Overview

- General Principles of Pipelining
  - Goal
  - Difficulties
- Creating a Pipelined x86-64 Processor
  - Rearranging SEQ
  - Inserting pipeline registers
  - Problems with data and control hazards

Real-World Pipelines: Car Washes

- Idea
  - Divide process into independent stages
  - Move objects through stages in sequence
  - At any given times, multiple objects being processed

Computational Example

- System
  - Computation requires total of 300 picoseconds
  - Additional 20 picoseconds to save result in register
  - Must have clock cycle of at least 320 ps

3-Way Pipelined Version

- System
  - Divide combinational logic into 3 blocks of 100 ps each
  - Can begin new operation as soon as previous one passes through stage A.
    - Begin new operation every 120 ps
  - Overall latency increases
    - 360 ps from start to finish

Pipeline Diagrams

- Unpipelined
- Cannot start new operation until previous one completes
- 3-Way Pipelined
  - Up to 3 operations in process simultaneously
Data Hazards

- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system

Limitations: Nonuniform Delays

- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Data Dependencies

- System
  - Each operation depends on result from preceding one

Data Dependencies in Processors

- Result from one instruction used as operand for another
  - Read-after-write (RAW) dependency
  - Very common in actual programs
  - Must make sure our pipeline handles these properly
    - Get correct results
    - Minimize performance impact
SEQ Hardware
- Stages occur in sequence
- One operation in process at a time

SEQ+ Hardware
- Still sequential implementation
- Reorder PC stage to put at beginning
- PC Stage
  - Task is to select PC for current instruction
  - Based on results computed by previous instruction
- Processor State
  - PC is no longer stored in register
  - But, can determine PC based on other stored information

Adding Pipeline Registers
- Pipeline stages
  - Fetch
    - Select current PC
    - Read instruction
    - Compute incremented PC
  - Decode
    - Read program registers
  - Execute
    - Operate ALU
  - Memory
    - Read or write data memory
  - Write Back
    - Update register file

PIPE- Hardware
- Pipeline registers hold intermediate values from instruction execution
- Forward (Upward) Paths
  - Values passed from one stage to next
  - Cannot jump past stages
    - e.g., valC passes through decode

Signal Naming Conventions
- S_Field
  - Value of Field held in stage S pipeline register
- s_Field
  - Value of Field computed in stage S
Feedback
Paths

• Predicted PC
  • Guess value of next PC
• Branch information
  • Jump taken/not-taken
  • Fall-through or target address
• Return point
  • Read from memory
• Register updates
  • To register file write ports

Predicting the PC

• Start fetch of new instruction after current one has completed fetch stage
  • Not enough time to reliably determine next instruction
• Guess which instruction will follow
  • Recover if prediction was incorrect

Our Prediction Strategy

• Instructions that Don’t Transfer Control
  • Predict next PC to be valP
  • Always reliable
• Call and Unconditional Jumps
  • Predict next PC to be valC (destination)
  • Always reliable
• Conditional Jumps
  • Predict next PC to be valC (destination)
  • Only correct if branch is taken
    • Typically right 60% of time
• Return Instruction
  • Don’t try to predict

Recovering from PC Misprediction

• Mispredicted Jump
  • Will see branch condition flag once instruction reaches memory stage
  • Can get fall-through PC from valA (value M_valA)
• Return Instruction
  • Will get return PC when ret reaches write-back stage (W_valM)

Pipeline Demonstration

• File: demo-basic.ys

Data Dependencies: 3 Nop’s

• File: demo-h3.ys
Data Dependencies: 2 Nop's

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000:</td>
<td>movq 10, rdx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00a:</td>
<td>movq $3, rax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x014:</td>
<td>movq 2, rdx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x015:</td>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x016:</td>
<td>addq rdx, rdx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x018:</td>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cycle 6

- Misprediction:
  - Trace
  - Data Dependencies:
    - No
    - Nop

- Incorrectly execute two instructions at branch target

- Require lots of nops to avoid data hazards

Branch Misprediction Example

data=j.ys

- Should only execute first 8 instructions

Return Example
data=ret.ys

- Initialize stack pointer
- Avoid hazard on trap
- Procedure call
- Procedure
- Initial stack pointer
Incorrect Return Example

- Incorrectly execute 3 instructions following `ret`

Pipeline Summary

- Concept
  - Break instruction execution into 5 stages
  - Run instructions through in pipelined mode
- Limitations
  - Can’t handle dependencies between instructions when instructions follow too closely
  - Data dependencies
    - One instruction writes register, later one reads it
  - Control dependency
    - Instruction sets PC in way that pipeline did not predict correctly
    - Mispredicted branch and return
- Fixing the Pipeline
  - We’ll do that next