ECE/CS 552: Pipelining
Instructor: Mikko H Lipasti
Fall 2010
University of Wisconsin-Madison
Lecture notes based on set created by Mark Hill and John P. Shen
Updated by Mikko Lipasti

Motivation
- Single cycle implementation
  - CPI = 1
  - Cycle = imem + RFrd + ALU + dmem + RFwr + muxes + control
  - E.g. 500+250+500+500+250+0+0 = 2000ps
  - Time/program = P x 2ns

Muticycle
- Muticycle implementation:
  - CPI = 3, 4, 5
  - Cycle = max(memory, RF, ALU, mux, control)
  - Time/program = P x 4 x 500 = P x 2000ps = P x 2ns
  - Would like:
    - CPI = 1 + overhead from hazards (later)
    - Cycle = 500ps + overhead
    - In practice, ~3x improvement

Pipelining
- Forecast
  - Big Picture
  - Datapath
  - Control
  - Data Hazards
    - Stalls
    - Forwarding
  - Control Hazards
  - Exceptions

Multicycle Implementation

Big Picture
- Instruction latency = 5 cycles
- Instruction throughput = 1/5 instr/cycle
- CPI = 5 cycles per instruction
- Instead
  - Pipeline: process instructions like a lunch buffet
  - ALL microprocessors use it
    - E.g. Core i7, AMD Barcelona, ARM11
Big Picture

- Instruction Latency = 5 cycles (same)
- Instruction throughput = 1 instr/cycle
- CPI = 1 cycle per instruction
- CPI = cycle between instruction completion = 1

Ideal Pipelining

- Bandwidth increases linearly with pipeline depth
- Latency increases by latch delays

Example: Integer Multiplier

- 16x16 combinational multiplier
- ISCAS-85 C6288 standard benchmark
- Tools: Synopsys DC/LSI Logic 110nm gflxp ASIC

Example: Integer Multiplier

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Delay</th>
<th>MPS</th>
<th>Area (FF/wiring)</th>
<th>Area Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinational</td>
<td>3.52ns</td>
<td>284</td>
<td>7535 (~1759)</td>
<td></td>
</tr>
<tr>
<td>2 Stages</td>
<td>1.87ns</td>
<td>534</td>
<td>8725 (1078/1870)</td>
<td>16%</td>
</tr>
<tr>
<td>4 Stages</td>
<td>1.17ns</td>
<td>855</td>
<td>11227 (3388/2312)</td>
<td>50%</td>
</tr>
<tr>
<td>8 Stages</td>
<td>0.80ns</td>
<td>1250(4.46)</td>
<td>17127 (4038/2612)</td>
<td>127%</td>
</tr>
</tbody>
</table>

Pipelining Idealisms

- Uniform subcomputations
  - Can pipeline into stages with equal delay
- Identical computations
  - Can fill pipeline with identical work
- Independent computations
  - No relationships between work units
- Are these practical?
  - No, but can get close enough to get significant speedup
Complications
- Datapath
  - Five (or more) instructions in flight
- Control
  - Must correspond to multiple instructions
- Instructions may have
  - data and control flow dependences
  - I.e. units of work are not independent
    - One may have to stall and wait for another

Datapath
- Datapath
  - Set by 5 different instructions
  - Divide and conquer: carry IR down the pipe
- MIPS ISA requires the appearance of sequential execution
  - Precise exceptions
  - True of most general purpose ISAs

Control
- Control
  - Precise exceptions
  - True of most general purpose ISAs

Program Dependences
- A true dependence between two instructions may only involve one subcomputation of each instruction.
- The implied sequential precedences are an overspecification. It is sufficient but not necessary to ensure program correctness.

Program Data Dependences
- True dependence (RAW)
  - j cannot execute until i produces its result
- Anti-dependence (WAR)
  - j cannot write its result until i has read its sources
- Output dependence (WAW)
  - j cannot write its result until i has written its result
Control Dependences

- Conditional branches
  - Branch must execute to determine which instruction to fetch next
  - Instructions following a conditional branch are control dependent on the branch instruction

Resolution of Pipeline Hazards

- Pipeline hazards
  - Potential violations of program dependences
  - Must ensure program dependences are not violated
- Hazard resolution
  - Static: compiler/programmer guarantees correctness
  - Dynamic: hardware performs checks at runtime
- Pipeline interlock
  - Hardware mechanism for dynamic hazard resolution
  - Must detect and enforce dependences at runtime

Pipeline Hazard Analysis

- Memory hazards
  - RAW: Yes/No?
  - WAR: Yes/No?
  - WAW: Yes/No?
- Register hazards
  - RAW: Yes/No?
  - WAR: Yes/No?
  - WAW: Yes/No?

RAW Hazard

- Earlier instruction produces a value used by a later instruction:
  - add $1, $2, $3
  - sub $4, $5, $1

Example (quicksort/MIPS)

- Necessary conditions:
  - WAR: write stage earlier than read stage
  - WAW: write stage earlier than write stage
  - RAW: read stage earlier than write stage

- If conditions not met, no need to resolve
- Check for both register and memory

Pipeline Hazards

- For conditions not met, no need to resolve
- Check for both register and memory
RAW Hazard - Stall

- Detect dependence and stall:
  - add $1, $2, $3
  - sub $4, $5, $1

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instr</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>F</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Control Dependence

- One instruction affects which executes next
  - sw $4, 0($5)
  - bne $2, $3, loop
  - sub $6, $7, $8

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instr</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Control Dependence - Stall

- Detect dependence and stall
  - sw $4, 0($5)
  - bne $2, $3, loop
  - sub $6, $7, $8

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instr</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipelined Datapath

- Start with single-cycle datapath
- Pipelined execution
  - Assume each instruction has its own datapath
  - But each instruction uses a different part in every cycle
  - Multiplex all on to one datapath
  - Latches separate cycles (like multicycle)
- Ignore hazards for now
  - Data
  - Control

Pipelined Datapath

- Instruction flow
  - add and load
  - Write of registers
  - Pass register specifiers
- Any info needed by a later stage gets passed down the pipeline
  - E.g. store value through EX
Pipelined Control

- IF and ID
  - None
- EX
  - ALUop, ALUsrc, RegDst
- MEM
  - Branch, MemRead, MemWrite
- WB
  - MempToReg, RegWrite

Datapath Control Signals

Pipelining

- Controlled by different instructions
- Decode instructions and pass the signals down the pipe
- Control sequencing is embedded in the pipeline
  - No explicit FSM
  - Instead, distributed FSM

Datapath Control Signals

All Together

Pipelining

- Not too complex yet
  - Data hazards
  - Control hazards
  - Exceptions
RAW Hazards

- Must first detect RAW hazards
  - Pipeline analysis proves that WAR/WAW don’t occur

ID/EX.WriteRegister = IF/ID.ReadRegister1
ID/EX.WriteRegister = IF/ID.ReadRegister2
EX/MEM.WriteRegister = IF/ID.ReadRegister1
EX/MEM.WriteRegister = IF/ID.ReadRegister2
MEM/WB.WriteRegister = IF/ID.ReadRegister1
MEM/WB.WriteRegister = IF/ID.ReadRegister2

RAW Hazards

- Not all hazards because
  - WriteRegister not used (e.g. sw)
  - ReadRegister not used (e.g. addi, jump)
  - Do something only if necessary

RAW Hazard Forwarding

- A better response – forwarding
  - Also called bypassing
- Comparators ensure register is read after it is written
  - Instead of stalling until write occurs
    - Use mux to select forwarded value rather than register value
    - Control mux with hazard detection logic

Forwarding Paths (ALU instructions)

- Forwarding via Path a
  - i+1 writes R1 before i+3 reads R1
- Forwarding via Path b
  - i+1 reads R1 before i+3 writes R1

Write before Read RF

- Register file design
  - 2-phase clocks common
  - Write RF on first phase
  - Read RF on second phase
- Hence, same cycle:
  - Write $1
  - Read $1
- No bypass needed
  - If read before write or DFF-based, need bypass
ALU Forwarding

Forwarding Paths (Load instructions)

Implementation of Load Forwarding

Control Flow Hazards

Control Flow Hazards

Control Flow Hazards

What to do?
– Always stall
– Easy to implement
– Performs poorly
– 1/6th instructions are branches, each branch takes 3 cycles
– CPI = 1 + 3 x 1/6 = 1.5 (lower bound)

Predict branch not taken
– Send sequential instructions down pipeline
– Kill instructions later if incorrect
– Must stop memory accesses and RF writes
– Late flush of instructions on misprediction
  – Complex
  – Global signal (wire delay)
Control Flow Hazards

- Even better but more complex
  - Predict taken
  - Predict both (eager execution)
  - Predict one or the other dynamically
    - Adapt to program branch patterns
    - Lots of chip real estate these days
      - Pentium III, 4, Alpha 21264
      - Current research topic
    - More later (lecture on branch prediction)
- Adapt to program branch patterns
- Lots of chip real estate these days
  - Pentium III, 4, Alpha 21264
  - Current research topic
- More later (lecture on branch prediction)

Exceptions and Pipelining

- add $1, $2, $3 overflows
- A surprise branch
  - Earlier instructions flow to completion
  - Kill later instructions
  - Save PC in EPC, set PC to EX handler, etc.
- Costs a lot of designer sanity
  - 554 teams that try this sometimes fail

Exceptions

- Even worse: in one cycle
  - I/O interrupt
  - User trap to OS (EX)
  - Illegal instruction (ID)
  - Arithmetic overflow
  - Hardware error
  - Etc.
- Interrupt priorities must be supported

Review

- Big Picture
- Datapath
- Control
  - Data hazards
    - Stalls
    - Forwarding or bypassing
  - Control flow hazards
    - Branch prediction
- Exceptions