Pipelined Implementation
Data Hazards

CMPU 224 – Computer Organization
Jason Waterman
Overview

Make the pipelined processor work!

• 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

• 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
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., \texttt{valC} passes through decode
Data Dependencies: 2 Nop’s

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

Cycle 6

W
R[%rax] ← 3

D
valA ← R[ %rdx] = 10
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
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 \texttt{nop} into execute stage

0x000: \texttt{irmovq} $10, \%rdx
0x00a: \texttt{irmovq} $3, \%rax
0x014: \texttt{nop}
0x015: \texttt{nop}  
\textit{bubble}
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
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**
  - \( \text{srcA} \) and \( \text{srcB} \) of current instruction in decode stage

• **Destination Registers**
  - \( \text{dstE} \) and \( \text{dstM} \) 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`

0x00a: `irmovq $3, %rax`

`bubble`

`bubble`

`bubble`

0x014: `addq %rdx, %rax`

0x016: `halt`

Cycle 4
- `e_dstE = %rax`
- `M_dstE = %rax`
- `M = 0`
- `srcA = %rdx`
- `srcB = %rax`

Cycle 5
- `W = 0`
- `W_dstE = %rax`

Cycle 6
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>Write Back</th>
<th>Memory</th>
<th>Execute</th>
<th>Decode</th>
<th>Fetch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0x014:  addq  %rdx,%rax</td>
<td>0x016:  halt</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x014:  addq  %rdx,%rax</td>
<td>0x016:  halt</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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

**Stall**

- Input = y  
  - Output = x  
  - stall = 1  
  - bubble = 0

**Bubble**

- Input = y  
  - Output = x  
  - stall = 0  
  - bubble = 1
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 decode stage
  • Value just needs to be available by the end of decode stage
Data Forwarding Example

- `irmovq` in write-back stage
- Destination value in \( W \) pipeline register
- Forward as \( \text{valB} \) for decode stage

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

Cycle 6

\[
\begin{array}{ll}
W_{\text{dstE}} &= %rax \\
W_{\text{valE}} &= 3 \\
R[\%rax] &\leftarrow 3
\end{array}
\]

\[
\begin{array}{ll}
srcA &= %rdx \\
srcB &= %rax \\
valA &\leftarrow R[\%rdx] = 10 \\
valB &\leftarrow W_{\text{valE}} = 3
\end{array}
\]
Bypass Paths

• Decode Stage
  • Forwarding logic selects $\text{valA}$ and $\text{valB}$
  • Normally from register file
  • Forwarding: get $\text{valA}$ or $\text{valB}$ from later pipeline stage

• Forwarding Sources
  • Execute: $\text{valE}$
  • Memory: $\text{valE}$, $\text{valM}$
  • Write back: $\text{valE}$, $\text{valM}$
Data Forwarding Example #2

- **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`

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

#define A
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

4x000: `irmovq $128,%rdx`
0x00a: `irmovq $3,%rcx`
0x014: `rmmovq %rcx, 0(%rdx)`
0x01e: `irmovq $10,%rbx`
0x028: `rmmovq 0(%rdx),%rax`  # Load %rax
0x032: `addq %rbx,%rax`  # Use %rax
0x034: `halt`

Cycle 7
- `M_dslE = %rbx`
- `M_valE = 10`

Cycle 8
- `M_dstM = %rax`
- `m_valM = M[128] = 3`

Diagram:
- Forwarding logic
  - Load-use dependency indicated by arrows
  - Cycle stages: 1 to 11
  - Memory stage (M) and Decode stage (D) highlighted
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 { IMRMVQ, 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 \texttt{ret} instruction
    • Pipeline executes three extra instructions