Lecture 8: “Pipelined Processor Design”

John P. Shen & Gregory Kesden
September 25, 2017

Required Reading Assignment:
- Chapter 4 of CS:APP (3rd edition) by Randy Bryant & Dave O’Hallaron.

Recommended Reference:
- Chapters 1 and 2 of Shen and Lipasti (SnL).
Lecture 8: “Pipelined Processor Design”

1. Instruction Pipeline Design
   a. Motivation for Pipelining
   b. Typical Processor Pipeline
   c. Resolving Pipeline Hazards

2. Y86-64 Pipelined Processor (PIPE)
   a. Pipelining of the SEQ Processor
   b. Dealing with Data Hazards
   c. Dealing with Control Hazards

3. Motivation for Superscalar
Computational Example

System
- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Must have clock cycle of at least 320 ps
3-Way Pipelined Version

System
• Divide combinational logic into 3 blocks of 100 ps each
• Can begin new operation as soon as previous one passes through stage A.
  • Begin new operation every 120 ps
• Overall latency increases
  • 360 ps from start to finish

Delay = 360 ps
Throughput = 8.33 GIPS
Pipeline Diagrams

➢ Unpipelined

• Cannot start new operation until previous one completes

➢ 3-Way Pipelined

• Up to 3 operations in process simultaneously
Operating a Pipeline

Clock

OP1
A
B
C

OP2
A
B
C

OP3
A
B
C

Time

0
120
240
360
480
640

100 ps
20 ps
100 ps
20 ps
100 ps
20 ps

Comb. logic A
Reg
Comb. logic B
Reg
Comb. logic C
Reg

Clock

9/25/2017 (©J.P. Shen)
Pipelining Fundamentals

➢ Motivation:
   • Increase throughput with little increase in hardware.

   Bandwidth or Throughput = Performance

➢ Bandwidth (BW) = no. of tasks/unit time
➢ For a system that operates on one task at a time:
   • BW = 1/delay (latency)
➢ BW can be increased by pipelining if many operands exist which need the same operation, i.e. many repetitions of the same task are to be performed.
➢ Latency required for each task remains the same or may even increase slightly.
Limitations: Register Overhead

As we try to deepen pipeline, overhead of loading registers becomes more significant.

Percentage of clock cycle spent loading register:

- 1-stage pipeline: 6.25%
- 3-stage pipeline: 16.67%
- 6-stage pipeline: 28.57%

High speeds of modern processor designs obtained through very deep pipelining.

Delay = 420 ps, Throughput = 14.29 GIPS
Starting from an un-pipelined version with propagation delay $T$ and $BW = 1/T$

$P_{\text{pipelined}} = BW_{\text{pipelined}} = 1 / (T/k + S)$

where

$S = \text{delay through latch and overhead}$
Starting from an un-pipelined version with hardware cost $G$

$\text{Cost}_\text{pipelined} = kL + G$

where
- $L =$ cost of adding each latch, and
- $k =$ number of stages
Cost/Performance Trade-off

Cost/Performance:

\[ \frac{C}{P} = \frac{Lk + G}{1/(T/k + S)} = (Lk + G)\left(\frac{T}{k} + S\right) = LT + GS + LSk + GT/k \]

Optimal Cost/Performance: find min. \( \frac{C}{P} \) w.r.t. choice of \( k \)

\[
\frac{d}{dk}\left( \frac{Lk + G}{\frac{1}{T/k + S}} \right) = 0 + 0 + LS - \frac{GT}{k^2}
\]

\[ LS - \frac{GT}{k^2} = 0 \]

\[ k_{opt} = \sqrt{\frac{GT}{LS}} \]
"Optimal" Pipeline Depth ($k_{opt}$) Examples

Cost/Performance Ratio (C/P)

- $G=175, L=41, T=400, S=22$
- $G=175, L=21, T=400, S=11$

Pipeline Depth $k$

9/25/2017 (©J.P. Shen)
Typical Instruction Processing Steps

Processor State
- Program counter register (PC)
- Condition code register (CC)
- Register File
- Memories
  - Access same memory space
  - Data: for reading/writing program data
  - Instruction: for reading instructions

Instruction Processing Flow
- Read instruction at address specified by PC
- Process through (four) typical steps
- Update program counter
- (Repeat)

1. Fetch
   - Read instruction from instruction memory

2. Decode
   - Determine Instruction type;
     Read program registers

3. Execute
   - Compute value or address

4. Memory
   - Read or write data in memory

5. Write Back
   - Write program registers

6. PC Update
   - Update program counter
5-Stage Pipeline (PIPE)

1. Fetch
   - Instruction memory
   - PC increment

2. Decode
   - Instruction IFun
   - rA, rB, valC
   - srcA, srcB, dstA, dstB
   - Instruction register file

3. Execute
   - ALU
   - Cnd
   - aluA, aluB
   - valA, valB

4. Memory
   - Addr, Data
   - Data memory
   - valE

5. Write back
   - valM
   - Register file

6. PC update
   - newPC
   - valE, valM

Carnegie Mellon University
Instruction Dependencies & Pipeline Hazards

Sequential Code Semantics

The implied sequential precedence’s are over specifications. It is sufficient but not necessary to ensure program correctness.

A true dependency between two instructions may only involve one subcomputation of each instruction.
Inter-Instruction Dependencies

- **True data dependency**
  \[ r_3 \leftarrow r_1 \text{ op } r_2 \]
  \[ r_5 \leftarrow r_3 \text{ op } r_4 \]
  Read-after-Write (RAW)

- **Anti-dependency**
  \[ r_3 \leftarrow r_1 \text{ op } r_2 \]
  \[ r_1 \leftarrow r_4 \text{ op } r_5 \]
  Write-after-Read (WAR)

- **Output dependency**
  \[ r_3 \leftarrow r_1 \text{ op } r_2 \]
  \[ r_5 \leftarrow r_3 \text{ op } r_4 \]
  \[ r_3 \leftarrow r_6 \text{ op } r_7 \]
  Write-after-Write (WAW)

- **Control dependency**
Example: Quick Sort for MIPS

```
bge $10, $9, L2
mul $15, $10, 4
addu $24, $6, $15
lw $25, 0($24)
mul $13, $8, 4
addu $14, $6, $13
lw $15, 0($14)
```

L1:
```
addu $10, $10, 1
...  
```

L2:
```
addu $11, $11, -1
...  
```

# for (;(j<high)&&(array[j]<array[low]);++j);
# $10 = j; $9 = high; $6 = array; $8 = low
Resolving Pipeline Hazards

➢ Pipeline Hazards:
  • Potential violations of program dependencies
  • Must ensure program dependencies are not violated

➢ Hazard Resolution:
  • Static Method: Performed at compiled time in software
  • Dynamic Method: Performed at run time using hardware

➢ Pipeline Interlock:
  • Hardware mechanisms for dynamic hazard resolution
  • Must detect and enforce dependencies at run time
Pipeline Hazards

➢ Necessary conditions for data hazards:
  • WAR: write stage earlier than read stage
    • Is this possible in the F-D-E-M-W pipeline?
  • WAW: write stage earlier than write stage
    • Is this possible in the F-D-E-M-W pipeline?
  • RAW: read stage earlier than write stage
    • Is this possible in the F-D-E-M-W pipeline?

➢ If conditions not met, no need to resolve
➢ Check for both register and memory dependencies
### Pipeline Hazards Analysis (ALU)

<table>
<thead>
<tr>
<th>WAR:</th>
<th>WAW:</th>
<th>RAW:</th>
<th>RAW:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i) R3 ←</td>
<td>(i) R3 ←</td>
<td>(i) R3 ←</td>
<td>(i) R3 ←</td>
</tr>
<tr>
<td>(j) R3 ←</td>
<td>(j) R3 ←</td>
<td>(j) ← R3</td>
<td>(j) ← R3</td>
</tr>
</tbody>
</table>

1. **Fetch** & PC update
2. **Decode**
3. **Execute**
4. **Memory**
5. **Write back**
### Pipeline Stalling for RAW (ALU)

<table>
<thead>
<tr>
<th>Step</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fetch</td>
</tr>
<tr>
<td>2</td>
<td>Decode</td>
</tr>
<tr>
<td>3</td>
<td>Execute</td>
</tr>
<tr>
<td>4</td>
<td>Memory</td>
</tr>
<tr>
<td>5</td>
<td>Write back</td>
</tr>
</tbody>
</table>

#### Example

1. Fetch
   - \( R_3 \leftarrow R_2 + R_1 \)

2. Decode
   - \( R_3 \leftarrow R_2 + R_1 \)

3. Execute
   - \( R_3 \leftarrow R_2 + R_1 \)

4. Memory
   - \( R_3 \leftarrow R_2 + R_1 \)

5. Write back
   - \( R_3 \leftarrow R_2 + R_1 \)

---

[Diagram of pipeline stages]
Dealing with Data Hazards

➢ Must first detect **RAW hazards**
  • Compare read register specifiers for newer instructions with write register specifiers for older instructions
  • Newer instruction in D; older instructions in E, M

➢ Resolve hazard dynamically
  • **Stall** or **forward**

➢ Not all hazards because
  • No register written (store or branch)
  • No register is read (e.g. addi, jump)
  • Do something only if necessary
    • Use special encodings for these cases to prevent spurious detection
Data Forwarding for RAW (ALU)
Data Forwarding for RAW (Load)

1. Fetch
2. Decode
3. Execute
4. Memory
5. Write back

(i) \( R3 \leftarrow M[x] \)

(i+1) \( \leftarrow R3+R4 \)

(i+2) \( \leftarrow R3 \)

(i+3) \( \leftarrow R3 \)

(i+2) \( \leftarrow R3 \)

(i+1) \( \leftarrow R3+R4 \)

(i) \( R3 \leftarrow M[x] \)

(i) \( \leftarrow M[x] \)

(i) \( \leftarrow R3 \)

(i+2) \( \leftarrow R3 \)

(i+1) \( \leftarrow R3+R4 \)

(i+2) \( \leftarrow R3 \)

(i+1) \( \leftarrow R3+R4 \)

(i) \( \leftarrow M[x] \)

(i) \( \leftarrow M[x] \)

(i+1) \( \leftarrow R3+R4 \)

(i+2) \( \leftarrow R3 \)

(i+3) \( \leftarrow R3 \)

(i) \( \leftarrow M[x] \)

(i+1) \( \leftarrow R3+R4 \)

(i+2) \( \leftarrow R3 \)

(i+3) \( \leftarrow R3 \)

(i+1) \( \leftarrow R3+R4 \)

(i+2) \( \leftarrow R3 \)

(i+3) \( \leftarrow R3 \)
Dealing With Branches

1. Fetch
2. Decode
3. Execute
4. Memory
5. Write back

(i) cond: PC ← Y

(i+1) ← R1 + R2

(i+2) ← R3 + R4

(i+3) ← R5 + R6

(k) (target of br) fetch from M[Y]

PC ← Y(i+1)

R1 + R2

R3 + R4

R5 + R6

Write back

Fetch & PC update

Cond:

M

Data memory

Addr, Data

aluA, aluB

CC

ALU

Reg file

W, valM

W_valE, W_valM, W_dateE, W_dataE

Instruction memory

PC increment

W, valE

valA, valB

d_srcA, d_srcB

W_val

cond, ilth, (i, b, wsrc)

predPC

18-600 Lecture #8

Dealing With Branches
1. Instruction Pipeline Design
   a. Motivation for Pipelining
   b. Typical Processor Pipeline
   c. Resolving Pipeline Hazards

2. Y86-64 Pipelined Processor (PIPE)
   a. Pipelining of the SEQ Processor
   b. Dealing with Data Hazards
   c. Dealing with Control Hazards

3. Motivation for Superscalar
PIPE Pipeline Stages

- **Fetch (F)**
  - Select current PC
  - Read instruction
  - Compute incremented PC

- **Decode (D)**
  - Read program registers

- **Execute (E)**
  - Operate ALU

- **Memory (M)**
  - Read or write data memory

- **Write Back (W)**
  - Update register file
PIPE Hardware

- Pipeline registers hold intermediate values from instruction execution

- Instructions propagate “upward”
  - Older instructions “higher” in PIPE
  - Values passed from one stage to next
  - Cannot jump past stages
    - e.g., valC passes through decode
Feedback Paths

➢ Predicted PC
  • Guess value of next PC

➢ Branch information
  • Jump taken/not-taken
  • Fall-through or target address

➢ Return point
  • Read from memory

➢ Register updates
  • To register file write ports
Predicting the PC

- Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
- Guess which instruction will follow
  - Recover if prediction was incorrect
Our Prediction Strategy

➢ Instructions that Don’t Transfer Control
  • Predict next PC to be valP
  • Always reliable

➢ Call and Unconditional Jumps
  • Predict next PC to be valC (destination)
  • Always reliable

➢ Conditional Jumps
  • Predict next PC to be valC (destination)
  • Only correct if branch is taken
    • Typically right 60% of time

➢ Return Instruction
  • Don’t try to predict
Recovering from PC Misprediction

- Mispredicted Jump
  - Will see branch condition flag once instruction reaches memory stage
  - Can get fall-through PC from valA (value M_valA)

- Return Instruction
  - Will get return PC when ret reaches write-back stage (W_valM)
Resolving Pipeline Hazards

- **Data Hazards**
  - Instruction having register R as source follows shortly after instruction having register R as destination (RAW)
  - Common condition, don’t want to slow down pipeline

- **Control Hazards**
  - Mispredict conditional branch
    - Our design predicts all branches as being taken
    - Naïve pipeline executes two extra instructions
  - Getting return address for `ret` instruction
    - Naïve pipeline executes three extra instructions

- **Making Sure It Really Works**
  - What if multiple special cases happen simultaneously?
# demo-h2.ys

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

---

### Data Dependencies:

2 Nop's

---

Cycle 6

- W
  - `R[ %rax] ← 3`
  - ...
  - ...

- D
  - `valA ← R[ %rdx] = 10`
  - `valB ← R[ %rax] = 0`

Error
Data Dependencies:

# demo-h0.ys

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

Cycle 4

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

# demo-h2.ys
0x000: irmovq $10,%rdx
0x008: irmovq $3,%rax
0x014: nop
0x015: nop  
   bubble
0x016: addq %rdx,%rax
0x018: halt

1 2 3 4 5 6 7 8 9 10 11
F D E M W  
F D E M W  
F D E M W  
F D E M W  
F D E M W
F D E M W  
F F D E M W
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 cond. move
# demo-h2.ys

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

## Detecting Stall Condition

<table>
<thead>
<tr>
<th>Cycle 6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>W</strong></td>
</tr>
<tr>
<td><strong>W_dstE = %rax</strong></td>
</tr>
<tr>
<td><strong>W_valE = 3</strong></td>
</tr>
</tbody>
</table>

- **srcA = %rdx**
- **srcB = %rax**
# demo-h0.ys

0x000: irmovq $10,%rdx

0x00a: irmovq $3,%rax

bubble

degree

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

# demo-h0.ys

<table>
<thead>
<tr>
<th>Cycle 8</th>
<th>Write Back</th>
<th>Memory</th>
<th>Execute</th>
<th>Decode</th>
<th>Fetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle 4</td>
<td></td>
<td></td>
<td>0x014:</td>
<td>0x016:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>addq %rdx,%rax</td>
<td>halt</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bubble</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 5</td>
<td></td>
<td></td>
<td>0x014:</td>
<td>0x016:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>addq %rdx,%rax</td>
<td>halt</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bubble</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 6</td>
<td></td>
<td></td>
<td>0x014:</td>
<td>0x016:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>addq %rdx,%rax</td>
<td>bubble</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bubble</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 7</td>
<td></td>
<td></td>
<td>0x014:</td>
<td>0x016:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>addq %rdx,%rax</td>
<td>bubble</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bubble</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 8</td>
<td></td>
<td></td>
<td>0x014:</td>
<td>0x016:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>addq %rdx,%rax</td>
<td>bubble</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bubble</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

---

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition
Pipeline Register Modes

**Normal**
- Input = y
- Output = x
- stall = 0
- bubble = 0
- Rising clock

**Stall**
- Input = y
- Output = x
- stall = 1
- bubble = 0
- Rising clock

**Bubble**
- Input = y
- Output = x
- stall = 0
- bubble = 1
- Rising clock

- y
- x
- n o p

Output = y
Output = x
Output = x
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

```
# demo-h2.ys
0x000:  irmovq$10,%rdx
0x00a:  irmovq $3,%rax
0x014:  nop
0x015:  nop
0x016:  addq %rdx,%rax
0x018:  halt
```

Cycle 6

- `W_dstE` = `%rax`
- `W_valE` = 3
- `R[ %rax ]` ← 3
- `srcA` = `%rdx` ← `R[ %rdx ]` = 10
- `srcB` = `%rax`
- `valA` ← `W_valE` = 3
- `valB` ← `W_valE` = 3
Forwarding 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`

# demo-h0.ys
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

`valA ← M_valE = 10`
`valB ← e_valE = 3`

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`.
Forwarding Priority

- Multiple Forwarding Choices
  - Which one should have priority
  - Match serial semantics
  - Use matching value from earliest pipeline stage

# demo-priority.ys
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

# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq $3,%rcx
0x014: mmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mmovq 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
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 } &amp;&amp; $E_{\text{dstM}}$ 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>
Branch Misprediction Example

demo-j.ys

0x000: xorq %rax, %rax
0x002: jne t # Not taken
0x00b: irmovq $1, %rax # Fall through
0x015: nop
0x016: nop
0x017: nop
0x018: halt
0x019: t: irmovq $3, %rdx # Target
0x023: irmovq $4, %rcx # Should not execute
0x02d: irmovq $5, %rdx # Should not execute

• Should only execute first 8 instructions
Handling Misprediction

Predict branch as taken
- Fetch 2 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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredicted Branch</td>
<td>$E_{\text{icode}} = \text{IJXX} &amp; \neg e_{\text{Cnd}}$</td>
</tr>
</tbody>
</table>
Control for Misprediction

```
# demo-j.ys
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>
Return Example

demo-retb.ys

0x000: irmovq Stack,%rsp  # Initialize stack pointer
0x00a: call p           # Procedure call
0x013: irmovq $5,%rsi   # Return point
0x01d: halt
0x020: .pos 0x20
0x020: p: irmovq $-1,%rdi  # procedure
0x02a: ret
0x02b: irmovq $1,%rax   # Should not be executed
0x035: irmovq $2,%rcx   # Should not be executed
0x03f: irmovq $3,%rdx   # Should not be executed
0x049: irmovq $4,%rbx   # Should not be executed
0x100: .pos 0x100
0x100: Stack:           # Stack: Stack pointer

• Previously executed three additional instructions
Correct Return Example

# demo- retb
0x026: ret
     bubble
     bubble
     bubble
0x013: irmovq$5,%rsi # Return

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

# demo-retb

0x026:    ret

bubble
bubble
bubble

0x014:  irmovq $5,%rsi # 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</td>
<td>ret</td>
<td>stall</td>
<td>bubble</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 \textit{ret}</td>
<td>IRET in { D_icode, E_icode, M_icode }</td>
</tr>
<tr>
<td>Load/Use Hazard</td>
<td>E_icode in { IMRMMOVQ, 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 \textit{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
Control Combinations

- Special cases that can arise on same clock cycle

- **Combination A**
  - Not-taken branch
  - `ret` instruction at branch target

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

<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>Mispredicted Branch</td>
<td>normal</td>
<td>bubble</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>Combination</td>
<td>stall</td>
<td>bubble</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>

- Should handle as mispredicted branch
- Stalls F pipeline register
- But PC selection logic will be using M_valM anyhow
## Control Combination B

### Condition

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processing ret</strong></td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td><strong>Load/Use Hazard</strong></td>
<td>stall</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td><strong>Combination</strong></td>
<td>stall</td>
<td>bubble + stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>

- 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 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>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
Corrected Pipeline Control Logic

```c
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 }
  # but not condition for a 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>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
Lecture 8: “Pipelined Processor Design”

1. Instruction Pipeline Design
   a. Motivation for Pipelining
   b. Typical Processor Pipeline
   c. Resolving Pipeline Hazards

2. Y86-64 Pipelined Processor (PIPE)
   a. Pipelining of the SEQ Processor
   b. Dealing with Data Hazards
   c. Dealing with Control Hazards

3. Motivation for Superscalar
3 Major Penalty Loops of (Scalar) Pipelining

Performance Objective: Reduce CPI as close to 1 as possible.
Best Possible for Real Programs is as Low as CPI = 1.15.
CAN WE DO BETTER? … CAN WE ACHIEVE IPC > 1.0?

IBM RISC Experience:
[Agerwala and Cocke 1987]
➢ Load Penalty: 0.0625 CPI
➢ Branch Penalty: 0.085 CPI

Total CPI = 1.0 + 0.0625 + 0.085
= 1.1475 CPI
= 0.87 IPC
Amdahl’s Law and Instruction Level Parallelism

- h = fraction of time in serial code
- f = fraction that is vectorizable or parallelizable
- N = max speedup for f
- Overall speedup

\[ \text{Speedup} = \frac{1}{(1 - f) + \frac{f}{N}} \]
Revisit Amdahl’s Law

➢ Sequential bottleneck
➢ Even if N is infinite
   • Performance limited by non-vectorizable portion (1-f)

\[
\lim_{{N \to \infty}} \frac{1}{(1 - f) + \frac{f}{N}} = \frac{1}{1 - f}
\]
Pipelined Processor Performance Model

- $g = \text{fraction of time pipeline is filled}$
- $1 - g = \text{fraction of time pipeline is not filled (stalled)}$
“Tyranny of Amdahl’s Law”

- When \( g \) is even slightly below 100%, a big performance hit will result.
- Stalled cycles in the pipeline are the key adversary and must be minimized as much as possible.
- Can we somehow fill the pipeline bubbles (stalled cycles)?
Motivation for Superscalar Design

[Tilak Agerwala and John Cocke, 1987]

Speedup jumps from 3 to 4.3 for N=6, f=0.8, but s =2 instead of s=1 (scalar)

Typical Range
Superscalar Proposal

- Moderate the tyranny of Amdahl’s Law
  - Ease the sequential bottleneck
  - More generally applicable
  - Robust (less sensitive to f)
  - Revised Amdahl’s Law:

\[
\text{Speedup} = \frac{1}{\left(1 - f\right) + \frac{f}{N}}
\]
Iron Law of Processor Performance

\[
\frac{1}{\text{Processor Performance}} = \frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program (path length)}} \times \frac{\text{Cycles}}{\text{Instruction (CPI)}} \times \frac{\text{Time}}{\text{Cycle (cycle time)}}
\]

- **In the 1980’s** (decade of **pipelining**):
  - CPI: 5.0 $\rightarrow$ 1.15

- **In the 1990’s** (decade of **superscalar**):
  - CPI: 1.15 $\rightarrow$ 0.5 OR IPC: 0.87 $\rightarrow$ 2.0 (current best)

- **In the 2000’s** (decade of **multicore**):
  - Core CPI unchanged; chip CPI scales with #cores
Lecture 9:
“Superscalar Out-of-Order (O3) Processors”

John P. Shen & Gregory Kesden
September 27, 2017

Next Time ...

- **Required Reading Assignment:**
  - Chapter 4 of CS:APP (3rd edition) by Randy Bryant & Dave O’Hallaron.

- **Recommended Reading Assignment:**
  - Chapter 4 of Shen and Lipasti (SnL).