Pipeline Implementation Wrapup

CMPU 224 – Computer Organization
Jason Waterman
Overview

Make the pipelined processor work!

• 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

• Control Hazards
  • Mispredict conditional branch
  • Getting return address for `ret` instruction

• Exceptional Conditions

• Performance Analysis
Control Hazards

• Occurs when the processor cannot reliably determine the address of the next instruction based on the current instruction in the fetch stage

• Happens in two places
  • Jump instructions (when mispredicting)
  • Return
Branch Misprediction Example

0x000: xorq %rax, %rax
0x002: jne target  # Not taken
0x00b: irmovq $1, %rax  # Fall through
0x015: halt
0x016: target:
0x016: irmovq $2, %rdx  # Target
0x020: irmovq $3, %rcx  # Target+1
0x02a: halt
Handling Misprediction

- Predict branch as taken
  - Fetch two instructions at target

- Cancel when mispredicted
  - Detect branch not-taken in execute stage
  - On following cycle, replace instructions in execute and decode by bubbles
  - **No side effects have occurred yet**
Detecting Mispredicted Branch

Condition | Trigger
------------|------------------
Mispredicted Branch | E_icode = IJXX & !e_Cnd
Control for Misprediction

0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovq $2,%rdx # Target
    bubble
0x020: irmovq $3,%rbx # Target+1
    bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt

<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>Mispredicted Branch</td>
<td>normal</td>
<td>bubble</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>
Handling returns

• Return address is stored on the stack
  • Need to read the address from memory
  • Return address is not available until the end of the memory stage

• Can not predict the return address
  • Fetch the next instruction (incorrectly)
    • Have to fetch something
  • Immediately replace the just fetched instruction with a bubble for the decode stage
  • `ret` instruction proceeds through the memory stage
    • We then have the correct address in $W_{valM}$
Return Example

0x000: irmovq stack,%rsp # Intialize stack pointer
0x00a: call proc # Procedure call
0x013: irmovq $10,%rdx # Return point
0x01d: halt
0x020: .pos 0x20
0x020: proc:
0x020: ret # Return immediately
0x021: rrmovq %rdx,%rbx # Not executed
0x030: .pos 0x30
0x030: stack: # stack: Stack pointer
Return Details

0x00: irmovq stack, %rsp
0x0a: call proc
0x20: ret
0x21: rrmovq %rdx, %rbx (bubble)
0x21: rrmovq %rdx, %rbx (bubble)
0x21: rrmovq %rdx, %rbx (bubble)
0x13: irmovq $10, %rdx
Simplified Return Example

- As `ret` passes through the pipeline, stall at fetch stage
- While in decode, execute and memory stage
  - Inject bubble into decode
- Release stall when reach write-back stage

```
0x020:  ret

bubble
bubble
bubble

0x013:  irmovq $10,%rdx # Return

valM = 0x013

valC ← 10
rB ← %rdx
```
Detecting Return

<table>
<thead>
<tr>
<th>Condition</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing ret</td>
<td>IRET in { D_icode, E_icode, M_icode }</td>
</tr>
</tbody>
</table>
Control for Return

0x020: ret

bubble

bubble

bubble

0x013: irmovq $10, %edx # Return

<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>Processing ret</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>
Special Control Cases

• Detection

<table>
<thead>
<tr>
<th>Condition</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing ret</td>
<td>IRET in { D_icode, E_icode, M_icode }</td>
</tr>
<tr>
<td>Load/Use Hazard</td>
<td>E_icode in { IMRMOVQ, IPOPQ } &amp;&amp; E_dstM in { d_srcA, d_srcB }</td>
</tr>
<tr>
<td>Mispredicted Branch</td>
<td>E_icode = IJXX &amp; !e_Cnd</td>
</tr>
</tbody>
</table>

• Action (on next cycle)

<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>Processing ret</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Load/Use Hazard</td>
<td>stall</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Mispredicted Branch</td>
<td>normal</td>
<td>bubble</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>
Implementing Pipeline Control

- Combinational logic generates pipeline control signals
- Action occurs at start of following cycle
Initial Version of Pipeline Control

bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB } ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode };

bool D_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };

bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode };

bool E_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };

<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>Processing ret</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Load/Use Hazard</td>
<td>stall</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Mispredicted Branch</td>
<td>normal</td>
<td>bubble</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>
Control Combinations

• Special cases that can arise on same clock cycle

• Combination A
  • Not-taken branch
  • `ret` instruction at branch target (`ret` should not be executed)

• Combination B
  • Instruction that reads from memory to `%rsp`
  • Followed by `ret` instruction
Control Combination A

- Should handle as mispredicted branch
- Stalls F pipeline register
- Our current pipeline logic handles this case correctly
Control Combination B

• Would attempt to bubble *and* stall pipeline register D
• Signaled by processor as pipeline error
Handling Control Combination B

- Load/use hazard should get priority
- `ret` instruction should be held in decode stage for additional cycle

<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>Processing <code>ret</code></td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Load/Use Hazard</td>
<td>stall</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Combination</td>
<td>stall</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>

- Load/use hazard should get priority
- `ret` instruction should be held in decode stage for additional cycle
Pipeline Summary

• Data Hazards
  • Most handled by forwarding
    • No performance penalty
    • Load/use hazard requires one cycle stall

• Control Hazards
  • Cancel instructions when detect mispredicted branch
    • Two clock cycles wasted
  • Stall fetch stage while ret passes through pipeline
    • Three clock cycles wasted

• Control Combinations
  • Must analyze carefully
  • First version had a pipeline error
    • Only arises with unusual instruction combination
Overview

Make the pipelined processor work!

- 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

- Control Hazards
  - Mispredict conditional branch
  - Getting return address for `ret` instruction

- Exceptional Conditions

- Performance Analysis
Exceptions

• Conditions under which processor cannot continue normal operation

• Causes
  • Halt instruction
  • Bad address for instruction or data
  • Invalid instruction

• Typical Desired Action
  • Complete some instructions
    • Either current or previous (depends on exception type)
  • Discard others
  • Call exception handler
    • Like an unexpected procedure call

• Our Implementation
  • Halt when instruction causes exception
Exception Examples

• Detect in Fetch Stage

```
jmp $-1    # Invalid jump target
.byte 0xFF # Invalid instruction code
halt       # Halt instruction
```

• Detect in Memory Stage

```
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # invalid address
```
Exceptions in Pipeline Processor #1

irmovq $100,%rax
rmmovq %rax,0x10000(%rax)  # Invalid address
nop
.byte 0xFF  # Invalid instruction code

• Desired Behavior
  • rmmovq should cause exception
  • Following instructions should have no effect on processor state
Exceptions in Pipeline Processor #2

- Desired Behavior
  - No exception should occur

```
0x000: xorq %rax,%rax  # Set condition codes
0x002: jne t          # Not taken
0x00b: irmovq $1,%rax
0x015: irmovq $2,%rdx
0x01f: halt
0x020: t: .byte 0xFF  # Target
```

```
0x000: xorq %rax,%rax
0x002: jne t
0x020: t: .byte 0xFF
0x???: (I’m lost!)
0x00b: irmovq $1,%rax
```

Exception detected
Maintaining Exception Ordering

- Add status field to pipeline registers
- Fetch stage sets to either “AOK,” “ADR” (when bad fetch address), “HLT” (halt instruction) or “INS” (illegal instruction)
- Decode & execute pass values through
- Memory either passes through or sets to “ADR”
- Exception triggered only when instruction hits the write back stage
**Exception Handling Logic**

- **Fetch Stage**
  
  ```c
  # Determine status code for fetched instruction
  int f_stat = [
  imem_error: SADR;
  !instr_valid : SINS;
  f_icode == IHALT : SHLT;
  1 : SAOK;
  ];
  ```

- **Memory Stage**
  
  ```c
  # Update the status
  int m_stat = [
  dmem_error : SADR;
  1 : M_stat;
  ];
  ```

- **Writeback Stage**
  
  ```c
  int Stat = [
  # SBUB in earlier stages indicates bubble
  W_stat == SBUB : SAOK;
  1 : W_stat;
  ];
  ```
Side Effects in Pipeline Processor

irmovq $100,%rax
rmmovq %rax,0x10000(%rax)  # invalid address
addq %rax,%rax  # Sets condition codes

• Desired Behavior
  • \texttt{rmmovq} should cause exception
  • No following instruction should have any effect
Avoiding Side Effects

• Presence of Exception Should Disable State Update
  • Invalid instructions are converted to pipeline bubbles
    • Except have stat indicating exception status
  • Data memory will not write to invalid address
  • Prevent invalid update of condition codes
    • Detect exception in memory stage
    • Disable condition code setting in execute
    • Must happen in same clock cycle
  • Handling exception in final stages
    • When detect exception in memory stage
      • Start injecting bubbles into memory stage on next cycle
    • When detect exception in write-back stage
      • Stall excepting instruction
  • Included in HCL code
Overview

Make the pipelined processor work!

• 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

• Control Hazards
  • Mispredict conditional branch
  • Getting return address for `ret` instruction

• Exceptional Conditions

• Performance Analysis
Performance Metrics

• Clock rate
  • Measured in Gigahertz
  • Function of stage partitioning and circuit design
    • Keep amount of work per stage small

• Rate at which instructions executed
  • CPI: cycles per instruction
  • On average, how many clock cycles does each instruction require?
  • Function of pipeline design and benchmark programs
    • E.g., how frequently are branches mispredicted?
CPI for PIPE

- CPI ≈ 1.0
  - Fetch instruction each clock cycle
  - Effectively process new instruction almost every cycle
    - Although each individual instruction has latency of 5 cycles
  - CPI > 1.0
    - Sometimes must stall or cancel branches

- Computing CPI
  - C clock cycles
  - I instructions executed to completion
  - B bubbles injected (C = I + B)
  - CPI = \( \frac{C}{I} = \frac{(I+B)}{I} = 1.0 + \frac{B}{I} \)
    - Factor \( \frac{B}{I} \) represents average penalty due to bubbles
CPI for PIPE

- \( B/I = LP + MP + RP \)

- **LP:** Penalty due to load/use hazard stalling
  - Fraction of instructions that are loads: 0.25
  - Fraction of load instructions requiring stall: 0.20
  - Number of bubbles injected each time: 1
  \[ LP = 0.25 \times 0.20 \times 1 = 0.05 \]

- **MP:** Penalty due to mispredicted branches
  - Fraction of instructions that are cond. jumps: 0.20
  - Fraction of cond. jumps mispredicted: 0.40
  - Number of bubbles injected each time: 2
  \[ MP = 0.20 \times 0.40 \times 2 = 0.16 \]

- **RP:** Penalty due to ret instructions
  - Fraction of instructions that are returns: 0.02
  - Number of bubbles injected each time: 3
  \[ RP = 0.02 \times 3 = 0.06 \]

- Net effect of penalties: \( 0.05 + 0.16 + 0.06 = 0.27 \)
  \[ CPI = 1.27 \] (Not bad!)
Processor Summary

• Design Technique
  • Create uniform framework for all instructions
    • Want to share hardware among instructions
    • Connect standard logic blocks with bits of control logic

• Operation
  • State held in memories and clocked registers
  • Computation done by combinational logic
  • Clocking of registers/memories sufficient to control overall behavior

• Enhancing Performance
  • Pipelining increases throughput and improves resource utilization
  • Must make sure to maintain ISA behavior