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 \texttt{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

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```

![Diagram showing data dependencies and cycle 6 with an error]

Cycle 6

- `W`: 
  - `R[\%rax] \leftarrow 3`
  - `...`
  - `D`
  - `valA \leftarrow R[\%rdx] = 10`
  - `valB \leftarrow R[\%rax] = 0`
Data Dependencies: No Nop

# demo-h0.ys

0x000: irmovq $10, %rdx
0x00a: irmovq $3, %rax
0x014: addq %rdx, %rax
0x016: halt

Cycle 4

M

M_valE = 10
M_dstE = %rdx

E

e_valE ← 0 + 3 = 3
E_dstE = %rax

D

valA ← R[%rdx] = 0
valB ← R[%rax] = 0

Error
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
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)
  - Indicates absence of register operand
  - Or failed cond. move
Detecting Stall Condition

```plaintext
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
    bubble
0x016: addq %rdx,%rax
0x018: halt
```

Diagram:

```
Cycle 6
W
W_dstE = %rax
W_valE = 3
...
D
srcA = %rdx
srcB = %rax
```
# demo-h0.ys

0x000: `irmovq $10,%rdx`

0x00a: `irmovq $3,%rax`

`bubble`

`bubble`

`bubble`

0x014: `addq %rdx,%rax`

0x016: `halt`
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

# demo-h0.ys
0x000:  irmovq  $10,%rdx
0x00a:  irmovq  $3,%rax
0x014:  addq  %rdx,%rax
0x016:  halt

<table>
<thead>
<tr>
<th>Cycle 8</th>
<th>Write Back</th>
<th>Memory</th>
<th>Execute</th>
<th>Decode</th>
<th>Fetch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bubble</td>
<td>bubble</td>
<td>0x014:</td>
<td>0x016:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>addq  %rdx,%rax</td>
<td>halt</td>
<td></td>
</tr>
</tbody>
</table>
Implementing Stalling

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

Pipeline Control
Pipeline Register Modes

**Normal**
- Input = y
- Output = x
- stall = 0
- bubble = 0
- Rising clock
- Output = y

**Stall**
- Input = y
- Output = x
- stall = 1
- bubble = 0
- Rising clock
- Output = x

**Bubble**
- Input = y
- Output = x
- stall = 0
- bubble = 1
- Rising clock
- Output = nop
Data Forwarding

Naïve 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

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

```plaintext
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```

Cycle 6

```
W
W_dstE = %rax
W_valE = 3

D
srcA = %rdx
srcB = %rax
valA ← R[%rdx] = 10
valB ← W_valE = 3
```
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 #2

```plaintext
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

Register `%rdx`
- Generated by ALU during previous cycle
- Forward from memory as valA

Register `%rax`
- Value just generated by ALU
- Forward from execute as valB
**Forwarding Priority**

# demo-priority.ys
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt

**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
Implementing Forwarding

## What should be the A value?

```java
int d_valA = [
    // Use incremented PC
    # D_icode in { ICALL, IJXX } : D_valP;
    # Forward valE from execute
    d_srcA == e_dstE : e_valE;
    # Forward valM from memory
    d_srcA == M_dstM : m_valM;
    # Forward valE from memory
    d_srcA == M_dstE : M_valE;
    # Forward valM from write back
    d_srcA == W_dstM : W_valM;
    # Forward valE from write back
    d_srcA == W_dstE : W_valE;
    # Use value read from register file
    1 : d_rvalA;
];
```
# 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
Detecting Load/Use Hazard

<table>
<thead>
<tr>
<th>Condition</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/Use Hazard</td>
<td>( E_{\text{icode}} \in { \text{IMRMOVQ, IPOPQ} } ) &amp; ( E_{\text{dstM}} \in { \text{d_srcA, d_srcB} } )</td>
</tr>
</tbody>
</table>
## Control for Load/Use Hazard

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

### # demo-luh.ys

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>irmovq $128, %rdx</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>0x00a</td>
<td>irmovq $3, %rcx</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>0x014</td>
<td>rmmovq %rcx, 0(%rdx)</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>0x01e</td>
<td>irmovq $10, %ebx</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>0x028</td>
<td>mrmovq 0(%rdx), %rax</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>

### Condition

<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>
Branch Misprediction Example

demo-j.ys

0x000: xorq %rax,%rax
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
0x023: irmovq $4, %rcx # Should not execute
0x02d: irmovq $5, %rdx # Should not execute

- Should only execute first 8 instructions -
Handling Misprediction

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredicted Branch</td>
<td>$E_{i\text{code}} = IJXX &amp; !e_{Cnd}$</td>
</tr>
</tbody>
</table>
Control for Misprediction

Control for Misprediction

```
# demo-j.ys
0x000: xorq %rax, %rax
0x002: jne target  # Not taken
0x016: irmovq $2, %rdx  # Target
       bubble
0x020: irmovq $3, %rbx  # Target+1
       bubble
0x00b: irmovq $1, %rax  # Fall through
0x015: halt
```

<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

demo-retb.ys

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

- Previously executed three additional instructions
Correct Return Example

```plaintext
# demo-retb

0x026:    ret

bubble

bubble

bubble

0x013:    irmovq $5,%rsi  # 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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing ret</td>
<td>IRET in { D_iCode, E_iCode, M_iCode }</td>
</tr>
</tbody>
</table>
Control for Return

# demo-retb

0x026: ret

bubble

bubble

bubble

0x014: irmovq $5,%rsi # Return

<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>Processing</td>
<td>ret</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>
## Special Control Cases

### Detection

<table>
<thead>
<tr>
<th>Condition</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing <code>ret</code></td>
<td>IRET in { D_icode, E_icode, M_icode }</td>
</tr>
<tr>
<td>Load/Use Hazard</td>
<td>E_icode in { IMRMOVQ, IPOPQ } &amp;&amp; E_dstM in { d_srcA, d_srcB }</td>
</tr>
<tr>
<td>Mispredicted Branch</td>
<td>E_icode = IJXX &amp; !e_Cnd</td>
</tr>
</tbody>
</table>

### Action (on next cycle)

<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>Processing <code>ret</code></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>
Implementing Pipeline Control

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

```c
bool F_stall =
    # Conditions for a load/use hazard
    E_icode in \{ IMRMOVQ, IPOPQ \} && E_dstM in \{ d_srcA, d_srcB \} ||
    # Stalling at fetch while ret passes through pipeline
    IRET in \{ D_icode, E_icode, M_icode \};

bool D_stall =
    # Conditions for a load/use hazard
    E_icode in \{ IMRMOVQ, IPOPQ \} && E_dstM in \{ d_srcA, d_srcB \};

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 \};

bool E_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Load/use hazard
    E_icode in \{ IMRMOVQ, IPOPQ \} && E_dstM in \{ d_srcA, d_srcB \};
```
Control Combinations

- 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 `%rsp`
- 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

<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>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>
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 { IMRMVQ, IPOPQ }
        && E_dstM in { d_srcA, d_srcB });
```

<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>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><strong>stall</strong></td>
<td><strong>stall</strong></td>
<td><strong>bubble</strong></td>
<td><strong>normal</strong></td>
<td><strong>normal</strong></td>
</tr>
</tbody>
</table>

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