Pipelined Implementation of Y86 (1)

CSci 2021: Machine Architecture and Organization
Lecture #20, March 9th, 2015
Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O’Hallaron, Antonia Zhai

Overview

General Principles of Pipelining
- Goal
- Difficulties

Creating a Pipelined Y86 Processor
- Rearranging SEQ
- Inserting pipeline registers
- Problems with data and control hazards

Exercise Break: Instruction Stages

Fetch
valA ← R[%esp]
valM ← Mₐ(valA)
PC ← valM

Decode
valB ← R[%esp]
valE ← valB + 4
valIP ← PC + 1
icode:ifun ← Mₐ(PC)

Execute
R[%esp] ← valE

Memory
PC ← valM

Write-back
valM ← M₄[valA]

Write-back
PC update
valP ← PC + 1

What instruction is this?
ret

Real-World Pipelines: Car Washes

Sequential
Parallel

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

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

Data Dependencies

- Each operation depends on result from preceding one

Data Hazards

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

---

Quiz 1 Statistics

N = 211 (both sections)
Mean = 74.3
Median = 77
Standard deviation = 19.41

---

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 valIP
- 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 66% 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 ref. reaches write-back stage (`W_valM`)
Pipeline Demonstration

File: demo-basic.ys

Data Dependencies: No Nops

Data Dependencies: 2 Nops

Data Dependencies: 1 Nop

Branch Misprediction Example

demo-j.ys

Should only execute first 8 instructions
Branch Misprediction Trace

Incorrectly execute two instructions at branch target

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 time

Return Example

Incorrect Return Example

Requires lots of nops to avoid data hazards