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,
\[ \text{Speedup} = \frac{\text{Time}_{\text{sequential}}}{\text{Pipeline Depth}} \]
\[ \text{Throughput} = \frac{\text{Time}_{\text{sequential}}}{\text{Pipeline Depth}} \]

Practical Limit 1 – Unbalanced Stages
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
With an \( n \)-stage pipeline

Throughput
Latency
Speedup

Practical Limit 2 - Overheads
Let \( \Delta > 0 \) be extra delay per stage
e.g., latches
\( \Delta \) limits the useful depth of a pipeline.

With an \( n \)-stage pipeline

\[ \text{Throughput} = \frac{1}{\Delta + \max t_i} < \frac{n}{T} \]
\[ \text{Latency} = n \times (\Delta + \max t_i) \geq n\Delta + T \]
\[ \text{Speedup} = \frac{\Sigma t_i}{\Delta + \max t_i} < n \]
Example

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

Assumptions:

- 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
  - Address = PC + offset
  - Comparison between a register and zero

The last assumption is different from the 6th edition of the text and results in a slightly different pipeline. We will discuss reasons and implications in class.

A Simple Five Stage RISC Pipeline

<table>
<thead>
<tr>
<th>Pipeline Stages</th>
<th>IF – Instruction Fetch</th>
<th>ID – Instruction decode, register read, branch computation</th>
<th>EX – Execution and Effective Address</th>
<th>MEM – Memory Access</th>
<th>WB – Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>IF ID EX MEM WB</td>
<td>i+1 IF ID EX MEM WB</td>
<td>i+2 IF ID EX MEM WB</td>
<td>i+3 IF ID EX MEM WB</td>
<td>i+4 IF ID EX MEM WB</td>
</tr>
<tr>
<td>i+1</td>
<td>IF ID EX MEM WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>IF ID EX MEM WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>IF ID EX MEM WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td>IF ID EX MEM WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Naive Pipeline Implementation

Figure C.28 of 5th edition

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 different instructions want to use the same hardware resource in the 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
  - (⇒ 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

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<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>**</td>
<td>IF</td>
<td>ID</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>
</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

1. ADD r1, r2,
2. SUB r2, r1
3. OR r1, r1,

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

\[
\begin{array}{cccccc}
& & & & & \\
\text{ADD } r_1 & & & & & \text{NOT OK!} \\
& & & & & \\
\text{SUB } r_1 & & & & & \\
& & & & & \\
\text{LW } r_1 & & & & & \text{NOT OK!} \\
& & & & & \\
\text{SW } r_1, 100(r_0) & & & & & \text{CORRECT!} \\
& & & & & \\
\text{LW } d_2, 100(r_0) & & & & & \\
& & & & & \\
\end{array}
\]

(Unless LW instrn is at address 100(r_0))
RAW Solutions

Solutions must first detect RAW, and then ...

Stall

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD r1, _, _</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB _, r1, _</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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/Short Circuit

Use data before it is in register

+ Reduces (avoids) stalls
  * More complex
  * Critical for common RAW hazards

Bypass, cont.

Figure C.27

5th edition

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:**

\[
\begin{align*}
a &= b + c; \\
&\text{LW } Rb, b \\
&\text{LW } Rc, c \\
&\text{ADD } Ra, Rb, Rc \\
&\text{SW } a, Ra \\
\end{align*}
\]

\[
\begin{align*}
d &= e - f; \\
&\text{LW } Re, e \\
&\text{LW } Rf, f \\
&\text{SUB } Rd, Re, Rf \\
&\text{SW } d, Rd \\
\end{align*}
\]

**After:**

\[
\begin{align*}
a &= b + c; \\
&\text{LW } Rb, b \\
&\text{LW } Rc, c \\
&\text{ADD } Ra, Rb, Rc \\
&\text{SW } a, Ra \\
\end{align*}
\]

\[
\begin{align*}
d &= e - f; \\
&\text{LW } Re, e \\
&\text{LW } Rf, f \\
&\text{SUB } Rd, Re, Rf \\
&\text{SW } d, Rd \\
\end{align*}
\]

### Other Data Hazards

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

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

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

\[
\begin{align*}
i &\text{ MULT }, (r2), r1 /* RX mult */ \\
i+1 &\text{ LW }, (r1)+ /* autoincrement */ \\
i+2 &\text{ ADD } r1, , /* fast */
\end{align*}
\]

### Control Hazards

When an instruction affects which instructions are executed next -- branches, jumps, calls

\[
\begin{align*}
i &\text{ BEQZ } r1, #8 \\
i+1 &\text{ SUB }, .. \\
i+8 &\text{ OR }, .. \\
i+9 &\text{ ADD }, \\
\end{align*}
\]

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...
Handling Control Hazards, cont.

Predict NOT-TAKEN Always

Not Taken:

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

Taken:

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

Don't change machine state until branch outcome is known

Basic pipeline: State always changes late (WB)

Handling Control Hazards, cont.

Predict TAKEN Always

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

Must know what address to fetch at BEFORE branch is decoded

Not practical for our basic pipeline

Handling Control Hazards, cont.

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

CPI\text{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>

Real Processors

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?

---

**Interrupts, cont.**

Example: Data Page Fault

<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 &lt;- page fault (MEM)</td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM WB &lt;- squash</td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM WB &lt;- squash</td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td>trap -&gt; IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM WB</td>
<td></td>
</tr>
<tr>
<td>i+5</td>
<td>trap handler -&gt; IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM WB</td>
<td></td>
</tr>
<tr>
<td>i+6</td>
<td>trap handler -&gt; IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM WB</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

---

**Interrupts, cont.**

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 WB</td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM WB &lt;- Exception (EX)</td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM WB &lt;- squash</td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM WB &lt;- squash</td>
<td></td>
</tr>
<tr>
<td>i+5</td>
<td>trap -&gt; IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM WB</td>
<td></td>
</tr>
<tr>
<td>i+6</td>
<td>trap handler -&gt; IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM WB</td>
<td></td>
</tr>
</tbody>
</table>

Let preceding instructions complete
- Squash succeeding instruction
**Interrupts, cont.**

Example: Illegal Opcode

<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 &lt;- ill. op (ID)</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 &lt;- squash</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+5</td>
<td>trap -&gt; IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+6</td>
<td>trap handler -&gt; IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let preceding instructions complete
Squash succeeding instruction

---

**Interrupts, cont.**

Example: Out-of-order Interrupts

<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 &lt;- page fault (MEM)</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 &lt;- page fault (IF)</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>
<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>
</tr>
</tbody>
</table>

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

---

**Interrupts, cont.**

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

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)

FP Instruction Issue (Cont.)

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

Out-of-order completion
   Possible imprecise interrupt

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

We'll discuss solutions later.