Pipelined Implementation
Data Hazards
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. \texttt{irmovq $50, @rax}
2. \texttt{addq @rax, @rbx}
3. \texttt{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., \texttt{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
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: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
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
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
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: 

```assembly
 irmovq $10, %rdx
```

0x00a: 

```assembly
 irmovq $3, %rax
```

0x014: 

```assembly
 nop
```

0x015: 

```assembly
 nop
```

*bubble*

0x016: 

```assembly
 addq %rdx, %rax
```

0x018: 

```assembly
 halt
```
Stalling x3

0x000: `irmovq $10, %rdx`
0x00a: `irmovq $3, %rax`

`bubble`

`bubble`

`bubble`

0x014: `addq %rdx, %rax`
0x016: `halt`

Cycle 4

- `w_dstE = %rax`
- `e_dstE = %rax`
- `M_dstE = %rax`
- `srcA = %rdx`
- `srcB = %rax`

Cycle 5

- `w_dstE = %rax`

Cycle 6

- `w`
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

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

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>bubble</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bubble</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x014:</td>
<td>addq</td>
<td>%rdx,%rax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x016:</td>
<td>halt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x014:</td>
<td>addq</td>
<td>%rdx,%rax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x016:</td>
<td>halt</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

Output = x

Output = y

Output = x

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

```
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

- 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

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

Cycle 4

- `M_dstE = %rdx`
  - `M_valE = 10`
- `E_dstE = %rax`
  - `e_valE := 0 + 3 = 3`
- `srcA = %rdx`
- `srcB = %rax`
  - `valA := M_valE = 10`
  - `valB := e_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?

```c
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

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