Performance and Pipelining

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

See P&H Chapter: 1.6, 4.5-4.6
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 MHz, $10^6$ Hz: cycle is 1 microsecond ($10^{-6}$)
- 1 Ghz, $10^9$ Hz: cycle is 1 nanosecond ($10^{-9}$)
- 1 Thz, $10^{12}$ Hz: cycle is 1 picosecond ($10^{-12}$)

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 delay 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  \hspace{1cm} ms = 10^{-3} \text{ second}
- arithmetic: 50 ns $\leftarrow$ 20 MHz  \hspace{1cm} us = 10^{-6} \text{ seconds}
- branches: 33 ns $\leftarrow$ 30 MHz  \hspace{1cm} ns = 10^{-9} \text{ seconds}

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 $\leftrightarrow$ 10 MHz
- arithmetic: 50 ns $\leftrightarrow$ 20 MHz
- branches: 33 ns $\leftrightarrow$ 30 MHz

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:

Multi-Cycle @ 30 MHz

Single-Cycle @ 10 MHz
Total Time

CPU Time = \# Instructions \times CPI \times Clock Cycle Time

= Instr \times cycles/instr \times seconds/cycle

E.g. Say for a program with 400k instructions, 30 MHz:
CPU [Execution] 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
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
The Kids

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
Elapsed Time for Alice: 4
Elapsed Time for Bob: 4
Total elapsed time: 4*N
Can we do better?

Latency:
Throughput:
Concurrency:

CPI =
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 ½ as long?

Latency:
Throughput:  
CPI =
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
A Processor

Instruction Fetch
- Memory
  - inst
- PC
- new pc

Instruction Decode
- register file
- control
- extend
- imm

Execute
- alu
- compute
  - jump/branch targets
- addr
- compute
  - d_out
  - memory

Write-Back
- new pc
- PC
- +4
- extend
- inst
Clock cycle

Latency: Throughput: Concurrency:
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
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)
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

IF/ID

PC+4

inst

ID

ID/EX

ctrl

PC+4

imm

Rest of pipeline

WE
Rd
D
register file
Ra
Rb

A
B
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

EX

alu

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

ctrl
target

EX/MEM

MEM

ctrl

Rest of pipeline

memory

addr

M

d_{in}
d_{out}

mc

D

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
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)