Hazards in a Pipeline Processor

Pipeline is good but you need be careful.

Pipelining: Its Natural!

- Laundry Example
  - Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - “Folder” takes 20 minutes
Sequential Laundry

- Sequential laundry takes 6 hours for 4 loads.
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP

- Pipelined laundry takes 3.5 hours for 4 loads.
Pipelining Lessons

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload.
- Pipeline rate limited by slowest pipeline stage.
- Multiple tasks operating simultaneously.
- Potential speedup = Number of pipe stages.
- Unbalanced lengths of pipe stages reduces speedup.
- Time to “fill” pipeline and time to “drain” it reduces speedup.

Why not pipeline for instructions?

[Diagram showing pipelining process and potential speedup calculations]
Review: Single Cycle, Multiple Cycle, vs. Pipeline

Single Cycle Implementation:

Cycle 1
Load

Cycle 2
Store

Waste

Multiple Cycle Implementation:

Cycle 1
Load

Cycle 2
Ifetch

Cycle 3
Reg

Cycle 4
Exec

Cycle 5
Mem

Cycle 6
Wr

Cycle 7
Ifetch

Cycle 8
Reg

Cycle 9
Exec

Cycle 10
Mem

Pipeline Implementation:

Load

Ifetch

Reg

Exec

Mem

Wr

Store

Ifetch

Reg

Exec

Mem

Wr

R-type

Ifetch

Reg

Exec

Mem

Wr

Review: A Pipelined Datapath
Review: Pipeline Control “Data Stationary Control”

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later

Review: Pipeline Summary

- Pipeline Processor:
  - Natural enhancement of the multiple clock cycle processor
  - Each functional unit can only be used once per instruction
  - If a instruction is going to use a functional unit:
    - it must use it at the same stage as all other instructions
- Pipeline Control:
  - Each stage’s control signal depends ONLY on the instruction that is currently in that stage
Its not that easy for computers

- **Limits to pipelining**: Hazards prevent next instruction from executing during its designated clock cycle

- **Structural hazards** (hardware resource conflicts): HW cannot support this combination of instructions (*single person to fold and put clothes away*)

- **Data hazards** (data dependencies): Instruction depends on result of prior instruction still in the pipeline (*missing sock*)

- **Control hazards** (changes in program flow): Pipelining of branches & other instructions *stall* the pipeline until the hazard (*“bubbles”*) in the pipeline disappear.

\[
\text{Hazards} = \text{Such Hazardous events}
\]

---

**Single Memory is a Structural Hazard**

Time (clock cycles)

<table>
<thead>
<tr>
<th>Instr Order</th>
<th>Instr 1</th>
<th>Instr 2</th>
<th>Instr 3</th>
<th>Instr 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Mem</td>
<td>Reg</td>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
</tr>
</tbody>
</table>
Option 1: Stall to resolve Memory Structural Hazard

Option 2: Duplicate to Resolve Structural Hazard

- Separate Instruction Cache (Im) & Data Cache (Dm)
**Data Hazard on r1**

\[
\begin{align*}
\text{add} & \ r1, r2, r3 \\
\text{sub} & \ r4, r1, r3 \\
\text{and} & \ r6, r1, r7 \\
\text{or} & \ r8, r1, r9 \\
\text{xor} & \ r10, r1, r11
\end{align*}
\]

Data Hazard on r1:

(Fig. 6.36, page 477)

- Dependencies backwards in time are hazards

Shading on right/left half means read/write on that element.
Option 1: HW Stalls to Resolve Data Hazard

- Dependencies backwards in time are hazards

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
</tr>
<tr>
<td>ID/RF</td>
</tr>
<tr>
<td>EX</td>
</tr>
<tr>
<td>MEM</td>
</tr>
<tr>
<td>WB</td>
</tr>
</tbody>
</table>

Instr. Order:

- add \( r1, r2, r3 \)
- sub \( r4, r1, r3 \)
- and \( r6, r1, r7 \)
- or \( r8, r1, r9 \)
- xor \( r10, r1, r11 \)

- The Control logic can be complex

But recall use of “Data Stationary Control”

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later
**Option 1: How HW really stalls pipeline**

- HW doesn’t change PC ➞ keeps fetching same instruction & sets control signals to benign values (0)

```
Time (clock cycles)
IF ID/RF EX MEM WB
```

- add \( r_1, r_2, r_3 \)
  - stall
  - stall
  - stall
  - sub \( r_4, r_1, r_3 \)
  - and \( r_6, r_1, r_7 \)

**Option 2: SW inserts independent instructions**

- Worst case inserts NOP instructions

```
Time (clock cycles)
IF ID/RF EX MEM WB
```

- add \( r_1, r_2, r_3 \)
  - nop
  - nop
  - nop
  - sub \( r_4, r_1, r_3 \)
  - and \( r_6, r_1, r_7 \)
Option 3 Insight: Data is available!  (Fig. 6.37, page 481)

- Pipeline registers already contain needed data

HW Change for “Forwarding” (Bypassing): (Fig 6.38-40, p 482-484)

- Increase multiplexors to add paths from pipeline registers
- Assumes register read during write gets new value (otherwise more results to be forwarded)
**From Last Lecture: The Delay Load Phenomenon**

Although Load is fetched during Cycle 1:
- The data is NOT written into the Reg File until the end of Cycle 5
- We cannot read this value from the Reg File until Cycle 6
- 3-instruction delay before the load take effect

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plus 1</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plus 2</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plus 3</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plus 4</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Forwarding reduces Data Hazard to 1 cycle:**
(Fig. 6.44, p489)

- \( \text{lw } r1, 0(r2) \)
- \( \text{sub } r4, r1, r6 \)
- \( \text{and } r6, r1, r7 \)
- \( \text{or } r8, r1, r9 \)
Option 1: HW Stalls to Resolve Data Hazard

- "Interlock": checks for hazard & stalls

```
lw r1, 0(r2)
stall
sub r4,r1,r3
and r6,r1,r7
or r8,r1,r9
```

Option 2: SW inserts independent instructions

- Worst case inserts NOP instructions
- MIPS I solution: No HW checking

```
lw r1, 0(r2)
nop
sub r4,r1,r3
and r6,r1,r7
or r8,r1,r9
```
Software Scheduling to Avoid Load Hazards

Try producing fast code for
\[ a = b + c; \]
\[ d = e - f; \]
assuming \( a, b, c, d, e, \) and \( f \)
in memory.

Slow code:

```
LW Rb,b
LW Rc,c
ADD Ra,Rb,Rc
SW a,Ra
LW Re,e
LW Rf,f
SUB Rd,Re,Rf
SW d,Rd
```

Fast code:

```
LW Rb,b
LW Rc,c
ADD Ra,Rb,Rc
LW Re,e
LW Rf,f
SUB Rd,Re,Rf
SW d,Rd
```

Compiler Avoiding Load Stalls:

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Scheduled</th>
<th>Unscheduled</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>25%</td>
<td>31%</td>
</tr>
<tr>
<td>spice</td>
<td>14%</td>
<td>42%</td>
</tr>
<tr>
<td>tex</td>
<td>25%</td>
<td>65%</td>
</tr>
</tbody>
</table>

% loads stalling pipeline
From Last Lecture: The Delay Branch Phenomenon

Although Beq is fetched during Cycle 4:
- Target address is NOT written into the PC until the end of Cycle 7
- Branch’s target is NOT fetched until Cycle 8
- 3-instruction delay before the branch take effect

Control Hazard on Branches: 3 stage stall (Fig. 6.50, P497)
Branch Stall Impact

- If CPI = 1, 30% branch, Stall 3 cycles \( \implies \) new CPI = 1.9!
- 2 part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests = 0 or \( \neq 0 \)
- Solution Option 1:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch vs. 3

Option 1: move HW forward to reduce branch delay
Branch Delay now 1 clock cycle

Option 2: Define Branch as Delayed

- Worst case, Software inserts NOP into branch delay
- Where to get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: only valuable when branch
  - From fall through: only valuable when don’t branch
- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - about 50% (60% x 80%) of slots usefully filled
When is pipelining hard? (1)

- **Interrupts:** 5 instructions executing in 5 stage pipeline
  - How to stop the pipeline?
  - Restart?
  - Who caused the interrupt?

<table>
<thead>
<tr>
<th>Stage</th>
<th>Problem interrupts occurring</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on instruction fetch; misaligned memory access; memory-protection violation</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined or illegal opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic interrupt</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data fetch; misaligned memory access; memory-protection violation</td>
</tr>
</tbody>
</table>

- Load with data page fault, Add with instruction page fault?
- Solution 1: interrupt vector/instruction 2: interrupt ASAP, restart everything incomplete

When is pipelining hard? (2)

- **Complex Addressing Modes and Instructions**
  - Address modes: Auto increment causes register change during instruction execution
    - Interrupts?
      - Now worry about write hazards since write no longer last stage
        - Write After Read (WAR): Write occurs before independent read
        - Write After Write (WAW): Writes occur in wrong order, leaving wrong result in registers
        - (Previous data hazard called RAW, for Read After Write)
  - Memory-memory Move instructions
    - Multiple page faults
    - make progress?
When is pipelining hard? (3)

- **Floating Point:** long execution time
  - Also, may pipeline FP execution unit so that can initiate new instructions without waiting full latency

<table>
<thead>
<tr>
<th>FP Instruction</th>
<th>Latency</th>
<th>Initiation Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, Subtract</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Multiply</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Divide</td>
<td>36</td>
<td>35</td>
</tr>
<tr>
<td>Square root</td>
<td>112</td>
<td>111</td>
</tr>
<tr>
<td>Negate</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Absolute value</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>FP compare</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

- Divide, Square Root take -10X to -30X longer than Add
  - Exceptions?
  - Adds WAR and WAW hazards since pipelines are no longer same length

Data Hazards

- **Read After Write (RAW):**
  - Attempting to read a value that hasn't been written yet. This is the most common type, and can be overcome by forwarding.

- **Write after write (WAW):**
  - Writing a value before a preceding write has completed. This can only happen in complex pipes that allow instructions to proceed out of order, or that have multiple write-back stages (mostly CISC), or that have multiple pipes that can write.

- **Write after read (WAR):**
  - Writing a value before a preceding read has completed. These also require a complex pipeline that can sometimes write in an early stage, and read in a later stage. It is also possible when multiple pipelines or out-of-order issue are employed.

- **Read after read (RAR) does not produce a hazard.**
Hazard Detection

Suppose instruction \( i \) is about to be issued and a predecessor instruction \( j \) is in the instruction pipeline.

\[
\begin{align*}
Rregs(i) &= \text{Registers read by instruction } i \\
Wregs(i) &= \text{Registers written by instruction } i
\end{align*}
\]

- A RAW hazard exists on register \( \rho \) if \( \exists \rho, \rho \in Rregs(i) \cap Wregs(j) \)
  - Keep a record of pending writes (for inst's in the pipe) and compare with operand regs of current instruction.
  - When instruction issues, reserve its result register.
  - When one operation completes, remove its write reservation.

- A WAW hazard exists on register \( \rho \) if \( \exists \rho, \rho \in Wregs(i) \cap Wregs(j) \)

- A WAR hazard exists on register \( \rho \) if \( \exists \rho, \rho \in Wregs(i) \cap Rregs(j) \)

Avoiding Data Hazards by Design

Suppose instructions are executed in a pipelined fashion such that instructions are initiated in order.

- **WAW avoidance**: if writes to a particular resource (e.g., reg) are performed in the same stage for all instructions, then no WAW hazards occur.

  proof: writes are in the same time sequence as instructions.

\[
\begin{array}{c|c|c|c|c|}
\hline
I & R/D & E & W \\
\hline
I & R/D & E & W \\
I & R/D & E & W \\
\hline
\end{array}
\]

- **WAR avoidance**: if in all instructions reads of a resource occur at an earlier stage than writes to that resource occur in any instruction, then no WAR hazards occur.

  proof: A successor instruction must issue later, hence it will perform writes only after all reads for the current instruction.
First Generation RISC Pipelines

- All instructions follow same pipeline order ("static schedule").
- Register write in last stage
  - Avoid WAW hazards
- All register reads performed in first stage after issue.
  - Avoid WAR hazards
- Memory access in stage 4
  - Avoid all memory hazards
- Control hazards resolved by delayed branch (with fast path)
- RAW hazards resolved by bypass, except on load results which are resolved by fiat (delayed load).

Substantial pipelining with very little cost or complexity.
Machine organization is (slightly) exposed!
Relies very heavily on "hit assumption" of memory accesses in cache

Review: Summary of Pipelining Basics

- Speed Up $\uparrow$ Pipeline Depth; if ideal CPI is 1, then:
  \[
  \text{Speedup} = \frac{\text{Pipeline depth}}{\text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}
  \]
- Hazards limit performance on computers:
  - structural: need more HW resources
  - data: need forwarding, compiler scheduling
  - control: early evaluation & PC, delayed branch, prediction
- Increasing length of pipe increases impact of hazards since pipelining helps instruction bandwidth, not latency
- Compilers key to reducing cost of data and control hazards
  - load delay slots
  - branch delay slots
- Exceptions, Instruction Set, FP makes pipelining harder
- Longer pipelines $\Rightarrow$ Branch prediction, more instruction parallelism?