Pipelined Implementation
Data Hazards

CMPU 224 – Computer Organization
Jason Waterman
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., valC passes through decode
Data Dependencies: No Nop

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
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 \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: 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

```plaintext
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

```
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 \texttt{nop} into execute stage

```
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

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

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

```
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>bubble</td>
<td>bubble</td>
<td>0x014: addq %rdx,%rax</td>
<td>0x016: halt</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
- 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
  • Source operands read from register file in decode stage
    • Needs to be in register file at start of stage
    • Register is not written until completion of write-back stage

• Observation
  • Value generated in execute or memory stage

• 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

- \texttt{irmovq} in write-back stage
- Destination value in W pipeline register
- Forward as valB for decode stage

```
0x000: \texttt{irmovq} $10, rdx
0x00a: \texttt{irmovq} $3, rax
0x014: nop
0x015: nop
0x016: addq rdx, rax
0x018: halt
```
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: \( \text{valE, valM} \)
  • Write back: \( \text{valE, valM} \)
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: `irmovq $10,%rdx`
0x00a: `irmovq $3,%rax`
0x014: `addq %rdx,%rax`
0x016: `halt`

### Cycle 4

- **M**
  - `M_dstE = %rdx`
  - `M_valE = 10`
- **E**
  - `E_dstE = %rax`
  - `e_valE ← 0 + 3 = 3`
- **D**
  - `srcA = %rdx`
  - `valA ← M_valE = 10`
  - `srcB = %rax`
  - `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

```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 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:  mrmovq 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

0x000: `imovq $128,%rdx`  F D E M W
0x00a: `imovq $3,%rcx`  F D E M W
0x014: `rmovq %rcx, 0(%rdx)`  F D E M W
0x01e: `imovq $10,%rbx`  F D E M W
0x028: `mimovq 0(%rdx),%rax # Load %rax`  F D E M W
    bubble
0x032: `addq %rbx,%rax # Use %rax`  F D E M W
0x034: `halt`  F F D E M W

Cycle 8

W

W_dstE = %rbx
W_valE = 10

M

M_dstM = %rax
m_valM ← M[128] = 3

D

valA ← W_valE = 10
valB ← m_valM = 3
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