Review: 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

Review: 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

Overview

- 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
  - Naïve pipeline executes two extra instructions
  - Getting return address for ret instruction
  - Naïve pipeline executes three extra instructions

Making Sure It Really Works!!

- Need to handle multiple special cases happen simultaneously.
Stalling for Data Dependencies

What Happens When Stalling?

Stall Condition

Detecting Stall Condition

Stalling x 3

Implementing Stalling
Pipeline Register Modes

- Normal
- Stall
- Bubble

Data Forwarding

- Naive Pipeline
  - Register isn’t written until completion of write-back stage
  - Source operands read from register file in decode stage
    - Needs to be in register file at start of stage
- Observation
  - Value generated in execute stage or memory stage
- Trick
  - Pass value directly from generating instruction to decode stage
  - Needs to be available at end of decode stage

Data Forwarding Example

- instrmovq in write-back stage
- Destination value in W pipeline register
- Forward as valB for decode stage

Data Forwarding Example #2

- Register %rax
  - Generated by ALU during previous cycle
  - Forward from memory as valA
- Register %rax
  - Value just generated by ALU
  - Forward from execute as valB

Bypass Paths

- Decode Stage
  - Forwarding logic selects valA and valB
  - Normally from register file
  - Forwarding: get valA or valB from later pipeline stage
- Forwarding Sources
  - Execute: valE
  - Memory: valE, valM
  - Write back: valE, valM

Forwarding Priority

- Multiple Forwarding Choices
  - Which one should have priority
  - Match serial semantics
  - Use matching value from earliest pipeline stage
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

- Load-use dependency
  - 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

Control for Load/Use Hazard

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoadUse Hazard</td>
<td>stall</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>
Branch Misprediction Example

demo-j.ys
0x000: xorq %rax,%rax # Not taken
0x002: jne t # Fall through
0x007: nop
0x009: t: imovq $1, %rdx # Target
0x00b: imovq $2, %rcx # Should not execute
0x00d: imovq $3, %rdx # Should not execute

• Should only execute first 8 instructions

Detecting Mispredicted Branch

Condition Trigger
Mispredicted Branch

Control for Misprediction

Condition F D E M W
Mispredicted Branch normal bubble bubble normal normal

Return Example
demo-retb.ys
0x000: imovq Stack,%rsp # Initialize stack pointer
0x00a: call p # Procedure call
0x013: imovq $5,%rsi # Return point
0x01d: halt
0x020: pos 0x20
0x020: p: imovq $-1,%rdi # Procedure
0x02a: ret
0x02b: imovq $1,%rax # Should not be executed
0x035: imovq $2,%rcx # Should not be executed
0x03f: imovq $3,%rdx # Should not be executed
0x049: imovq $4,%rbx # Should not be executed
0x100: pos 0x100
0x100: Stack # Stack: Stack pointer

• Previously executed three additional instructions

Handling Misprediction

Condition F D E M W
Mispredicted Branch normal bubble bubble normal normal

Correct Return Example
demo-retb.ys
0x024: ret
0x02d: pos 0x2d
0x02d: bubble
0x02d: bubble
0x02d: bubble
0x032: imovq $5,%esi # Return
0x033: imovq $5,%esi # Return

• 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

Return Example
Detected Return

**Condition** | **Trigger**
--- | ---
Processing ret | IRET in { D_icode, E_icode, M_icode } 
Load/Use Hazard | E_icode in [ IMMMOVQ, IPOPOQ ] &
Mispredicted Branch | E_icode = JXX & Ie_Cnd 

**Action (on next cycle)**

<table>
<thead>
<tr>
<th><strong>Condition</strong></th>
<th><strong>F</strong></th>
<th><strong>D</strong></th>
<th><strong>E</strong></th>
<th><strong>al</strong></th>
<th><strong>W</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing ret</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Load/Use Hazard</td>
<td>stall</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Mispredicted Branch</td>
<td>normal</td>
<td>bubble</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>

Control for Return

# demo-retb
S016: set bubble bubble bubble set \[ F \rightarrow E \rightarrow M \rightarrow W \] 
S014: immovq $5, $eal # Return \[ F \rightarrow E \rightarrow M \rightarrow W \]

<table>
<thead>
<tr>
<th><strong>Condition</strong></th>
<th><strong>F</strong></th>
<th><strong>D</strong></th>
<th><strong>E</strong></th>
<th><strong>al</strong></th>
<th><strong>W</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing ret</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>

Special Control Cases

**Detection**

<table>
<thead>
<tr>
<th><strong>Condition</strong></th>
<th><strong>Trigger</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing ret</td>
<td>IRET in { D_icode, E_icode, M_icode }</td>
</tr>
</tbody>
</table>
| Load/Use Hazard | E_icode in [ IMMMOVQ, IPOPOQ ] &
Mispredicted Branch | E_icode = JXX & Ie_Cnd |

**Implementing Pipeline Control**

- Combinational logic generates pipeline control signals
- Action occurs at start of following cycle

Initial Version of Pipeline Control

```python
bool F_stall = 
# Conditions for a load/use hazard
E_icode in [ IMMMOVQ, IPOPOQ ] &
D_dstM in { d_srcA, d_srcB } 

# Stalling at fetch while ret passes through pipeline
IRET in { E_icode, E_icode, M_icode } |

bool D_stall = 
# Conditions for a load/use hazard
E_icode in [ IMMMOVQ, IPOPOQ ] &

# Stalling at fetch while ret passes through pipeline
IRET in { E_icode, E_icode, M_icode } |

bool D_bubble = 
# Mispredicted branch
E_icode = JXX & Ie_Cnd |

# Stalling at fetch while ret passes through pipeline
IRET in { E_icode, E_icode, M_icode } |

bool E_bubble = 
# Mispredicted branch
E_icode = JXX & Ie_Cnd |

# Load/Use hazard
E_icode in [ IMMMOVQ, IPOPOQ ] &
D_dstM in { d_srcA, d_srcB } |
```

Control Combinations

- Special cases that can arise on same clock cycle
- Combination A
  - Not-taken branch
  - set instruction at branch target
- Combination B
  - Instruction that reads from memory to Ie_Cnd
  - Followed by set instruction
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 zet passes through pipeline
    - Three clock cycles wasted

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

Control Combination A

<table>
<thead>
<tr>
<th>Condition</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>al</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing ret</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Mispredicted Branch</td>
<td>normal</td>
<td>bubble</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Combination</td>
<td>stall</td>
<td>bubble</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>

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

Control Combination B

<table>
<thead>
<tr>
<th>Condition</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>al</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing ret</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Load/Use Hazard</td>
<td>stall</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Combination</td>
<td>stall</td>
<td>bubble</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
</tr>
</tbody>
</table>

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

Handling Control Combination B

<table>
<thead>
<tr>
<th>Condition</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>al</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing ret</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Load/Use Hazard</td>
<td>stall</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Combination</td>
<td>stall</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>

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

Corrected Pipeline Control Logic

```cpp
bool b_bubble =
    # Mispredicted branch
    (E_icode == IMM64 || !e_Cnd) ||
    # Stalling at fetch while ret passes through pipeline
    ( M_icode & E_icode ) &&
    # but not condition for a load/use hazard
    && !E_dstM in { IMRMOVQ, IPOPQ } &&
    && !E_outM in { d_srcA, d_srcB });
```

<table>
<thead>
<tr>
<th>Condition</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>al</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing ret</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Load/Use Hazard</td>
<td>stall</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Combination</td>
<td>stall</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>

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