Pipelining and Hazards

CS 3410, Spring 2014
Computer Science
Cornell University

See P&H Chapter: 4.6-4.8
Announcements

Prelim next week
   Tuesday at 7:30.

   Go based on netid
Prelim reviews
   Friday and Sunday evening. 7:30 again.
   Location: TBA on piazza
Prelim conflicts
   Contact KB, Prof. Weatherspoon, Andrew Hirsch

Survey
   Constructive feedback is very welcome
Administrivia

Prelim1:

• Time: We will start at 7:30pm sharp, so come early
• Loc: Upson B17 [a-e]*, Olin 255[f-m]*, Philips 101 [n-z]*
• Closed Book
  • Cannot use electronic device or outside material
• Practice prelims are online in CMS

• Material covered everything up to end of this week
  • Everything up to and including data hazards
  • Appendix B (logic, gates, FSMs, memory, ALUs)
  • Chapter 4 (pipelined [and non] MIPS processor with hazards)
  • Chapters 2 (Numbers / Arithmetic, simple MIPS instructions)
  • Chapter 1 (Performance)
  • HW1, Lab0, Lab1, Lab2
Pipelining

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 (this and next lecture)
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
Pipelined Implementation

• Each instruction goes through the 5 stages
  • Each stage takes one clock cycle
    • So slowest stage determines clock cycle time
The pipeline achieves

A) Latency: 1, throughput: 1 instr/cycle
B) Latency: 5, throughput: 1 instr/cycle
C) Latency: 1, throughput: 1/5 instr/cycle
D) Latency: 5, throughput: 5 instr/cycle
E) None of the above
Latency: 5
Throughput: 1 instruction/cycle
Concurrency: 5
Pipelined Implementation

- Each instruction goes through the 5 stages
  - Each stage takes one clock cycle
    - So slowest stage determines clock cycle time

- Stages must share information. How?
  - Add pipeline registers (flip-flops) to pass results between different stages
Pipelined Implementation

- Each instruction goes through the 5 stages
  - Each stage takes one clock cycle
    - So slowest stage determines clock cycle time

- Stages must share information. How?
  - Add pipeline registers (flip-flops) to pass results between different stages

And is this it? Not quite....
Hazards

3 kinds

- Structural hazards
  - Multiple instructions want to use same unit

- Data hazards
  - Results of instruction needed before ready

- Control hazards
  - Don’t know which side of branch to take

Will get back to this
First, how to pipeline when no hazards
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)
00 = read word
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

IF/ID

decode

ID/EX

register file

WE
Rd
D

Ra
Rb

result
dest

extend

ID

rest of pipeline
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
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

- pcsel
- pcrel
- pcabs
- pcreg

Branch?

MEM

memory

addr

d

d_{out}

MC

Rest of pipeline

EX/MEM

MEM/WB
WB

Stage 5: Write-back

On every cycle:

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

MEM/WB

result

dest

WB
IF/ID

ID/EX

EX/MEM

MEM/WB

inst

mem

PC

Rd

D

B

Ra

Rb

OP

B

A

Rd

OP

Rd

OP

addr

d_{in}
d_{out}

mem
Example: Sample Code (Simple)

add   r3, r1, r2;
nand  r6, r4, r5;
lw    r4, 20(r2);
add   r5, r2, r5;
sw    r7, 12(r3);
Example: Sample Code (Simple)

Assume eight-register machine

Run the following code on a pipelined datapath

```
add  r3  r1  r2  ; reg 3 = reg 1 + reg 2
nand r6  r4  r5  ; reg 6 = ~(reg 4 & reg 5)
lw   r4  20 (r2) ; reg 4 = Mem[reg2+20]
add  r5  r2  r5  ; reg 5 = reg 2 + reg 5
sw   r7  12(r3) ; Mem[reg3+12] = reg 7
```

Slides thanks to Sally McKee
At time 1, Fetch

**add r3 r1 r2**
Fetch: add 3 1 2

Time: 1 2

IF/ID  ID/EX  EX/MEM  MEM/WB
Fetch:
sw 7 12(3)

Time: 5
IF/ID ID/EX EX/MEM MEM/WB

sw 7 12(3)
add 5 2 5 lw 4 20 (2) nand 6 4 5 add 3 1 2

PC Inst mem

MUX

Inst mem

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX
PC |
| Inst mem |
| 4 |

MUX |
| MUX |

PC |
| Inst mem |

No more instructions

IF/ID |
| ID/EX |
| EX/MEM |
| MEM/WB |

Time: 8

Slides thanks to Sally McKee
Takeaway

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)
Hazards

See P&H Chapter: 4.7-4.8
Hazards

3 kinds

• Structural hazards
  – Multiple instructions want to use same unit

• Data hazards
  – Results of instruction needed before

• Control hazards
  – Don’t know which side of branch to take
Data Hazards

What about data dependencies (also known as a data hazard in a pipelined processor)?

i.e. \(\text{add}(r3, r1, r2)\)

\(\text{sub}(r5, r3, r4)\)

Need to detect and then fix such hazards
Why do data hazards occur?

Data Hazards

• register file reads occur in stage 2 (ID)
• register file writes occur in stage 5 (WB)
• instruction may read (need) values that are being computed further down the pipeline
  – In fact this is quite common
add r3, r1, r2

sw r6, 12(r3)
add r3, r1, r2
sub r5, r3, r4
lw r6, 4(r3)
or r5, r3, r5
sw r6, 12(r3)

How many data hazards due to r3 only

A) 1
B) 2
C) 3
D) 4
E) 5
<2.	
  add	
  r3,	
  r1,	
  r2

3.	
  lw	
  r6,	
  4(r3)

4.	
  or	
  r5,	
  r3,	
  r5

5.	
  sw	
  r6,	
  12(r3)

OK

Data Hazards

Clock cycle
1  2  3  4  5  6  7  8  9

r3 = 10
1. add r3, r1, r2
r3 = 20
4. or r5, r3, r5

- IF
- ID
- MEM
- WB

- time

- lw(r6, 4(r3))
- or(r5, r3, r5)
- sw(r6, 12(r3))
- OK

- r3 = 10
- r3 = 20

- Clock cycle 4 is a hazard because r3 is required by the add instruction in the next cycle (clock cycle 5).
Data Hazards

What about data dependencies (also known as a data hazard in a pipelined processor)?

i.e. $\text{add}(r3, r1, r2)$

$\text{sub}(r5, r3, r4)$

How to detect?
detect hazard

for rA (IF/ID.rA ≠ 0 &&
  (IF/ID.rA==ID/Ex.Rd
  IF/ID.rA==Ex/M.Rd
  IF/ID.rA==M/W.Rd))
Detecting Data Hazards

Data Hazards

- register file reads occur in stage 2 (ID)
- register file writes occur in stage 5 (WB)
- next instructions may read values about to be written
  - In fact this is quite common

How to detect?  \((IF/ID.Ra \neq 0 \&\&
  (IF/ID.Ra == ID/EX.Rd ||
  IF/ID.Ra == EX/M.Rd ||
  IF/ID.Ra == M/WB.Rd))
  || (same for Rb)\)
Next Goal

• What to do if data hazard detected?

• Options
  • Nothing
  • Change the ISA to match implementation
  • Stall
  • Pause current and subsequent instructions till safe
  • Forward/bypass
    • Forward data value to where it is needed
Stalling

How to stall an instruction in ID stage

• prevent IF/ID pipeline register update
  – stalls the ID stage instruction

• convert ID stage instr into nop for later stages
  – innocuous “bubble” passes through pipeline

• prevent PC update
  – stalls the next (IF stage) instruction
<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>IF</th>
<th>ID</th>
<th>Ex</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>add r3, r1, r2</td>
<td>r3 = 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>sub r5, r3, r5</td>
<td>r3 = 20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>or r6, r3, r4</td>
<td>3 Stalls</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>add r6, r3, r8</td>
<td>3 Stalls</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Stalling**

- **IF** stage: Instruction Fetch
- **ID** stage: Instruction Decode
- **Ex** stage: Execution
- **M** stage: Memory Access
- **W** stage: Write Back

Stalls: 3 Stalls
Detecting Data Hazards

IF/ID
- IF/ID
  - WE=0
  - RegWr=0

ID/EX
- ID/EX
  - detect hazard
  - MemWr=0

EX/MEM
- EX/MEM
  - MemWr=0
  - RegWr=0

MEM/WB
- MEM/WB
  - MemWr=0

opcodes:
- add r3, r1, r2
- sub r5, r3, r5
- or r6, r3, r4
- add r6, r3, r8

PC
- PC
  - PC+4

inst
- inst
  - Ra, Rb
  - Rd, A, D

detect hazard
- detect hazard
  - if detect hazard

OP
- OP
  - Rd
  - Rt
  - imm

loop back
- loop back
NOP = \text{If}(\text{IF/ID.rA} \neq 0 \text{ && (IF/ID.rA} = \text{ID/Ex.Rd}} \\
\text{IF/ID.rA} = \text{Ex/M.Rd} \\
\text{IF/ID.rA} = \text{M/W.Rd}))

\begin{align*}
\text{add} & \ r3, r1, r2 \\
\text{sub} & \ r5, r3, r5 \\
\text{nop} & \\
\text{inst mem} & \rightarrow \text{inst mem} \\
\text{PC} & \\
\text{or} r6 & (r3, r4) \rightarrow \text{inst mem} \\
\text{inst mem} & \rightarrow \text{inst mem}
\end{align*}
NOP = If(IF/ID.rA ≠ 0 &&
             (IF/ID.rA==ID/Ex.Rd
              IF/ID.rA==Ex/M.Rd
              IF/ID.rA==M/W.Rd))

or r6,r3,r4

sub r5,r3,r5

(MemWr=0 RegWr=0)

op

(MemWr=0 RegWr=0)

add r3,r1,r2

or r6,r3,r4

sub r5,r3,r5

(MemWr=0 RegWr=0)

op

(MemWr=0 RegWr=0)
NOP = \text{If(\text{IF/ID.rA} \neq 0 \&\& (\text{IF/ID.rA}==\text{ID/Ex.Rd}) \text{IF/ID.rA}==\text{Ex/M.Rd})} \\
(M/M\text{W.Rd}))
Stalling

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>IF</th>
<th>ID</th>
<th>Ex</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

r3 = 10  
add r3, r1, r2

r3 = 20

sub r5, r3, r5

3 Stalls

or r6, r3, r4

add r6, r3, r8

Stalls 3
Stalling

How to stall an instruction in ID stage

• prevent IF/ID pipeline register update
  – stalls the ID stage instruction

• convert ID stage instr into nop for later stages
  – innocuous “bubble” passes through pipeline

• prevent PC update
  – stalls the next (IF stage) instruction
Data hazards occur when an operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.

Stalling, preventing a dependent instruction from advancing, is one way to resolve data hazards.

Stalling introduces NOPs (“bubbles”) into a pipeline. Introduce NOPs by (1) preventing the PC from updating, (2) preventing writes to IF/ID registers from changing, and (3) preventing writes to memory and register file. Bubbles in pipeline significantly decrease performance.