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

Data Dependencies: No Nop
 Techniques for Pipeline Correctness

- Stalling: delay later instructions until ready
  - Used for return instructions
  - Could use for data hazards, but slow
- Forwarding: pass needed data back early
  - More efficient for data hazards
- Cancelling: remove/disable instructions that should not execute
  - Used for mis-predicted branches
  - Used for exceptions (see textbook)

Stalling for Data Dependencies

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

Detecting Stall Condition

- Don't stall for register ID 15 (0xF)
  - Indicates absence of register operand
- Don't stall for failed conditional move

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

Stall Condition

Source Registers
- srcA and srcB of current instruction in decode stage

Destination Registers
- dstE and dstM fields
- Instructions in execute, memory, and write-back stages

Special Case
- Don't stall for register ID 15 (0xF)
- Don't stall for failed conditional move

Stalling X3

- 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 Stalling

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

Pipeline Control

<table>
<thead>
<tr>
<th>Pipelines</th>
<th>Control Logic</th>
<th>Mode Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Stall</td>
<td>E_bubble, 0</td>
</tr>
<tr>
<td>M</td>
<td>Stall</td>
<td>M_bubble, 0</td>
</tr>
<tr>
<td>W</td>
<td>Stall</td>
<td>W_bubble, 0</td>
</tr>
</tbody>
</table>

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 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

Instruction: irmovl

<table>
<thead>
<tr>
<th>Register</th>
<th>Forwarding Example</th>
<th>Final Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>Forward from memory</td>
<td>10</td>
</tr>
<tr>
<td>%eax</td>
<td>Forward from execute</td>
<td>3</td>
</tr>
</tbody>
</table>

Data Forwarding Example #2

Instruction: irmovl

<table>
<thead>
<tr>
<th>Register</th>
<th>Forwarding Example</th>
<th>Final Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>Forward from memory</td>
<td>10</td>
</tr>
<tr>
<td>%eax</td>
<td>Forward from execute</td>
<td>3</td>
</tr>
</tbody>
</table>

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

Data Forwarding Example

Instruction: irmovl

<table>
<thead>
<tr>
<th>Register</th>
<th>Forwarding Example</th>
<th>Final Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>Forward from memory</td>
<td>10</td>
</tr>
<tr>
<td>%eax</td>
<td>Forward from execute</td>
<td>3</td>
</tr>
</tbody>
</table>
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

- Condition
  - Load/Use Hazard
- Trigger
  - E_load in (IMMOVIL, IPOPL) &
  - E_dstM in (d_srcA, d_srcB)
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>Load/Use Hazard</td>
<td>stall</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>

Administrative Break

- Buffer lab due tonight
- Assignment III posted today
  - Due Monday after spring break (careful, easy to forget)
- Quiz grade appeal process
  - Recommended: first discuss concern with a TA
  - Write short note describing the mistake
  - Give to instructor, I will recheck the entire quiz
  - N.B., no guarantee your total grade will increase

Branch Misprediction Example

demo-j.ys

0x000: xorl %eax, %eax # Not taken
0x002: jne t # Fall through
0x006: nop
0x006a: nop
0x010: halt
0x011: t: irmovl $2, %edx # Target (Should not execute)
0x017: irmovl $4, %edx # Should not execute

- Should only execute first 8 instructions

Handling Misprediction

demo-j.ys

0x000: xorl %eax, %eax
0x002: jne target # Not taken
0x011: t: irmovl $2, %edx # Target
0x017: irmovl $3, %edx # Target+1
0x007: irmovl $1, %eax # Fall through
0x00d: 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

Detecting Mispredicted Branch

demo-j.ys

0x000: xorl %eax, %eax
0x002: jne target # Not taken
0x011: t: irmovl $2, %edx # Target
0x017: irmovl $3, %edx # Target+1
0x007: irmovl $1, %eax # Fall through
0x00d: nop

Detecting Mispredicted Branch

<table>
<thead>
<tr>
<th>Condition</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredicted Branch</td>
<td>E_code = IJXX &amp; t_e_Cnd</td>
</tr>
</tbody>
</table>

Control for Misprediction

demo-j.ys

0x000: xorl %eax, %eax
0x002: jne target # Not taken
0x011: t: irmovl $2, %edx # Target
0x017: irmovl $3, %edx # Target+1
0x007: irmovl $1, %eax # Fall through
0x00d: nop

<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>Mispredicted Branch</td>
<td>normal</td>
<td>bubble</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>
Return Example

0x000:  IrmaVL Stack,%esp  # Initialize stack pointer
0x006:  call p            # Procedure call
0x00b:  IrmaVL $5,%esi   # Return point
0x011:  halt
0x020:  .pos 0x20
0x020:  p:  IrmaVL $
0x026:  ret
0x027:  IrmaVL $1,%eax  # Should not be executed
0x02d:  IrmaVL $2,%ecx  # Should not be executed
0x033:  IrmaVL $3,%edx  # Should not be executed
0x039:  IrmaVL $4,%ebx  # Should not be executed
0x100: .pos 0x100
0x100:  Stack:              # Stack: Stack pointer

Previously executed three additional instructions

Correct Return Example

0x00b:  IrmaVL $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

Detecting Return

Condition | Trigger
---|---
Processing ret | RET in (D_icode, E_icode, M_icode)

Control for Return

Condition | F | D | E | M | W
---|---|---|---|---|---
Processing ret | stall | bubble | normal | normal | normal

Special Control Cases

Detection

Condition | Trigger
---|---
Processing ret | RET in (D_icode, E_icode, M_icode)
Load/Use Hazard | E_icode in (IMRMVOL, IPOPL) &&
E_dstM in (d_srcA, d_srcB)
Mispredicted Branch | E_icode = IJXX & !e_Cnd

Action (on next cycle)

Condition | F | D | E | M | W
---|---|---|---|---|---
Processing ret | stall | bubble | normal | normal | normal
Load/Use Hazard | stall | stall | bubble | normal | normal
Mispredicted Branch | normal | bubble | bubble | normal | normal

Implementing Pipeline Control

Combination logic generates pipeline control signals
Action occurs at start of following cycle
Initial Version of Pipeline Control

bool E_stall =
# Conditions for a load/use hazard
E_icode in { IMRMOVL, IPOPL } || E_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 { IMRMOVL, IPOPL } || E_dstM in { d_srcA, d_srcB };
bool E_bubble =
# Mispredicted branch
(E_icode == JXX && \!E_Cnd) ||
# Load/use hazard
E_icode in { IMRMOVL, IPOPL } || E_dstM in { d_srcA, d_srcB };

Control Combinations

![Image of control combinations]

Load/use Mispredict ret 1 ret 2 ret 3
M E Load E M E M ret M ret E bubble D bubble
D Use D ret D ret D bubble D bubble

Combination A

Special cases that can arise on same clock cycle
Combination A
- Not-taken branch
- ret instruction at branch target
Combination B
- Instruction that reads from memory to hspec
- Followed by ret instruction

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 Mispredict ret 1
M E Load E M ret E M ret
D Use D ret D ret D ret

Combination B

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

Corrected Pipeline Control Logic

bool D_bubble =
# Mispredicted branch
(E_icode == JXX && \!E_Cnd) ||
# Stalling at fetch while ret passes through pipeline
IRET in { E_icode, E_icode, M_icode } \# but not condition for a load/use hazard
44 (E_icode in { IMRMOVL, IPOPL } ||
44 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