Overview

Make the pipelined processor work!

Data Hazards
- Instruction having register R as source follows shortly after instruction having register R as destination
- Common condition, don't want to slow down pipeline

Control Hazards
- Mispredict conditional branch
  - Our design predicts all branches as being taken
  - Naive pipeline executes two extra instructions
- Getting return address for ret instruction
  - Naive pipeline executes three extra instructions

Making Sure It Really Works
- What if multiple special cases happen simultaneously?

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

Data Dependencies: 2 Nop’s
```
# demo-h2.ys
0x000: irmovq $10, %rdx
0x005: irmovq $3, %rax
0x014: nop
0x015: nop
0x016: addq %rdx, %rax
0x018: halt
```

Data Dependencies: No Nop
```
# demo-h0.ys
0x000: irmovq $10, %rdx
0x005: irmovq $3, %rax
0x014: addq %rdx, %rax
0x016: halt
```

Cycle 6
- N[%rdx] = 3
- N[%rax] = 0
- Error

Cycle 4
- N[%rdx] = 0
- N[%rax] = 10
- Error
Stalling for Data Dependencies

- Like dynamically generated nop's
- Stalling instruction held back in decode stage
- Following instruction stays in fetch stage
- Bubbles injected into execute stage
- Like dynamically generated nop's
- Move through later stages

Stalling X3

- Cycle 4
  - W, dstM = trax
  - M, dstE = trax
  - Write Back: bubble
  - Memory: bubble
  - Decode: addq trds, trax
  - Fetch: halt

- Cycle 5
  - W, dstM = trax
  - M, dstE = trax
  - Write Back: bubble
  - Memory: bubble
  - Decode: addq trds, trax
  - Fetch: halt

- Cycle 6
  - W, dstM = trax
  - W, valE = 3
  - dstM = trax
  - dstE = trax

Implementing Stalling

- Combinational logic detects stall condition
- Sets mode signals for how pipeline registers should update

Detecting Stall Condition

- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode
- Dynamically inject nop into execute stage

Special Case

- Don’t stall for register ID 15 (0xF)
- Indicates absence of register operand
- Or failed cond. move

Stall Condition

Source Registers
- srcA and srcB of current instruction in decode stage
- Instructions in execute, memory, and write-back stages

Destination Registers
- dstE and dstM fields

What Happens When Stalling?

- Stalling instruction held back in decode stage
- Following instruction stays in fetch stage
- Bubbles injected into execute stage
- Like dynamically generated nop's
- Move through later stages
Implementing Forwarding

- Add additional feedback paths from E, M, and W pipeline registers into decode stage
- Create logic blocks to select from multiple sources for valA and valB in decode stage

Limitation of Forwarding

- Value needed by end of decode stage in cycle 7
- Value read from memory in memory stage of cycle 8

Avoiding Load/Use Hazard

- Stall using instruction for one cycle
- Can then pick up loaded value by forwarding from memory stage

Detecting Load/Use Hazard

- Stall instructions in fetch and decode stages
- Inject bubble into execute stage

Control for Load/Use Hazard
Branch Misprediction Example

```
0x000:  xorq %rax,%rax    # Not taken
0x002:  jne t            # Not taken
0x00b:  irmovq $1, %rax  # Fall through
0x015:  nop
0x016:  nop
0x017:  nop
0x018:  halt
0x019:  t: irmovq $3, %rdx # Target
0x020:  irmovq $3, %rbx  # Target
0x023:  irmovq $4, %r cx # Should not execute
0x02d:  irmovq $5, %rdx  # Should not execute
```

- Should only execute first 8 instructions

Detecting Mispredicted Branch

```
Condition | Trigger
---|---
Mispredicted Branch | E_icode = IJXX & !e_Cnd
```

Handling Misprediction

```
# demo-j.ys
0x000:  xorq %rax,%rax    F D E M W
0x002:  jne target # Not taken F D E M W
0x016:  t: irmovq $2,%rdx # Target 1 bubble F D E M W
0x020:  irmovq $3,%rbx # Target+1 bubble F D E M W
0x02b:  irmovq $1, %rax # Fall through F D E M W
0x035:  nop

Predict branch as taken
- Fetch 2 instructions at target
- Cancel when mispredicted
- Detect branch not taken in execute stage
- On following cycle, replace instructions in execute and decode by bubbles
- No side effects have occurred yet
```

Control for Misprediction

```
# demo-j.ys
0x000:  xorq %rax,%rax    F D E M W
0x002:  jne target # Not taken F D E M W
0x016:  t: irmovq $2,%rdx # Target 1 bubble F D E M W
0x020:  irmovq $3,%rbx # Target+1 bubble F D E M W
0x02b:  irmovq $1, %rax # Fall through F D E M W
0x035:  nop

Condition | F | D | E | M | W
---|---|---|---|---|---
Mispredicted Branch | normal | bubble | bubble | normal | normal
```

Return Example

```
0x000:  irmovq %rsp, %esp    # Initialize stack pointer
0x00a:  call p                # Procedure call
0x013:  irmovq $5,%r si       # Return point
0x01d:  halt
0x020:  pos 0xa20
0x022:  irmovq $-1,%rdi      # procedure
0x026:  ret
0x035:  irmovq $2,%rcx       # Should not be executed
0x03f:  irmovq $3,%rcx       # Should not be executed
0x049:  irmovq $4,%rdx       # Should not be executed
0x05:   pos 0xa10
0x100:  pos 0xa10
0x100:  Stack:    # Stack: Stack pointer
```

- Previously executed three additional instructions

Correct Return Example

```
# demo-rath.ys
0x000:  irmovq %esp,%esp      # Initialize stack pointer
0x00a:  call p                # Procedure call
0x013:  irmovq $5,%rdi       # Return

Condition | F | D | E | M | W
---|---|---|---|---|---
Mispredicted Branch | bubble | bubble | bubble | normal | normal
```

- As ret passes through pipeline, stall at fetch stage
- While in decode, execute, and memory stage
- Inject bubble into decode stage
- Release stall when reach write-back stage
Detecting Return

Control for Return

Special Control Cases

Implementing Pipeline Control

Initial Version of Pipeline Control

Control Combinations
Control Combination A

- Should handle as mispredicted branch
- Stalls F pipeline register
- But PC selection logic will be using M_valM anyhow

Control Combination B

- Would attempt to bubble and stall pipeline register D
- Signaled by processor as pipeline error

Handling Control Combination B

- Load/use hazard should get priority
- ret instruction should be held in decode stage for additional cycle

Corrected Pipeline Control Logic

```c
bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Stalling at fetch while ret passes through pipeline
    IRET in {D_icode, E_icode, M_icode} 
    # but not condition for a load/use hazard
    && (!E_icode in {IMMOVQ, IDPOPQ})
    && E_dstM in {d_srcA, d_srcB});
```

Pipeline Summary

Data Hazards
- Most handled by forwarding
  - No performance penalty
- Load/use hazard requires one cycle stall

Control Hazards
- Cancel instructions when detect mispredicted branch
  - Two clock cycles wasted
- Stall fetch stage while ret passes through pipeline
  - Three clock cycles wasted

Control Combinations
- Must analyze carefully
- First version had subtle bug
  - Only arises with unusual instruction combination