CS:APP Chapter 4
Computer Architecture
Pipelined Implementation
Part II

Randal E. Bryant

Carnegie Mellon University

http://csapp.cs.cmu.edu
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
  - 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?
Pipeline Stages

Fetch
- Select current PC
- Read instruction
- Compute incremented PC

Decode
- Read program registers

Execute
- Operate ALU

Memory
- Read or write data memory

Write Back
- Update register file
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: 2 Nop’s

```plaintext
# demo-h2.ys

0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```

![Cycle 6 Diagram]

```
valA ← R[%edx] = 10
valB ← R[%eax] = 0
```
# demo-h0.ys

0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: addl %edx,%eax
0x00e: halt

---

**Data Dependencies: No Nop**

---

Cycle 4

- **M**
  - $M_{valE} = 10$
  - $M_{dstE} = %edx$

- **E**
  - $e_{valE} \leftarrow 0 + 3 = 3$
  - $E_{dstE} = %eax$

- **D**
  - $valA \leftarrow R[%edx] = 0$
  - $valB \leftarrow R[%eax] = 0$

---

Error
Stalling for Data Dependencies

If instruction follows too closely after one that writes register, slow it down

Hold instruction in decode

Dynamically inject nop into execute stage

# demo-h2.ys

0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt

bubble
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
- Don’t stall for failed conditional move
Detecting Stall Condition

# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: nop
0x00d: nop
  
  bubble
0x00e: addl %edx,%eax
0x010: halt

Cycle 6

W
W_dstE = %eax
W_valE = 3

D
srcA = %edx
srcB = %eax
Stalling X3

```plaintext
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
    bubble
    bubble
    bubble
0x00c: addl %edx,%eax
0x00e: halt
```

Cycle 4 •

<table>
<thead>
<tr>
<th>F</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>

Cycle 5

<table>
<thead>
<tr>
<th>F</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

Cycle 6

<table>
<thead>
<tr>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>W_dstE = %eax</td>
</tr>
</tbody>
</table>

M

<table>
<thead>
<tr>
<th>M_dstE = %eax</th>
</tr>
</thead>
</table>

E

<table>
<thead>
<tr>
<th>E_dstE = %eax</th>
</tr>
</thead>
</table>

D

<table>
<thead>
<tr>
<th>srcA = %edx</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcB = %eax</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>srcA = %edx</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>srcB = %eax</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>srcA = %edx</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>srcB = %eax</td>
</tr>
<tr>
<td>---</td>
</tr>
</tbody>
</table>

-- 10 -- CS:APP2e
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
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: addl %edx,%eax
0x00e: 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>
Implementing Stalling

Pipe control logic

W_stat

W_stall

M_icode

M_bubble

M

stat

icode

Cnd

valE

valM

dstE
dstM

E

dstM

E_dstM

E_icode

E_bubble

d_srcB

d_srcA

D_icode

D_bubble

D_stall

D

stat

icode

ifun

valC

valA

valB
dstE
dstM
srcA
crB

dstB

F

predPC

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

Rising clock

Output = y

Output = x

Output = \text{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

- `irmovl` in write-back stage
- Destination value in W pipeline register
- Forward as `valB` for decode stage

```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```

![Diagram showing data forwarding example]
**Bypass Paths**

**Decode Stage**

- Forwarding logic selects \textit{valA} and \textit{valB}
- Normally from register file
- Forwarding: get \textit{valA} or \textit{valB} from later pipeline stage

**Forwarding Sources**

- Execute: \textit{valE}
- Memory: \textit{valE}, \textit{valM}
- Write back: \textit{valE}, \textit{valM}
Data Forwarding Example #2

```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```

**Register %edx**
- Generated by ALU during previous cycle
- Forward from memory as valA

**Register %eax**
- Value just generated by ALU
- Forward from execute as valB
Multiple Forwarding Choices

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

```
# demo-priority.ys
0x000: irmovl $1, %eax
0x006: irmovl $2, %eax
0x00c: irmovl $3, %eax
0x012: rrmovl %eax, %edx
0x014: 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
Implementing Forwarding

```c
int new_E_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**

```
# demo-luh.yc
0x000: irmovl $128,%edx
0x006: irmovl $3,%ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10,%ebx
0x018: mrmovl 0(%edx),%eax # Load %eax
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
```

**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
Detecting Load/Use Hazard

**Condition**

Load/Use Hazard

**Trigger**

\[
E\text{\_icode in \{ IMRMOVL, IPOPL \} } \land \land \nE\text{\_dstM in \{ d\_srcA, d\_srcB \}}
\]
## Control for Load/Use Hazard

### # demo-luh.ys

<table>
<thead>
<tr>
<th>0x000: irmovl $128,%edx</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x006: irmovl $3,%ecx</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>0x00c: rmmovl %ecx, 0(%edx)</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>0x012: irmovl $10,%ebx</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>0x018: mrmovl 0(%edx),%eax # Load %eax</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>0x01e: addl %ebx,%eax # Use %eax</td>
<td>F</td>
<td>D</td>
<td>D</td>
<td>E</td>
<td>M</td>
</tr>
<tr>
<td>0x020: halt</td>
<td>F</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
</tr>
</tbody>
</table>

- **Stall instructions in fetch and decode stages**
- **Inject bubble into execute stage**

### Condition

<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: xorl %eax,%eax
0x002: jne t # Not taken
0x007: irmovl $1, %eax # Fall through
0x00d: nop
0x00e: nop
0x00f: nop
0x010: halt
0x011: t: irmovl $3, %edx # Target (Should not execute)
0x017: irmovl $4, %ecx # Should not execute
0x01d: irmovl $5, %edx # Should not execute

- Should only execute first 8 instructions
Handling Misprediction

# demo-j.ys

0x000: xorl %eax,%eax

0x002: jne target # Not taken

0x011: t: irmovl $2,%edx # Target

bubble

0x017: irmovl $3,%ebx # Target+1

bubble

0x007: irmovl $1,%eax # Fall through

0x00d: nop

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

**Condition** | **Trigger**
--- | ---
Mispredicted Branch | $E_{\text{icode}} = \text{IJXX} \& !e_{\text{Cnd}}$
# demo-j.y8

0x000: xorl %eax,%eax

0x002: jne target # Not taken

0x011: t: irmovl $2,%edx # Target

  bubble

0x017: irmovl $3,%ebx # Target+1

  bubble

0x007: irmovl $1,%eax # Fall through

0x00d: nop

## Control for Misprediction

<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:   irmovl Stack,%esp          # Initialize stack pointer
0x006:   call p                    # Procedure call
0x00b:   irmovl $5,%esi            # Return point
0x011:   halt
0x020:   .pos 0x20
0x020:   p: irmovl $-1,%edi        # procedure
0x026:   ret
0x027:   irmovl $1,%eax            # Should not be executed
0x02d:   irmovl $2,%ecx            # Should not be executed
0x033:   irmovl $3,%edx            # Should not be executed
0x039:   irmovl $4,%ebx            # 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

0x00b:  irmovl $5,%esi # 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

```plaintext
# demo-retb

0x026:    ret

 bubble
 bubble
 bubble

0x00b:    irmovl $5,%esi # 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 { IMRMOV_L, IPO_PL } &amp;&amp; E_dstM in { d_src_A, d_src_B }</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
bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOV, IPOPL } && 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 { IMRMOV, IPOPL } && 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 { IMRMOV, IPOPL } && E_dstM in { d_srcA, d_srcB };
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 `%esp`
- Followed by `ret` instruction
Control Combination A

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

<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>
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
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 { IMRMOVl, IPOPL }
    && 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
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 \texttt{xxt} passes through pipeline
  - Three clock cycles wasted

Control Combinations

- Must analyze carefully
- First version had subtle bug
  - Only arises with unusual instruction combination