Pipelining

Anne Bracy

CS 3410

Computer Science

Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer.

See P&H Chapter: 4.5-4.8
Agenda

1. Intro to Pipelining
2. Implementing a 5-stage Pipeline
3. Pipeline in Action (example)
4. Hazards
Alice

Bob

They don’t always get along...
The Materials

Saw

Drill

Glue

Paint
The Instructions

N pieces, each built following same sequence:

1. Saw
2. Drill
3. Glue
4. Paint
Design 1: Sequential Schedule

Alice owns the room
Bob can enter when Alice is finished
Repeat for remaining tasks
No possibility for conflicts
Sequential Performance

Latency: 4 hours/task
Throughput: 1 task/4 hrs
Concurrency: 1

CPI = 4

Can we do better?
Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*

One person owns a stage at a time

4 stages

4 people working simultaneously

Everyone moves right in lockstep
Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*

One person owns a stage at a time

4 stages

4 people working simultaneously

Everyone moves right in lockstep

It still takes all four stages for one job to complete
Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*

One person owns a stage at a time
4 stages
4 people working simultaneously
Everyone moves right in lockstep
It still takes all four stages for one job to complete
Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*

One person owns a stage at a time

4 stages

4 people working simultaneously

Everyone moves right in lockstep

It still takes all four stages for one job to complete
Design 2: Pipelined Design

Partition room into stages of a pipeline

One person owns a stage at a time

4 stages

4 people working simultaneously

Everyone moves right in lockstep

It still takes all four stages for one job to complete
Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*

One person owns a stage at a time

4 stages

4 people working simultaneously

Everyone moves right in lockstep

It still takes all four stages for one job to complete
Latency: 4 hrs/task
Throughput: 1 task/hr
Concurrency: 4
CPI = 1
What if drilling takes twice as long, but gluing and paint take ½ as long?

Latency: 
Throughput: 
CPI =
Lessons

Principle:

Throughput increased by parallel execution
Balanced pipeline very important
Else slowest stage dominates performance

Pipelining:

• Identify pipeline stages
• Isolate stages from each other
• Resolve pipeline hazards (next)
Pipelining is great because:

A. You can fetch and decode the same instruction at the same time.

B. You can fetch two instructions at the same time.

C. You can fetch and decode two different instructions at the same time.

D. Instructions only need to visit the pipeline stages that they require.
Agenda

1. Intro to Pipelining
2. Implementing a 5-stage Pipeline
3. Pipeline in Action (example)
4. Hazards
MIPs designed for pipelining

- Instructions same length
  - 32 bits, easy to fetch and then decode

- 3 types of instruction formats
  - Easy to route bits between stages
  - Can read a register source before even knowing what the instruction is

- Memory access through lw and sw only
  - Access memory after ALU
Basic Pipeline

Five stage “RISC” load-store architecture

1. Instruction fetch (IF)
   – get instruction from memory, increment PC
2. Instruction Decode (ID)
   – translate opcode into control signals and read registers
3. Execute (EX)
   – perform ALU operation, compute jump/branch targets
4. Memory (MEM)
   – access memory if needed
5. Writeback (WB)
   – update register file
A Processor

Review: Single cycle processor
A Processor

Instruction Fetch
- Memory
  - inst
  - +4
  - PC
  - new pc

Instruction Decode
- register file
- imm
- extend
- control

Execute
- alu
- compute (jump/branch targets)
- addr
d_in
d_out
memory

Write-Back

Clock cycle | Time Graphs
--- | ---
1 | IF
2 | ID
3 | EX
4 | MEM
5 | WB
6 | IF
7 | ID
8 | EX
9 | MEM

Latency: 5 cycles
Throughput: 1 instr/cycle
Concurrency: 5

CPI = 1
Principles of Pipelined Implementation

Break instructions across multiple clock cycles (five, in this case)

Design a separate stage for the execution performed during each clock cycle

Add pipeline registers (flip-flops) to isolate signals between different stages
Pipelined Processor

- Instruction Fetch (IF/ID)
- Instruction Decode (ID/EX)
- Execute (EX/MEM)
- Memory (MEM/WB)

- Memory
- Register File
- ALU
- Control
- Compute jump/branch targets
- Compute new pc
- +4
- PC
- Inst
- Inst
- Ctrl
- Ctrl
- Ctrl
- Ctrl
- Addr
- Memory
- D_in
- D_out
- Write-Back

Steps:
1. Fetch
2. Decode
3. Execute
4. Memory
5. Write-Back
IF

Stage 1: Instruction Fetch

Fetch a new instruction every cycle
  • Current PC is index to instruction memory
  • Increment the PC at end of cycle (assume no branches for now)

Write values of interest to pipeline register (IF/ID)
  • Instruction bits (for later decoding)
  • PC+4 (for later computing branch targets)
PC instruction memory

addr mc

IF

- PC+4
- pcrel (PC-relative); e.g. BEQ, BNE
- pcabs (PC absolute); e.g. J and JAL
  \((PC+4)_{31..28} \cdot \text{target} \cdot 00\)
- pcreg (PC registers); e.g. JR
00 = read word
ID

Stage 2: Instruction Decode

On every cycle:

• Read IF/ID pipeline register to get instruction bits
• Decode instruction, generate control signals
• Read from register file

Write values of interest to pipeline register (ID/EX)

• Control information, Rd index, immediates, offsets, ...
• Contents of Ra, Rb
• PC+4 (for computing branch targets later)
Stage 1: Instruction Fetch

IF/ID

PC+4

inst

decode

Rest of pipeline

ID/EX

ctrl

PC+4

imm

extend

result
dest

regtister file

WE
Rd
D
Ra
Rb

A
B

ref
EX

Stage 3: Execute

On every cycle:

• Read ID/EX pipeline register to get values and control bits
• Perform ALU operation
• Compute targets (PC+4+offset, etc.) in case this is a branch
• Decide if jump/branch should be taken

Write values of interest to pipeline register (EX/MEM)

• Control information, Rd index, ...
• Result of ALU operation
• Value in case this is a memory store instruction
Stage 2: Instruction Decode

<table>
<thead>
<tr>
<th>ctrl</th>
<th>PC+4</th>
<th>imm</th>
<th>B</th>
<th>A</th>
<th>pcreg</th>
<th>branch?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID/EX</td>
<td>EX/</td>
<td>MEM</td>
<td>Rest of pipeline</td>
<td>alu</td>
<td>pcrel</td>
<td>pcabs</td>
</tr>
</tbody>
</table>

pcsel
MEM

Stage 4: Memory

On every cycle:

• Read EX/MEM pipeline register to get values and control bits
• Perform memory load/store if needed
  – address is ALU result

Write values of interest to pipeline register (MEM/WB)

• Control information, Rd index, ...
• Result of memory operation
• Pass result of ALU operation
Stage 3: Execute

EX/MEM

pcrel

pcabs

c ctrl

target

memory

MC

d_{in}

d_{out}

addr

Rest of pipeline

MEM/WB

c ctrl

MEM

branch?

pcsel
WB

Stage 5: Write-back

On every cycle:

- Read MEM/WB pipeline register to get values and control bits
- Select value and write to register file
Stage 4: Memory

ctrl

MEM/WB

result

dest

WB
Putting it all together!

IF/ID

ID/EX

EX/MEM

MEM/WB

PC

inst mem

+4

inst

PC+4

OP Rd

OP Rt Rd PC+4 imm

OP Rd

OP

addr d_in d_out mem
Consider a non-pipelined processor with clock period $C$ (e.g., 50 ns). If you divide the processor into $N$ stages (e.g., 6), your new clock period will be:

A. $C$
B. $N$
C. less than $C/N$
D. $C/N$
E. greater than $C/N$
Agenda

1. Intro to Pipelining
2. Implementing a 5-stage Pipeline
3. Pipeline in Action (example)
4. Hazards
Example: Sample Code (Simple)

```
add    r3, r1, r2;
nand   r6, r4, r5;
lw     r4, 20(r2);
add    r5, r2, r5;
sw     r7, 12(r3);
```
Example: Sample Code (Simple)

Assume eight-register machine

Run the following code on a pipelined datapath

```
add    r3  r1  r2  ; reg 3 = reg 1 + reg 2
nand   r6  r4  r5  ; reg 6 = ~(reg 4 & reg 5)
lw     r4  20 (r2) ; reg 4 = Mem[reg2+20]
add    r5  r2  r5  ; reg 5 = reg 2 + reg 5
sw     r7  12(r3) ; Mem[reg3+12] = reg 7
```
Fetch:
add 3 1 2

ID/EX

EX/MEM

MEM/ WB
sw 7 12(3) → add 5 2 5 → lw 4 20 (2) → nand 6 4 5 → add 3 1 2

Fetch:
sw 7 12(3)

Time: 5
IF/ID → ID/EX → EX/MEM → MEM/WB
sw 7 12(3)  add 5 2 5  lw 4 20(2)  nand 6 4!

No more instructions

Time: 6  IF/ID  ID/EX  EX/MEM  MEM/WB
No more instructions

Time: 8

IF/ID

ID/EX

EX/MEM

MEM/WB
Pipelining Recap

Pipelining is a powerful technique to mask latencies and increase throughput

- Logically, instructions execute one at a time
- Physically, instructions execute in parallel
  - Instruction level parallelism

Abstraction promotes decoupling

- Interface (ISA) vs. implementation (Pipeline)

Great! Are we done? Sadly, no....
1. Intro to Pipelining
2. Implementing a 5-stage Pipeline
3. Pipeline in Action (example)
4. Hazards
Hazards

Correctness problems associated w/processor design

3 kinds

- **Structural hazards**
  - Multiple instructions want to use same unit

- **Data hazards**
  - Results of instruction needed before

- **Control hazards**
  - Don’t know which side of branch to take
Dependences and Hazards

**Dependence**: relationship between two insns

- **Data**: two insns use same storage location
- **Control**: 1 insn affects whether another executes at all
- *Not a bad thing*, programs would be boring otherwise
- Enforced by making older insn go before younger one
  - Happens naturally in single-/multi-cycle designs
  - But not in a pipeline

**Hazard**: dependence & possibility of wrong insn order

- Effects of wrong insn order cannot be externally visible
- *Hazards are a bad thing*: most solutions either complicate the hardware or reduce performance
Data Hazards

Data Hazards

• register file reads occur in stage 2 (ID)
• register file writes occur in stage 5 (WB)
• next instructions may read values about to be written
  – i.e instruction may need values that are being computed
    further down the pipeline
  – *in fact, this is quite common*
Data Hazards

Clock cycle

1  2  3  4  5  6  7  8  9

time

add r3, r1, r2
sub r5, r3, r4
lw r6, 4(r3)
or r5, r3, r5
sw r6, 12(r3)
Data Hazards

• register file reads occur in stage 2 (ID)
• register file writes occur in stage 5 (WB)
• next instructions may read values about to be written

```
add r3, r1, r2
sub r5, r3, r4
```

Is there a dependence?
Is there a hazard?
How do we detect this?
Detecting Data Hazards

IF/ID: RD \neq 0 \&\& (IF/ID.RD = ID/EX.Rd \&\& IF/ID.RD = EX/MEM.Rd \&\& IF/ID.RD = MEM/WB.Rd)
Data Hazards

Data Hazards

- register file reads occur in stage 2 (ID)
- register file writes occur in stage 5 (WB)
- next instructions may read values about to be written

How to detect? Logic in ID stage:

\[
\text{stall} = (\text{IF/ID.Ra} \neq 0 \&\& \\
(\text{IF/ID.Ra} = \text{ID/EX.Rd} \mid \mid \\
\text{IF/ID.Ra} = \text{EX/M.Rd} \mid \mid \\
\text{IF/ID.Ra} = \text{M/WB.Rd}))
\]

\mid \mid \text{(same for Rb)}
Detecting Data Hazards

IF/ID

ID/EX

EX/MEM

MEM/WB

add r3, r1, r2
sub r5, r3, r5
or r6, r3, r4
add r6, r3, r8

inst

+4

PC

detect hazard

Rd

D

B

Ra

Rb

OP

 imm

PC+4

M

d_in
d_out

mem

addr

OP

Rd

OP

Rd

PC+4

OP

Rd

OP

PC

PC+4
Takeaway

Data hazards occur when an operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.
iClicker Question

Pipelining is great because:

A. You can fetch and decode the same instruction at the same time.
B. You can fetch two instructions at the same time.
C. You can fetch and decode two different instructions at the same time.
D. Instructions only need to visit the pipeline stages that they require.
Next Goal

What to do if data hazard detected?

Options

• Nothing
  – Change the ISA to match implementation

• Stall
  – Pause current and subsequent instructions till safe

• Forward/bypass
  – Forward data value to where it is needed
Stalling

How to stall an instruction in ID stage

• prevent IF/ID pipeline register update
  – stalls the ID stage instruction

• convert ID stage instr into nop for later stages
  – innocuous “bubble” passes through pipeline

• prevent PC update
  – stalls the next (IF stage) instruction
Detecting Data Hazards

IF/ID

PC

WE=0

MemWr=0
RegWr=0

add r3, r1, r2
sub r5, r3, r5
or r6, r3, r4
add r6, r3, r8

inst

PC+4

detect hazard

OP

Rd

Ra Rb

D

B

A

OP

OP

OP

Rd

M

D

addr
d_{in} d_{out}
mem

id

out

addr

PC+4

imm

Rt_Rd

OP

OP

OP

OP

OP

WE=0

MemWr=0
RegWr=0

if detect hazard

add r3,
r1,
r2

sub r5,
r3,
r5

or r6,
r3,
r4

add r6,
r3,
r8
add r3, r1, r2
sub r5, r3, r5
or r6, r3, r4
add r6, r3, r8
<table>
<thead>
<tr>
<th>time</th>
<th>Clock cycle</th>
<th>r3 = 10</th>
<th>r3 = 20</th>
<th>sub r5, r3, r5</th>
<th>or r6, r3, r4</th>
<th>add r6, r3, r8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IF</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>IF</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>Ex</td>
</tr>
<tr>
<td>5</td>
<td>IF</td>
<td>ID</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>6</td>
<td>IF</td>
<td>ID</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>7</td>
<td>IF</td>
<td>ID</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>

The diagram shows the execution of instructions with stalls indicated. The instruction `add r3, r1, r2` uses r3 = 10, and the instruction `add r6, r3, r8` uses r3 = 20. The `sub r5, r3, r5` instruction results in 3 stalls.
NOP = If(IF/ID.rA ≠ 0 && (IF/ID.rA==ID/Ex.Rd)
IF/ID.rA==Ex/M.Rd
IF/ID.rA==M/W.Rd))

or r6, r3, r4

sub r5, r3, r5

(MemWr=0, RegWr=0)

nop

add r3, r1, r2

WE=0

/stall

MemWr=0

 RegWr=0

NOP

s

sub

or

add

add r3, r1, r2

(MemWr=0, RegWr=0)

nop
NOP = \text{if}(IF/ID.rA \neq 0 \&\&$
\text{(IF/ID.rA)} = \text{ID/Ex.Rd)$
\text{(IF/ID.rA)} = \text{M/Ex.Rd}$
\text{(IF/ID.rA)} = \text{W/W.Rd)$

\text{add}\ r3, r1, r2

\text{sub}\ r5, r3, r5

\text{or}\ r6, r3, r4

\text{nop}$
\text{(MemWr=0)}$

\text{nop}$
\text{(RegWr=0)}$

\text{nop}$
\text{(MemWr=0)}$

\text{add}\ r3, r1, r2

\text{sub}\ r5, r3, r5

\text{or}\ r6, r3, r4

\text{nop}$
\text{(MemWr=0)}$

\text{nop}$
\text{(RegWr=0)}$

\text{nop}$
\text{(MemWr=0)}$

\text{add}\ r3, r1, r2

\text{sub}\ r5, r3, r5

\text{or}\ r6, r3, r4

\text{nop}$
\text{(MemWr=0)}$

\text{nop}$
\text{(RegWr=0)}$

\text{nop}$
\text{(MemWr=0)}$
NOP = If(IF/ID.rA ≠ 0 &&
(IF/ID.rA==ID/Ex.Rd
IF/ID.rA==Ex/M.Rd
(IF/ID.rA==M/W.Rd))

or r6, r3, r4

/sub stall

add r3, r1, r2

nop

sub r5, r3, r5

(MemWr=0
RegWr=0)

(MemWr=0
RegWr=0)

(MemWr=0
RegWr=0)
<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3 = 10</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r3, r1, r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3 = 20</td>
<td>IF</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>sub r5, r3, r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or r6, r3, r4</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>add r6, r3, r8</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stalling

How to stall an instruction in ID stage

• prevent IF/ID pipeline register update
  – stalls the ID stage instruction

• convert ID stage instr into nop for later stages
  – innocuous “bubble” passes through pipeline

• prevent PC update
  – stalls the next (IF stage) instruction
Data hazards occur when a operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.

Stalling—preventing a dependent instruction from advancing—is one way to resolve data hazards.

Stalling introduces NOPs (“bubbles”) into a pipeline. Introduce NOPs by (1) preventing the PC from updating, (2) preventing writes to IF/ID registers from changing, and (3) preventing writes to memory and register file.

*Bubbles in pipeline significantly decrease performance.
Next Goal

What to do if data hazard detected?

Options

• Nothing
  – Change the ISA to match implementation

• Stall
  – Pause current and subsequent instructions till safe

• Forward/bypass
  – Forward data value to where it is needed
Forwarding

Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register).

Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W→Ex)
- RegisterFile Bypass
Three types of forwarding/bypass
- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass
Three types of forwarding/bypass
• Forwarding from Ex/Mem registers to Ex stage (M→Ex)
• Forwarding from Mem/WB register to Ex stage (W → Ex)
• RegisterFile Bypass
Forwarding Datapath 1
Ex/MEM to EX Bypass
• EX needs ALU result that is still in MEM stage
  • Resolve:
    Add a bypass from EX/MEM.D to start of EX
  How to detect? Logic in Ex Stage:
    forward = (Ex/M.WE && EX/M.Rd != 0 &&
               ID/Ex.Ra == Ex/M.Rd)
    || (same for Rb)
Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass
Forwarding Datapath 1

ADD r3, r1, r2

SUB r5, r3, r1
Forwarding Datapath 2

Mem/WB to EX Bypass

• EX needs value being written by WB
• Resolve:
  Add bypass from WB final value to start of EX

How to detect? Logic in Ex Stage:

```
forward = (M/WB.WE && M/WB.Rd != 0 &&
           ID/Ex.Ra == M/WB.Rd &&
           not (ID/Ex.WE && Ex/M.Rd != 0 &&
                ID/Ex.Ra == Ex/M.Rd))
```

|| (same for Rb)

Check pg. 369
Three types of forwarding/bypass

• Forwarding from Ex/Mem registers to Ex stage (M→Ex)
• Forwarding from Mem/WB register to Ex stage (W → Ex)
• RegisterFile Bypass
Forwarding Datapath 2

add r3, r1, r2
sub r5, r3, r1
or r6, r3, r4
Register File Bypass

Register File Bypass

• Reading a value that is currently being written

Detect:

\[
((Ra == MEM/WB.Rd) \text{ or } (Rb == MEM/WB.Rd)) \text{ and } (WB \text{ is writing a register})
\]

Resolve:

Add a bypass around register file (WB to ID)

Better: (Hack) just negate register file clock
– writes happen at end of first half of each clock cycle
– reads happen during second half of each clock cycle
Three types of forwarding/bypass
• Forwarding from Ex/Mem registers to Ex stage (M→Ex)
• Forwarding from Mem/WB register to Ex stage (W → Ex)
• RegisterFile Bypass
<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>Ex</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r3, r1, r2</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>sub r5, r3, r1</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>or r6, r3, r4</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>add r6, r3, r8</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>
Forwarding Example

Clock cycle

<table>
<thead>
<tr>
<th>time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>r3 = 10</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r3, r1, r2</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3 = 20</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub r5, r3, r5</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or r6, r3, r4</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r6, r3, r8</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Forwarding Example 2

Clock cycle

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- add r3, r1, r2
- sub r5, r3, r4
- lw r6, 4(r3)
- or r5, r3, r5
- sw r6, 12(r3)
Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- Register File Bypass
Are we done yet?

add r3, r1, r2
lw(r4, 20(r8))
or r6, r3, r4
add r6, r3, r8

Memory “Load-Use Data Hazard”
Memory Load Data Hazard

What happens if data dependency after a load word instruction?

Memory Load Data Hazard

- Value not available until after the M stage
- So: next instruction can’t proceed if hazard detected
Memory Load Data Hazard

lw r4, 20(r8)

or r6, r3, r4

load-use stall

IF | ID | Ex | M  | W
---|----|----|----|---
    | IF | ID | Ex | M  | W

D B A

inst mem

data mem
Memory Load Data Hazard

`lw r4, 20(r8), or r6, r4, r1`

`load-use stall`

Table:

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>Ex</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
</tbody>
</table>

Diagram: A diagram showing the flow of data through the processor pipeline, with highlighted paths and stalls.
Memory Load Data Hazard

```
load-use stall

or r6, r4, r1
NOP
lw r4, 20(r8)
or r6, r4, r1
```

```
IF  ID  Ex  M  W

IF  ID  Ex  M  W
```

load-use stall
Memory Load Data Hazard

Instruction: `lw r4, 20(r8)`

Data Memory: `lw r4, 20(r8)`

Operation: `or r6, r4, r1`

NOP

Load-Use Stall
Stall =
If(ID/Ex.MemRead &&
  IF/ID.Ra == ID/Ex.Rd

Memory Load Data Hazard

Load Data Hazard

• Value not available until WB stage
• So: next instruction can’t proceed if hazard detected

Resolution:

• MIPS 2000/3000: one delay slot
  – ISA says results of loads are not available until one cycle later
  – Assembler inserts nop, or reorders to fill delay slot

• MIPS 4000 onwards: stall
  – But really, programmer/compiler reorders to avoid stalling in the load delay slot

For stall, how to detect? Logic in ID Stage

– Stall = ID/Ex.MemRead &&
  (IF/ID.Ra == ID/Ex.Rd | | IF/ID.Rb == ID/Ex.Rd)
Takeaway
Data hazards occur when a operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.

Stalling, preventing a dependent instruction from advancing, is one way to resolve data hazards. Stalling introduces NOPs (“bubbles”) into a pipeline. Introduce NOPs by (1) preventing the PC from updating, (2) preventing writes to IF/ID registers from changing, and (3) preventing writes to memory and register file. Bubbles (nops) in pipeline significantly decrease performance.

Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register). Better performance than stalling.
Data Hazard Recap

Delay Slot(s)

- Modify ISA to match implementation

Stall

- Pause current and all subsequent instructions

Forward/Bypass

- Try to steal correct value from elsewhere in pipeline
- Otherwise, fall back to stalling or require a delay slot
Hazards

3 kinds

• Structural hazards
  – Multiple instructions want to use same unit

• Data hazards
  – Results of instruction needed before

• Control hazards
  – Don’t know which side of branch to take
Control Hazards

What about branches?

A control hazard occurs if there is a control instruction (e.g. BEQ) and the program counter (PC) following the control instruction is not known until the control instruction computes if the branch should be taken.

e.g.

0x10: beq r1, r2, L
0x14: add r3, r0, r3
0x18: sub r5, r4, r6
0x1C: L: or r3, r2, r4
Control Hazards

Control Hazards

• instructions are fetched in stage 1 (IF)
• branch and jump decisions occur in stage 3 (EX)
• i.e. next PC is not known until 2 cycles after branch/jump

What happens to instr following a branch, if branch not taken?

A) Stall
B) Forward/Bypass
C) Zap/Flush
D) All the above
E) None of the above

e.g.

0x10: beq r1, r2, L
0x14: add r3, r0, r3
0x18: sub r5, r4, r6
0x1C: L: or r3, r2, r4
Control Hazards

Control Hazards

• instructions are fetched in stage 1 (IF)
• branch and jump decisions occur in stage 3 (EX)
• i.e. next PC is not known until 2 cycles after branch/jump

What happens to instr following a branch, if branch taken?

A) Stall
B) Forward/Bypass
C) Zap/Flush
D) All the above
E) None of the above

e.g.
0x10: beq r1, r2, L
0x14: add r3, r0, r3
0x18: sub r5, r4, r6
0x1C: L: or r3, r2, r4
Control Hazards

Control Hazards
- instructions are fetched in stage 1 (IF)
- branch and jump decisions occur in stage 3 (EX)
- i.e. next PC is not known until 2 cycles after branch/jump

What happens to instr following a branch, if branch taken?

Stall (+ Zap/Flush)
- prevent PC update
- clear IF/ID pipeline register
  - instruction just fetched might be wrong one, so convert to nop
- allow branch to continue into EX stage
Control Hazards

Diagram showing the control unit of a computer with blocks labeled 'inst mem', 'D', 'A', 'B', 'data mem', 'branch', 'calc', and 'decide branch'. The diagram illustrates the flow of instructions and data through the system.
Control Hazards

IF | ID | Ex | M | W
---|----|----|---|---
10: beq r1, r2, L | IF | ID | NOP | NOP | NOP
14: add r3, r0, r3 | IF | ID | NOP | NOP | NOP
18: sub r5, r4, r6 | IF | NOP | NOP | NOP | NOP
1C: L: or r3, r2, r4 | IF | ID | Ex | M | W

New PC = 1C
If branch Taken → Zap

branch calc decide branch
Control Hazards

New PC = 1C

10:  beq r1, r2, L

14:  add r3, r0, r3

18:  sub r5, r4, r6

1C:  L: or r3, r2, r4
Control Hazards

14: add r3, r0, r3
10: beq r1, r2, L
Control Hazards

18: sub r5, r4, r6
14: add r3, r0, r3
10: beq r1, r2, L
Control Hazards

inst mem

+4

PC

1C: or r3, r2, r4

NOP

NOP

10: beq r1, r2, L
Control Hazards

1C: or r3, r2, r4

NOP

NOP

10: beq r1, r2, L
Takeaway

Control hazards occur because the PC following a control instruction is not known until control instruction computes if branch should be taken or not.

If branch taken, then need to zap/flush instructions.

There is a performance penalty for branches: Need to stall, then may need to zap (flush) subsequent instructions that have already been fetched.
Next Goal

Can we reduce the cost of a control hazard?
Reduce the cost of control hazard?

Can we forward/bypass values for branches?

• We can move branch calc from EX to ID
• will require new bypasses into ID stage; or can just zap the second instruction

What happens to instructions following a branch, if branch taken?

• Still need to zap/flush instructions

Is there still a performance penalty for branches?

• Yes, need to stall, then may need to zap (flush) subsequent instructions that have already been fetched
Control Hazards
Control Hazards

inst mem

PC

+4

branch calc

decide branch

A

B

D

data mem
Control Hazards

10:  beq r1, r2, L
14:  add r3, r0, r3
18:  sub r5, r4, r6
1C:  or r3, r2, r4

IF  ID  Ex  M  W
---  ---  ---  ---  ---
IF  NOP NOP NOP NOP NOP

New PC = 1C
beq r1, r2,
add r3, r0,
sub r5, r4, r6
or r3, r2, r4

10: beq r1, r2, L
14: add r3, r0, r3
18: sub r5, r4, r6
1C: or r3, r2, r4

Control Hazards

inst mem
+4
PC

A
D
B

branch
calc
decide
branch

New PC = 1C

IF | ID | Ex | M | W
---|----|----|---|---
10: |     |     |   |   |
14: |     |     |   |   |
18: |     |     |   |   |
1C: |     |     |   |   |

If branch Taken → Zap

If

NOP
NOP
NOP
NOP

NOP
NOP
NOP
NOP

IF | ID | Ex | M | W
---|----|----|---|---
10: |     |     |   |   |
14: |     |     |   |   |
18: |     |     |   |   |
1C: |     |     |   |   |
Control Hazards

10: beq r1, r2, L
Control Hazards

14: add r3, r0, r3  
10: beq r1, r2, L
Control Hazards

1C: or r3, r2, r4  **NOP**

10: beq r1, r2, L
Control Hazards

20: inst mem

20 +4
calc

branch

decide branch

1C: or r3,r2,r4

NOP

10: beq r1, r2, L

data mem
Control Hazards

Control Hazards

- instructions are fetched in stage 1 (IF)
- branch and jump decisions occur in stage 3 (EX)
  i.e. next PC is not known until 2 \textit{cycles after} branch/jump
- Can optimize and move branch and jump decision to stage 2 (ID)
  i.e. next PC is not known until 1 \textit{cycles after} branch/jump

Stall (+ Zap)

- prevent PC update
- clear IF/ID pipeline register
  - instruction just fetched might be wrong one, so convert to nop
- allow branch to continue into EX stage
Control hazards occur because the PC following a control instruction is not known until control instruction computes if branch should be taken or not.

If branch taken, then need to zap/flush instructions. There still a performance penalty for branches: Need to stall, then may need to zap (flush) subsequent instructions that have already been fetched.

We can reduce cost of a control hazard by moving branch decision and calculation from Ex stage to ID stage. This reduces the cost from flushing two instructions to only flushing one.
Reduce cost of Control Hazards More

Delay Slot

• ISA says N instructions after branch/jump *always* executed
  – MIPS has 1 branch delay slot

  – i.e. Whether branch taken or not, instruction following branch is *always* executed
beq r1, r2, L
add r3, r0, r3
sub r5, r4, r6
1C: L: or r3, r2, r4

IF | ID | Ex | M | W
---|----|----|---|---

IF | ID | Ex | M | W
---|----|----|---|---

IF | ID | Ex | M | W
Delay Slot

If branch **not** taken next instr still exec’d

### Code Snippet

10: `beq r1, r2, L`

14: `add r3, r0, r3`

18: `sub r5, r4, r6`

1C: `or r3, r2, r4`
Control Hazards

Control Hazards
- instructions are fetched in stage 1 (IF)
- branch and jump decisions occur in stage 3 (EX)
  i.e. next PC is not known until 2 cycles after branch/jump
- Can optimize and move branch and jump decision to stage 2 (ID)
  i.e. next PC is not known until 1 cycles after branch/jump

Stall (+ Zap)
- prevent PC update
- clear IF/ID pipeline register
  - instruction just fetched might be wrong one, so convert to nop
- allow branch to continue into EX stage

Delay Slot
- ISA says N instructions after branch/jump always executed
  - MIPS has 1 branch delay slot
Takeaway

Control hazards occur because the PC following a control instruction is not known until control instruction computes if branch should be taken or not. If branch taken, then need to zap/flush instructions. There still a performance penalty for branches: Need to stall, then may need to zap (flush) subsequent instructions that have already been fetched.

We can reduce cost of a control hazard by moving branch decision and calculation from Ex stage to ID stage. This reduces the cost from flushing two instructions to only flushing one.

Delay Slots can potentially increase performance due to control hazards by putting a useful instruction in the delay slot since the instruction in the delay slot will always be executed. Requires software (compiler) to make use of delay slot. Put nop in delay slot if not able to put useful instruction in delay slot.
But in real processors...

Branch Predictors support Speculative Execution

- \textit{Guess} direction of the branch
  - Allow instructions to move through pipeline
  - Zap them later if wrong guess
- \textit{A must} for long pipelines
Summary

Control hazards

- Is branch taken or not?
- Performance penalty: stall and flush

Reduce cost of control hazards

- Move branch decision from Ex to ID
  - 2 nops to 1 nop
- Delay slot
  - Compiler puts useful work in delay slot. ISA level.
- Branch prediction
  - Correct. Great!
  - Wrong. Flush pipeline. Performance penalty
Hazards Summary

Data hazards
- register file reads occur in stage 2 (IF)
- register file writes occur in stage 5 (WB)
- next instructions may read values soon to be written

Control hazards
- branch instruction may change the PC in stage 3 (EX)
- next instructions have already started executing

Structural hazards
- resource contention
- so far: impossible because of ISA and pipeline design