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

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

Limitations: Register Overhead

Data Dependencies

Data Hazards

Data Dependencies in Processors
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 write-back stage ($W_{valM}$)

Pipeline Demonstration

| $i$
| $\text{irmovq} \%rax$
| $\text{irmovq} \%rcx$
| $\text{irmovq} \%rdx$
| $\text{irmovq} \%rbx$
| $\text{halt}$

Data Dependencies: 3 Nop’s

- $\text{addq} \%rdx, \%rax$
- $\text{nop}$
- $\text{nop}$
- $\text{nop}$
- $\text{halt}$

File: demo-basic.ys

Cycle 6

- $\text{valA} = M_{\text{valA}} = 3$
- $\text{valB} = \text{ret}_{\text{valB}} = 3$

Cycle 7

- $\text{valA} = W_{\text{valM}} = 3$
- $\text{valB} = W_{\text{valM}} = 3$
Data Dependencies: 2 Nop's

```
0x000: irmovq $18, rdx
0x004: irmovq $3, rax
0x014: nop
0x015: nop
0x016: addq rdx, rax
0x018: halt
```

Cycle 6

```
W
M
W
D
```

# Target+1

```
W
M
W
D
```

 Should only execute first 8 instructions

```
W
M
W
D
```

Branch Misprediction Example

demo-j.ys

```
0x000: xorq rax, rax # Not taken
0x002: jne t # Fall through
0x00b: irmovq $1, rax
0x015: nop
0x016: nop
0x017: nop
0x018: halt
0x019: t: irmovq $3, rdx # Target (Should not execute)
0x023: irmovq $4, rdx # Should not execute
0x02d: irmovq $5, rdx # Should not execute
```

Should only execute first 8 instructions

```
W
M
W
D
```

Return Example
demo-ret.ys

```
0x000: irmovq Stack, resp # Initialize stack pointer
0x00a: nop # Avoid hazard on r esp
0x00b: nop
0x00c: nop
0x00d: call p # Procedure call
0x014: 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
0x026: irmovq $2, rdx # Should not be executed
0x038: irmovq $3, rdx # Should not be executed
0x042: irmovq $4, rdx # Should not be executed
0x100: .pos 0x100
0x100: Stack: # Initial stack pointer
```

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

- Incorrectly execute 3 instructions following `ret`.

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

**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