Pipelined Implementation
Data Hazards

CMPU 224 – Computer Organization
Jason Waterman
Overview

• Today: 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

1. `irmovq $50, @rax`
2. `addq @rax, @rbx`
3. `mrmovq 100(@rbx), @rdx`
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
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
• If instruction follows too closely after one that writes register, slow it down
• Hold instruction in decode
• Dynamically inject nop into execute stage
Stalling for Data Dependencies

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

```plaintext
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
  bubble
0x016: addq %rdx,%rax
0x018: halt
```
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 conditional move
Detecting Stall Condition

0x000: irmovq $10, %rdx
0x00a: irmovq $3, %rax
0x014: nop
0x015: nop
  bubble
0x016: addq %rdx, %rax
0x018: halt
Stalling x3

0x000: `irmovq $10, %rdx`

bubble
bubble
bubble

0x00a: `irmovq $3, %rax`

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

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

Cycle 8

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

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

• Current Pipeline
  • Source operands read from register file in decode stage
    • Needs to be in the register file at start of stage
    • Register is not written until completion of write-back stage

• Observation
  • Value is generated in either execute or memory stage

• One Cool Trick
  • Pass value directly from generating instruction to the decode stage
  • Value just needs to be available by the end of the decode stage
Data Forwarding Example

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

```
Cycle 6

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

0x000: `irmovq $10,%rdx`
0x00a: `irmovq $3,%rax`
0x014: `nop`
0x015: `nop`
0x016: `addq %rdx,%rax`
0x018: `halt`

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

- Register \( \%rax \)
  - Value just generated by ALU
  - Forward from execute as \( \text{valB} \)

- Register \( \%rdx \)
  - Generated by ALU during previous cycle
  - Forward from memory as \( \text{valA} \)

0x000: `irmovq $10,\%rdx`

0x00a: `irmovq $3,\%rax`

0x014: `addq \%rdx,\%rax`

0x016: `halt`

Cycle 4

- \( \text{M} \)
  - \( M\_\text{dstE} = \%rdx \)
  - \( M\_\text{valE} = 10 \)

- \( \text{E} \)
  - \( E\_\text{dstE} = \%rax \)
  - \( e\_\text{valE} \leftarrow 0 + 3 = 3 \)

- \( \text{D} \)
  - \( \text{srcA} = \%rdx \)
  - \( \text{srcB} = \%rax \)
  - \( \text{valA} \leftarrow M\_\text{valE} = 10 \)
  - \( \text{valB} \leftarrow e\_\text{valE} = 3 \)
**Forwarding Priority**

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?
  - Use matching value from nearest 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?
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 Hazard

- **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_icode ) in { IMRMOVQ, IPOPQ } &amp;&amp; ( E_dstM ) in { 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

<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>
Wrapup

• Today: 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
    • Use data forwarding
  • Load use hazard requires stalling for one cycle
    • Hold instructions in the Decode and Fetch stage, inject a bubble into the Execute stage

• Next time: Control Hazards
  • Mispredict conditional branch
    • Our design predicts all branches as being taken
    • Pipeline executes two extra instructions with mispredict
  • Getting return address for `ret` instruction
    • Pipeline executes three extra instructions