Overview

General Principles of Pipelining
- Goal
- Difficulties

Creating a Pipelined Y86-64 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

Pipeline Diagrams

Unpipelined
- Cannot start new operation until previous one completes

3-Way Pipelined
- Up to 3 operations in process simultaneously
Operating a Pipeline

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

Limitations: Register Overhead
- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

Data Dependencies
- Each operation depends on result from preceding one
- Pipelining has changed behavior of system

Data Dependencies in Processors
- Read-after-write (RAW) dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact
Exercise Break: Instruction Stages

- Fetch
  - valA ← R[sp]
  - valM ← M[valA]
  - PC ← valM

- Decode
  - valB ← R[sp]
  - valE ← valB + 8

- Execute
  - valP ← PC + 1
  -icode:ifun ← M[PC]
  - R[sp] ← valE

- Memory
  - valA ← R[sp]

- Write-back
  - valB ← R[sp]

- PC update
  - PC ← valE

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 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

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
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 xrt reaches write stage (W, valM)

Pipeline Demonstration

```
irmovq $1,%rax #I1
irmovq $2,%rcx #I2
irmovq $3,%rdx #I3
irmovq $4,%rbx #I4
halt               #I5
```

File: demo-basic.ys
Branch Misprediction Example

demo-j.py

0x000: movq trx, rtr
0x002: jae t # Not taken
0x00b: imovq $1, rtr # Fall through
0x015: nop
0x016: nop
0x017: halt
0x019: t: imovq $3, rtrx # Target (Should not execute)
0x023: imovq $4, rtrx # Should not execute
0x02d: imovq $5, rtrx # Should not execute

- Should only execute first 8 instructions

Branch Misprediction Trace

demo-j.py

0x000: movq trx, rtr
0x002: jae t # Not taken
0x00b: imovq $1, rtr # Fall through
0x015: nop
0x016: nop
0x017: halt
0x019: t: imovq $3, rtrx # Target (Should not execute)
0x023: imovq $4, rtrx # Should not execute
0x02d: imovq $5, rtrx # Should not execute

- Incorrectly execute two instructions at branch target
Return Example

demo-ret.ya

0x00: irmovq Stack,%esp # Initial stack pointer
0x00a: nop # Avoid hazard on %esp
0x00b: nop
0x00c: nop
0x00d: call p # Procedure call
0x016: irmovq $5,%rsi # Return point
0x020: halt
0x020: .pos 0x20
0x020: p: nop # procedure
0x021: nop
0x022: nop
0x023: ret
0x024: irmovq $1,%rax # Should not be executed
0x025: irmovq $2,%rcx # Should not be executed
0x028: irmovq $3,%rdx # Should not be executed
0x02b: irmovq $4,%rbx # Should not be executed
0x100: .pos 0x100
0x100: Stack: # Initial stack pointer

• Require lots of nops to avoid data hazards

Incorrect Return Example

demo-ret

0x00: ret
0x01a: irmovl $3,%ecx # Oops!
0x01a: irmovl $3,%ecx # Oops!
0x01a: irmovl $3,%ecx # Return

• Incorrectly execute 3 instructions following ret

Fixing the Pipeline

• Stalling: make later stages wait until data is available
  • Insert fake instructions called “bubbles” in pipeline
  • Always possible, but can waste a lot of time
  • Used for PC after ret, and data loads

• Forwarding: add extra wires to make data available sooner
  • E.g., “bypass path” from e_valE to d_valA bypassing register file
  • Requires more complex control logic

• Branch prediction
  • Guess (e.g.) that branches will always be taken
  • If guess is wrong, mis-predicted instructions turn into bubbles

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
• Textbook gives more details of fixing techniques