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 \texttt{nop} into execute stage

0x000: \texttt{irmovq} $10,\%rdx
0x00a: \texttt{irmovq} $3,\%rax
0x014: \texttt{nop}
0x015: \texttt{nop}
0x016: \texttt{addq} %rdx,%rax
0x018: \texttt{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

11/28/2022
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
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
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Execution Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>irmovq $10, %rdx</td>
<td>FDEEMW</td>
</tr>
<tr>
<td>0x00a</td>
<td>irmovq $3, %rax</td>
<td>FDEEMW</td>
</tr>
<tr>
<td>0x014</td>
<td>nop</td>
<td>FDEEMW</td>
</tr>
<tr>
<td>0x015</td>
<td>nop</td>
<td>FDEEMW</td>
</tr>
<tr>
<td>0x016</td>
<td>nop bubble</td>
<td></td>
</tr>
<tr>
<td>0x018</td>
<td>halt</td>
<td>FDEEMW</td>
</tr>
</tbody>
</table>

11/28/2022
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

Cycle 6

W

W_dstE = %rax
W_valE = 3

D

srcA = %rdx
srcB = %rax
Stalling x3

0x000: `irmovq $10, %rdx`

0x00a: `irmovq $3, %rax`

bubble
bubble
bubble

0x014: `addq %rdx, %rax`

0x016: `halt`

11/28/2022
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

```
Cycle 4
0x016: halt

Cycle 5
0x014: addq %rdx,%rax

Cycle 6
0x014: addq %rdx,%rax

Cycle 7
0x014: addq %rdx,%rax

Cycle 8
0x014: addq %rdx,%rax
```

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000: halt</td>
<td>0x010: addq %rdx,%rax</td>
<td>0x014: addq %rdx,%rax</td>
<td>bubble</td>
<td>0x00a: irmovq $3,%rax</td>
</tr>
<tr>
<td>0x00a: irmovq $10,%rdx</td>
<td>0x014: addq %rdx,%rax</td>
<td>0x016: halt</td>
<td>bubble</td>
<td>0x000: irmovq $10,%rdx</td>
</tr>
</tbody>
</table>

11/28/2022 | CMPU 224 -- Computer Organization | 12
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

\[ \text{stall} = 0 \quad \text{bubble} = 0 \]

\[
\begin{array}{c}
\text{Rising} \\
\text{clock} \\
\end{array}
\]

Output = x

\[
\begin{array}{c}
\text{Input = y} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Output = y} \\
\end{array}
\]

Stall

Input = y, Output = x

\[ \text{stall} = 1 \quad \text{bubble} = 0 \]

\[
\begin{array}{c}
\text{Rising} \\
\text{clock} \\
\end{array}
\]

Output = x

\[
\begin{array}{c}
\text{Input = y} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Output = x} \\
\end{array}
\]

Bubble

Input = y, Output = x

\[ \text{stall} = 0 \quad \text{bubble} = 1 \]

\[
\begin{array}{c}
\text{Rising} \\
\text{clock} \\
\end{array}
\]

Output = \text{nop}

\[
\begin{array}{c}
\text{Input = y} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Output = nop} \\
\end{array}
\]
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

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

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

- Multiple Forwarding Choices
  - Which one should have priority?
  - Use matching value from nearest pipeline stage

0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt
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?

```c
int d_valA[] = {
    // Use incremented PC
    # D i_code 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

```
0x000:  irmovq $128,%rdx
0x00a:  irmovq $3,%rcx
0x014:  rmovq %rcx, 0(%rdx)
0x01e:  irmovq $10,%rbx
0x028:  rmovq 0(%rdx),%rax # Load %rax
0x032:  addq %rbx,%rax # Use %rax
0x034:  halt
```
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