Performance and Pipelining

CS 3410, Spring 2014
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.
PA 1 design doc
  Critical to do this, else PA 1 will be hard

HW 1 review session
  Fri (2/21) and Sun (2/23). 7:30pm.
  Location: Olin 165

Prelim 1 review session
  Next Fri and Sun. 7:30pm. Location: TBA
**Control Flow: Absolute Jump**

```
00001010100001001000011000000011
```

<table>
<thead>
<tr>
<th>op</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2</td>
<td>J target</td>
<td>PC = (PC+4) • target • 00</td>
</tr>
</tbody>
</table>

Absolute addressing for jumps
- Jump from 0x3000000000000000 to 0x2000000000000000?
  - But: Jumps from 0x2FFFFFFFFc to 0x3xxxxxxxxx are possible, but not reverse
- Trade-off: out-of-region jumps vs. 32-bit instruction encoding

**MIPS Quirk:**
- Jump targets computed using *already incremented* PC
# Two’s Complement

<table>
<thead>
<tr>
<th>Non-negatives (as usual):</th>
<th>Negatives (two’s complement: flip then add 1):</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0 = 0000</td>
<td>flip = 1111, -0 = 0000</td>
</tr>
<tr>
<td>+1 = 0001</td>
<td>flip = 1110, -1 = 1111</td>
</tr>
<tr>
<td>+2 = 0010</td>
<td>flip = 1101, -2 = 1110</td>
</tr>
<tr>
<td>+3 = 0011</td>
<td>flip = 1100, -3 = 1101</td>
</tr>
<tr>
<td>+4 = 0100</td>
<td>flip = 1011, -4 = 1100</td>
</tr>
<tr>
<td>+5 = 0101</td>
<td>flip = 1010, -5 = 1011</td>
</tr>
<tr>
<td>+6 = 0110</td>
<td>flip = 1001, -6 = 1010</td>
</tr>
<tr>
<td>+7 = 0111</td>
<td>flip = 1000, -7 = 1001</td>
</tr>
<tr>
<td>+8 = 1000</td>
<td>flip = 0111, -8 = 1000</td>
</tr>
</tbody>
</table>
2’s complement

1101 (-3)

\[ -1101 \rightarrow -3 \]

\[ \frac{-1}{1100} \sim 0011 \rightarrow 3 \]

100101

\[-2^5 + 4 + 1 = -27\]

100100

\[011011 \rightarrow 1 + 2 + 8 + 16 = 27\]
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 cycle, 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
- 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
- arithmetic: 50 ns $\leftarrow 20$ MHz
- branches: 33 ns $\leftarrow 30$ MHz

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 ➔ 30M cycles/sec ÷ 2.0 cycles/instr ≈ 15 MIPS

Single-Cycle @ 10 MHz ➔ 10 MIPS

MIPS = millions of instructions per second
Total Time

CPU Time = # Instructions x CPI x Clock Cycle Time

Say for a program with 400k instructions, 30 MHz:
Time = 400k x 2.1 x 33 ns = 27 millisecs
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
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

First lets try CPI of 1 for arithmetic. Is that 2x faster overall? No

How much does it improve performance?

\[
0.25 \times 3 + 0.6 \times 1 + 0.15 \times 1
\]

\[
0.75 + 0.6 + 0.15 = 1.5
\]
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

\[
\begin{align*}
2 \times 1 \text{ CPI} & \rightarrow 2 \times 1 \times \frac{2}{2} \\
& = 1 \cdot 0.5 \\
3 \times 0.25 + \chi \times 0.6 \\
& + 0.15 \times 1 = 1.05 \\
\chi & = 0.25
\end{align*}
\]
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

To double performance CPI has to go from 2 to 0.25
Amdahl’s Law

Amdahl’s Law

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

1. Saw
2. Drill
3. Glue
4. 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:
Throughput:
Concurrency:

Can we do better?

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
Pipelined Performance

Latency:  
Throughput: 1  
Concurrency:
Latency: 4 cycles/task
Throughput: 1 task/2 cycles
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

Memory

InstrucEon

Fetch

Execute

InstrucEon

Decode

register

file

alu

imm

memory

din
dout

addr

inst

PC

new
pc

Instruction
Fetch

Control

compute
jump/branch
targets

extend

new pc

+4

3

4

5

Write-Back
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

IF/ID

ID/EX

EX/MEM

MEM/WB

Memory

Register File

Control

Extend

Instruction Decode

Jump/Branch Targets

Compute

Write-Back

Instruction Fetch

New PC

PC

Inst

New pc

Instruction

Fetch

InstrucEon

Fetch

InstrucEon

Decode

alu

Memory

d

din

dout

addr

PC

new

imm

B

A

compute

jump/branch

targets

ctrl

ctrl

ctrl

ctrl

Write-Back

inst

extend

control

Memory

inst

extend

control
IF

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)
instruction memory

addr mc

PC

new pc

IF
00 = read word

Instruction memory

PC

new pc

pcsel

if

Rest of pipeline
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)
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

pcrel

pcabs

EX/MEM

Rest of pipeline

pcreg

pcsel

branch?
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

pcsel

branch?

MEM

EX/MEM

MEM/ WB

pcreg

pcrel

pcabs

addr

d_{in}

memory

d_{out}

mc
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

MEM/WB

result

dest

WB
**Pipelining Recap**

Powerful technique for masking latencies

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

Abstraction promotes decoupling

- Interface (ISA) vs. implementation (Pipeline)