Appendix C: Pipelining: Basic and Intermediate Concepts

Key ideas and simple pipeline (Section C.1)
Hazards (Sections C.2 and C.3)
  Structural hazards
  Data hazards
  Control hazards
Exceptions (Section C.4)
Multicycle operations (Section C.5)
Pipelining - Key Idea

Ideally,

\[ Time_{\text{pipeline}} = \frac{Time_{\text{sequential}}}{\text{Pipeline Depth}} \]

\[ Speedup = \frac{Time_{\text{sequential}}}{Time_{\text{pipeline}}} = \text{Pipeline Depth} \]
Consider an instruction that requires $n$ stages $s_1$, $s_2$, $\ldots$, $s_n$, taking time $t_1$, $t_2$, $\ldots$, $t_n$.

Let $T = \Sigma t_i$.

Without pipelining

Throughput =

Latency =

Speedup

With an n-stage pipeline

Throughput =

Latency =
Let $\Delta > 0$ be extra delay per stage
  e.g., latches
$\Delta$ limits the useful depth of a pipeline.

With an n-stage pipeline

$Throughput = \frac{1}{\Delta + \max t_i} < \frac{n}{T}$

$Latency = n \times (\Delta + \max t_i) \geq n\Delta + T$

$Speedup = \frac{\sum t_i}{\Delta + \max t_i} < n$
Example

Let $t_{1,2,3} = 8, 12, 10 \text{ ns} \text{ and } \Delta = 2 \text{ ns}$

Throughput =

Latency =

Speedup =
Practical Limit 3 - Hazards

\[
\text{Pipeline Speedup} = \frac{\text{Time}_{\text{sequential}}}{\text{Time}_{\text{pipeline}}} = \frac{\text{CPI}_{\text{sequential}}}{\text{CPI}_{\text{pipeline}}} \times \frac{\text{Cycle Time}_{\text{sequential}}}{\text{Cycle Time}_{\text{pipeline}}}
\]

If we ignore cycle time differences:

\[
\text{CPI}_{\text{ideal-pipeline}} = \frac{\text{CPI}_{\text{sequential}}}{\text{Pipeline Depth}}
\]

\[
\text{Pipeline Speedup} = \frac{\text{CPI}_{\text{ideal-pipeline}} \times \text{Pipeline Depth}}{\text{CPI}_{\text{ideal-pipeline}} + \text{Pipeline stall cycles}}
\]
Pipelining a Basic RISC ISA

MIPS ISA

Only loads and stores affect memory

  Base register + immediate offset = effective address

ALU operations

  Only access registers

  Two sources – two registers, or register and immediate

Branches and jumps

  Comparison between a register and zero

  Address = PC + offset
A Simple Five Stage RISC Pipeline

Pipeline Stages
- IF – Instruction Fetch
- ID – Instruction decode, register read, branch computation
- EX – Execution and Effective Address
- MEM – Memory Access
- WB – Writeback

<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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipelining really isn't this simple
Pipelining really isn't this simple
Hazards

Structural Hazards

Data Hazards

Control Hazards
Handling Hazards

Pipeline interlock logic

- Detects hazard and takes appropriate action

Simplest solution: stall

- Increases CPI
- Decreases performance

Other solutions are harder, but have better performance
Structural Hazards

When two \textit{different} instructions want to use the \textit{same} hardware resource in the \textit{same} cycle

Stall (cause bubble)
+ Low cost, simple
  Increases CPI
  Use for rare events
  E.g., ??

Duplicate Resource
+ Good performance
  Increases cost (and maybe cycle time for interconnect)
  Use for cheap resources
  E.g., ALU and PC adder
Structural Hazards, cont.

Pipeline Resource

+ Good performance
  Often complex to do
  Use when simple to do
  E.g., write & read registers every cycle

Structural hazards are avoided if each instruction uses a resource
  At most once
  Always in the same pipeline stage
  For one cycle
  \( \Rightarrow \) no cycle where two instructions use the same resource
## Structural Hazard Example

Loads/stores (MEM) use same memory port as instrn fetches (IF)

30% of all instructions are loads and stores

Assume $CPI_{old}$ is 1.5

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>← a load</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>**</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How much faster could a new machine with two memory ports be?
Data Hazards

When two different instructions use the same location, it must appear as if instructions execute one at a time and in the specified order

\[
\begin{align*}
i & \quad \text{ADD } r1, r2, \\
i+1 & \quad \text{SUB } r2, r1 \\
i+2 & \quad \text{OR } r1,--,
\end{align*}
\]

Read-After-Write (RAW, data-dependence)

A true dependence

MOST IMPORTANT

Write-After-Read (WAR, anti-dependence)

Write-After-Write (WAW, output-dependence)

NOT: Read-After-Read (RAR)
Example Read-After-Write Hazards

ADD r1,_,_  
IF  ID  EX  MEM  WB

SUB _, r1,_  
IF  ID  EX  MEM  WB

LW r1,_,_  
IF  ID  EX  MEM  WB

SUB _, r1,_  
IF  ID  EX  MEM  WB

SW r1,100(r0)  
IF  ID  EX  MEM  WB  
CORRECT!

LW r2,100(r0)  
IF  ID  EX  MEM  WB  
memory read

(Unless LW instrn is at address 100(r0))
RAW Solutions

Solutions must first detect RAW, and then ...

Stall

(Assumes registers written then read each cycle)

+ Low cost, simple
  
  Increases CPI (plus 2 per stall in 5 stage pipeline) 

Use for rare events
Bypass/Forward/ShortCircuit

Use data before it is in register

+ Reduces (avoids) stalls

More complex

Critical for common RAW hazards
Additional hardware

Muxes supply correct result to ALU

Additional control

Interlock logic must control muxes
Hybrid solution sometimes required:

One cycle bubble if result of load used by next instruction

Pipeline scheduling at compile time

Moves instructions to eliminate stalls
Pipeline Scheduling Example

Before:

\[ a = b + c; \]
\[ LW \ Rb,b \]
\[ LW \ Rc,c \]
\[ \xrightarrow{-\text{stall}} \]
\[ ADD \ Ra,Rb,Rc \]
\[ SW \ a, Ra \]
\[ d = e \ f; \]
\[ LW \ Re,e \]
\[ LW \ Rf,f \]
\[ \xrightarrow{-\text{stall}} \]
\[ SUB \ Rd,Re,Rf \]
\[ SW \ d, Rd \]

After:

\[ a = b + c; \]
\[ LW \ Rb,b \]
\[ LW \ Rc,c \]
\[ LW \ Re,e \]
\[ ADD \ Ra,Rb,Rc \]
\[ SW \ a, Ra \]
\[ d = e \ f; \]
\[ LW \ Rf,f \]
\[ SW \ a, Ra \]
\[ SUB \ Rd,Re,Rf \]
\[ SW \ d, Rd \]
Other Data Hazards

i ADD r1,r2,
i+1 SUB r2,,r1
i+2 OR r1,,

Write-After-Read (WAR, anti-dependence)

i MULT , (r2), r1 /* RX mult */
i+1 LW , (r1)+ /* autoincrement */

Write-After-Write (WAW, output-dependence)

i DIVF fr1, , /* slow */
i+1
i+2 ADDF fr1, , /* fast */
When an instruction affects which instructions are executed next -- branches, jumps, calls

```
i   BEQZ r1,#8
i+1  SUB ,, ...
i+8  OR ,, 
i+9  ADD ,, 
```

1 2 3 4 5 6 7 8 9
i  IF  ID  EX  MEM  WB
i+1  IF (aborted)
i+8      IF  ID  EX  MEM  WB
i+9      IF  ID  EX  MEM

Handling control hazards is very important
Handling Control Hazards

Branch Prediction
- Guess the direction of the branch
- Minimize penalty when right
- May increase penalty when wrong

Techniques
- Static – At compile time
- Dynamic – At run time

Static Techniques
- Predict NotTaken
- Predict Taken
- Delayed Branches

Dynamic techniques and more powerful static techniques later…
Predict NOT-TAKEN Always

NotTaken:

<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>i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Taken:

<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>i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
<td>(aborted)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+8</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+9</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Don't change machine state until branch outcome is known

Basic pipeline: State always changes late (WB)
Handling Control Hazards, cont.

Predict TAKEN Always

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
i     & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
i+8   & \text{‘IF’} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
i+9   & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
i+10  & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\end{array}
\]

Must know what address to fetch at BEFORE branch is decoded

Not practical for our basic pipeline
Delayed branch

Execute next instruction regardless (of whether branch is taken)

What do we execute in the DELAY SLOT?
Delay Slots

Fill from before branch
  When:
  Helps:

Fill from target
  When:
  Helps:

Fill from fall through
  When:
  Helps:
Delay Slots (Cont.)

Cancelling or nullifying branch

Instruction includes direction of prediction

Delay instruction squashed if wrong prediction

Allows second and third case of previous slide to be more aggressive
Comparison of Branch Schemes

Suppose 14% of all instructions are branches

Suppose 65% of all branches are taken

Suppose 50% of delay slots usefully filled

\[
\text{CPI penalty} = \% \text{ branches} \times \\
(\% \text{ taken} \times \text{ taken penalty} + \% \text{ not taken} \times \text{ not taken penalty})
\]

<table>
<thead>
<tr>
<th>Branch Scheme</th>
<th>Taken Penalty</th>
<th>Not-Taken Penalty</th>
<th>CPI Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Branch</td>
<td>1</td>
<td>1</td>
<td>.14</td>
</tr>
<tr>
<td>Not-Taken</td>
<td>1</td>
<td>0</td>
<td>.09</td>
</tr>
<tr>
<td>Taken0</td>
<td>0</td>
<td>1</td>
<td>.05</td>
</tr>
<tr>
<td>Taken1</td>
<td>1</td>
<td>1</td>
<td>.14</td>
</tr>
<tr>
<td>Delayed Branch</td>
<td>.5</td>
<td>.5</td>
<td>.07</td>
</tr>
</tbody>
</table>
MIPS R4000: 3 cycle branch penalty
  First cycle: cancelling delayed branch (cancel if not taken)
  Next two cycles: Predict not taken

Recent architectures:
  Because of deeper pipelines, delayed branches not very useful
  Processors rely more on hardware prediction (will see later) or may include both delayed and nondelayed branches
Interrupts

Interrupts (a.k.a. faults, exceptions, traps) often require

- Surprise jump
- Linking of return address
- Saving of PSW (including CCs)
- State change (e.g., to kernel mode)

Some examples

- Arithmetic overflow
- I/O device request
- O.S. call
- Page fault

Make pipelining hard
One Classification of Interrupts

1a. Synchronous
   function of program and memory state
   (e.g., arithmetic overflow, page fault)

1b. Asynchronous
   external device or hardware malfunction
   (printer ready, bus error)
Handling Interrupts

Precise Interrupts (Sequential Semantics)
  Complete instrns before offending one
  Squash (effects of) instrns after
  Save PC
  Force trap instrn into IF

Must handle simultaneous interrupts
  IF –
  ID –
  EX –
  MEM –
  WB –

Which interrupt should be handled first?
Example: Data Page Fault

<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>i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>← page fault (MEM)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>← squashed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>← squashed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>← squashed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+5</td>
<td>trap →</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+6</td>
<td>trap handler →</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Preceding instruction already complete
Squash succeeding instructions
Prevent from modifying state
‘Trap’ instruction jumps to trap handler
Hardware saves PC in IAR
Trap handler must save IAR
Example: Arithmetic Exception

<table>
<thead>
<tr>
<th>i</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>i+5</td>
<td>trap</td>
<td>→</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>i+6</td>
<td>trap handler</td>
<td>→</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Let preceding instructions complete

Squash succeeding instruction
Example: Illegal Opcode

```
  1  2  3  4  5  6  7  8
i    IF  ID  EX  MEM  WB
i+1  IF  ID  EX  MEM  WB
i+2  IF  ID  EX  MEM  WB
i+3  IF  ID  EX  MEM  WB <- ill. op (ID)
i+4  IF  ID  EX  MEM  WB <- squash
i+5  trap -> IF  ID  EX  MEM  WB
i+6  trap handler -> IF  ID  EX  MEM  WB
```

Let preceding instructions complete

Squash succeeding instruction
Example: Out-of-order Interrupts

| i   | IF | ID  | EX  | MEM | WB   | –— | page fault (MEM) |
| i+1 | IF | ID  | EX  | MEM | WB   | –— | page fault (IF)  |
| i+2 | IF | ID  | EX  | MEM | WB   |     |                  |
| i+3 | IF | ID  | EX  | MEM | WB   |     |                  |

Which page fault should we take?

For precise interrupts – Post interrupts on a status vector associated with instruction, disable later writes in pipeline
- Check interrupt bit on entering WB
- Longer latency

For imprecise interrupts – Handle immediately
- Interrupts may occur in different order than on a sequential machine
- May cause implementation headaches
Other complications

- Odd bits of state (e.g., CCs)
- Early writes (e.g., autoincrement)
- Out of order execution

Interrupts come at random times

- The frequent case isn't everything
- The rare case MUST work correctly
Multicycle Operations

Not all operations complete in one cycle

Floating point arithmetic is inherently slower than integer arithmetic

2 to 4 cycles for multiply or add
20 to 50 cycles for divide

Extend basic 5-stage pipeline

EX stage may repeat multiple times
Multiple function units
Not pipelined for now
Handling Multicycle Operations

Four Functional Units
- EX: Integer unit
- E*: FP/integer multiplier
- E+: FP adder
- E/: FP/integer divider

Assume
- EX takes one cycle & all FP units take 4
- Separate integer and FP registers
- All FP arithmetic from FP registers

Worry about
- Structural hazards
- RAW hazards & forwarding
- WAR & WAW between integer & FP ops
## Simple Multicycle Example

<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>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fp*</td>
<td>IF</td>
<td>ID</td>
<td>E*</td>
<td>E*</td>
<td>E*</td>
<td>E*</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB? (1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fp/</td>
<td>IF</td>
<td>ID</td>
<td>E/</td>
<td>E/</td>
<td>E/</td>
<td>E/</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>**</td>
<td>MEM</td>
<td>WB (2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fp/</td>
<td>(3) IF</td>
<td>ID</td>
<td>**</td>
<td>**</td>
<td>E/</td>
<td>E/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>(4) IF</td>
<td>**</td>
<td>**</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes**

(1) WAW possible only if?
(2) Stall forced by?
(3) Stall forced by?
(4) Stall forced by?
FP Instruction Issue

Check for RAW data hazard (in ID)
   Wait until source registers are not used as destinations by instructions in EX that will not be available when needed

Check for forwarding
   Bypass data from other stages, if necessary

Check for structural hazard in function unit
   Wait until function unit is free (in ID)

Check for structural hazard in MEM / WB
   Instructions stall in ID
   Instructions stall before MEM
      Static priority (e.g., FU with longest latency)
Check for WAW hazards

DIVF F0, F2, F4
SUBF F0, F8, F10

SUBF completes first
  (1) Stall SUBF
  (2) Abort DIVF's WB

WAR hazards?
More Multicycle Operations

Problems with Interrupts

DIVF F0, F2, F4
ADDF F2, F8, F10
SUBF F6, F4, F10

ADDF and SUBF complete before DIVF

Outoforder completion

Possible imprecise interrupt

What happens if DIVF generates an exception after ADDF and SUBF complete??

We'll discuss solutions later