Pipelined Implementation
Part 2

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., 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
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`

Cycle 4

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

Cycle 5

- `W_dstE = %rax`

Cycle 6

- `W`

11/20/2018

CMPU 224 -- Computer Organization
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

---

<table>
<thead>
<tr>
<th>Cycle 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Back</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>Execute</td>
</tr>
<tr>
<td>Decode</td>
</tr>
<tr>
<td>Fetch</td>
</tr>
</tbody>
</table>

11/20/2018
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  

Rising clock

Output = y

Stall

Input = y  
Output = x  

Rising clock

Output = x

Bubble

Input = y  
Output = x  

Rising clock

Output = nop
Data Forwarding

• Naïve Pipeline
  • Register isn’t written until completion of write-back stage
  • Source operands read from register file in decode stage
    • Needs to be in register file at start of stage

• Observation
  • Value generated in execute or memory stage

• Trick
  • Pass value directly from generating instruction to decode stage
  • Needs to be available at end of 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:  irmovq $10,%rdx
0x00a:  irmovq $3,%rax
0x014:  addq %rdx,%rax
0x016:  halt
Forced Priority

- Multiple Forwarding Choices
  - Which one should have priority
  - Use matching value from earliest 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
## 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 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: lmovq $128,%rdx
0x00a: lmovq $3,%rcx
0x014: rmovq %rcx, 0(%rdx)
0x01e: lmovq $10,%rbx
0x028: rmovq 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>$E\text{._icode in { IMRMOVQ, IPOPQ } \ E\text{._dstM in { d_srcA, d_srcB }}$</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>