Overview

General Principles of Pipelining
- Goal
- Difficulties

Creating a Pipelined V86-64 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
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

Data Hazards

- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system
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., \( \text{valC} \) passes through decode

Signal Naming Conventions
- \( S \_\text{Field} \): Value of Field held in stage \( S \) pipeline register
- \( s \_\text{Field} \): 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 wbc reaches write-back stage (W_valM)

Pipeline Demonstration

Data Dependencies: 3 Nop’s
Incorrectly execute two instructions at branch target

Branch Misprediction Example

data-j .ys

* Should only execute first 8 instructions

Branch Misprediction Trace

data-j .ys

* Incorrectly execute two instructions at branch target

Return Example

data-ret .ys

* Require lots of nops to avoid data hazards
**Incorrect Return Example**

- Incorrectly execute 3 instructions following `ret`

```plaintext
0x023:    ret       W
0x024:    irmovl $1,%rax # Oops!  F
0x02a:    irmovl $2,%rcx # Oops!  F
0x030:    irmovl $3,%rdx # Oops!  F
0x00e:    irmovl $5,%rsi # Return  F
```

**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