- Decode instruction, generate control signals
- Read from register file

Write values of interest to pipeline register (ID/EX)

- Control information, Rd index, immediates, offsets, ...
- Contents of Ra, Rb
- PC+4 (for computing branch targets later)
Stage 1: Instruction Fetch

ID

PC+4

inst

IF/ID

IF/ID

WE

Rd

D

register

file

Ra

Rb

ID/EX

ctrl

PC+4

imm

ID/EX

Rest of pipeline

dest
dest

mem/Acc result

decode

decode

extend

extend
Stage 1: Instruction Fetch

Rest of pipeline

ID/EX

ID

IF/ID

Instruction fetch

Register file

WE Rd D

decode

extend

result

dest
EX

Stage 3: Execute

On every cycle:

- Read ID/EX pipeline register to get values and control bits
- Perform ALU operation
- Compute targets (PC+4+offset, etc.) *in case* this is a branch
- Decide if jump/branch should be taken

Write values of interest to pipeline register (EX/MEM)

- Control information, Rd index, ...
- Result of ALU operation
- Value *in case* this is a memory store instruction
Stage 2: Instruction Decode

ID/EX

| ctrl | PC+4 | imm | B | A |

EX

alu

Rest of pipeline

| ctrl | target | B | D |

EX/MEM
Stage 2: Instruction Decode

- pcrel
- pcabs
- ctrl
- EX/MEM

Rest of pipeline

ID/EX

+ branch?

pcsel

alu

pcreg

pcreg

PC+4

imm

ctrl

ctrl

EX

EX/MEM

B

D

A

Rest of pipeline
MEM

Stage 4: Memory

On every cycle:

- Read EX/MEM pipeline register to get values and control bits
- Perform memory load/store if needed
  - address is ALU result

Write values of interest to pipeline register (MEM/WB)

- Control information, Rd index, ...
- Result of memory operation
- Pass result of ALU operation
Stage 3: Execute

MEM

Memory

addr

d_{in}
d_{out}
m_c

ALU Result
- Ra # Rb
- sw/cw

Rest of pipeline

EX/MEM

MEM/WB

ctrl

target

d

ctrl

D

B

M

D
Stage 3: Execute

psel

branch?

MEM

memory

addr

d_in
d_out

mc

ctrl

target
pcabs
pcsel

creg

Rest of pipeline

EX/MEM

MEM/WB
WB

Stage 5: Write-back

On every cycle:

• Read MEM/WB pipeline register to get values and control bits
• Select value and write to register file
Stage 4: Memory

ctrl

MEM/WB

WB
Stage 4: Memory

MEM/WB

result

dest

ctrl

M

D

WB
Pipelining Recap

Pipelining is a powerful technique to mask latencies and increase throughput

- Logically, instructions execute one at a time
- Physically, instructions execute in parallel
  - Instruction level parallelism

Abstraction promotes decoupling

- Interface (ISA) vs. implementation (Pipeline)