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

- Pipelined laundry takes 3.5 hours for 4 loads
Pipeline 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 pipe stages.
- Unbalanced lengths of pipe stages reduces speedup.
- Time to “fill” pipeline and time to “drain” it reduces speedup.
Computer Pipeline

• Execute billions of instructions, so *throughput* is what matters

• What is desirable in instruction sets for pipelining?
  – Variable length instructions vs. all instructions same length?
  – Memory operands part of any operation vs. memory operands only in loads or stores?
  – Register operand many places in instruction format vs. registers located in same place?
A "Typical" RISC

- 32-bit fixed format instruction (3 formats)
- Memory access only via load/store instructions
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction; registers in same place
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3
Example: MIPS
(Note register location)

Register-Register

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Opx</td>
</tr>
</tbody>
</table>
```

Register-Immediate

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
</tbody>
</table>
```

Branch

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2/Opx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
</tbody>
</table>
```

Jump / Call

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td></td>
<td></td>
<td>target</td>
</tr>
</tbody>
</table>
```
5 Steps of MIPS Datapath

Instruction Fetch
- Next PC
- Address
- Memory
- Adder

Instr. Decode Reg. Fetch
- Next SEQ PC
- Inst
- Reg File
- RS1
- RS2
- RD
- Imm
- Sign Extend

Execute Addr. Calc
- Zero
- ALU
- MUX

Memory Access
- Data Memory
- LMD
- MUX

Write Back
- WB Data
5 Steps of MIPS Datapath

- Instruction Fetch
- Instr. Decode Reg. Fetch
- Execute Addr. Calc
- Memory Access
- Write Back

- Data stationary control
  - local decode for each instruction phase / pipeline stage
Visualizing Pipelining

Cycle 1: Ifetch, Reg
Cycle 2: Reg, ALU
Cycle 3: ALU, DMem
Cycle 4: DMem, Reg
Cycle 5: Reg, ALU
Cycle 6: ALU, DMem
Cycle 7: DMem, Reg
Its Not That Easy for Computers

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - **Structural hazards**: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
One Memory Port/Structural Hazards

Time (clock cycles)

Cycle 1: Load, Instr 1
Cycle 2: Instr 1
Cycle 3: Instr 2
Cycle 4: Instr 3
Cycle 5: Instr 4
Cycle 6: Instr 4
Cycle 7: Instr 4

Instr. Order

Load
Instr 1
Instr 2
Instr 3
Instr 4
Resolving structural hazards

• Define: attempt to use same hardware for two different things at the same time

• Solution 1: Wait
  ⇒ must detect the hazard
  ⇒ must have mechanism to stall

• Solution 2: Throw more hardware at the problem
One Memory Port/Structural Hazards

Time (clock cycles)

Cycle 1  Cycle 2  Cycle 3  Cycle 4  Cycle 5  Cycle 6  Cycle 7

Load

Instr 1

Instr 2

Stall

Instr 3
Eliminating Structural Hazards at Design Time
Role of Instruction Set Design in Structural Hazard Resolution

• Simple to determine the sequence of resources used by an instruction
  – opcode tells it all

• Uniformity in the resource usage

• Compare MIPS to IA32?

• MIPS approach =&gt; all instructions flow through same 5-stage pipeling
Data Hazard on R1

Time (clock cycles)

Instr. Order

add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11
Three Generic Data Hazards

• **Read After Write (RAW)**
  Instr\(_j\) tries to read operand before Instr\(_i\) writes it
  
  \[
  \begin{align*}
  I &: \text{add } r1, r2, r3 \\
  J &: \text{sub } r4, r1, r3
  \end{align*}
  \]

• Caused by a “**Dependence**” (in compiler nomenclature). This hazard results from an actual need for communication.
Three Generic Data Hazards

- Write After Read (WAR)
  Instr\textsubscript{j} writes operand \textit{before} Instr\textsubscript{i} reads it

  \[\begin{align*}
  I: & \text{ sub } r4, r1, r3 \\
  J: & \text{ add } r1, r2, r3 \\
  K: & \text{ mul } r6, r1, r7
  \end{align*}\]

- Called an “anti-dependence” by compiler writers.
  This results from reuse of the name “r1”.

- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5
Three Generic Data Hazards

- **Write After Write (WAW)**
  Instr$_j$ writes operand *before* Instr$_i$ writes it.

  
  I:  sub  $r1$,r4,$r3$
  J:  add  $r1$,r2,$r3$
  K:  mul  $r6$,$r1$,$r7$

- Called an “*output dependence*” by compiler writers
  This also results from the reuse of name “*r1*”.

- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5

- Will see WAR and WAW in later more complicated pipes
Forwarding to Avoid Data Hazard

Time (clock cycles)

Instruction Order

add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11
HW Change for Forwarding
Data Hazard Even with Forwarding

Time (clock cycles)

Instr. Order

lw r1, 0(r2)
sub r4, r1, r6
and r6, r1, r7
or r8, r1, r9
Resolving this load hazard

• Adding hardware? ... not
• Detection?
• Compilation techniques?

• What is the cost of load delays?
Data Hazard Even with Forwarding

\[ \text{lw } r1, 0(r2) \]
\[ \text{sub } r4, r1, r6 \]
\[ \text{and } r6, r1, r7 \]
\[ \text{or } r8, r1, r9 \]

Time (clock cycles)
Try producing fast code for

\[ a = b + c; \]
\[ d = e - f; \]

assuming \( a, b, c, d, e, \) and \( f \) in memory.

<table>
<thead>
<tr>
<th>Slow code:</th>
<th>Fast code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW Rb, b</td>
<td>LW Rb, b</td>
</tr>
<tr>
<td>LW Rc, c</td>
<td>LW Rc, c</td>
</tr>
<tr>
<td>ADD Ra, Rb, Rc</td>
<td>LW Re, e</td>
</tr>
<tr>
<td>SW a, Ra</td>
<td>ADD Ra, Rb, Rc</td>
</tr>
<tr>
<td>LW Re, e</td>
<td>LW Rf, f</td>
</tr>
<tr>
<td>LW Rf, f</td>
<td>SW a, Ra</td>
</tr>
<tr>
<td>SUB Rd, Re, Rf</td>
<td>SW Rd, Re, Rf</td>
</tr>
<tr>
<td>SW d, Rd</td>
<td>SW d, Rd</td>
</tr>
</tbody>
</table>
Control Hazard on Branches
Three Stage Stall

10: beq r1, r3, 36
14: and r2, r3, r5
18: or r6, r1, r7
22: add r8, r1, r9
36: xor r10, r1, r11
Example: Branch Stall Impact

- If 30% branch, Stall 3 cycles significant
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3
• Data stationary control
  – local decode for each instruction phase / pipeline stage
Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken
   - Execute successor instructions in sequence
   - “Squash” instructions in pipeline if branch actually taken
   - Advantage of late pipeline state update
   - 47% MIPS branches not taken on average
   - PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken
   - 53% MIPS branches taken on average
   - But haven’t calculated branch target address in MIPS
     • MIPS still incurs 1 cycle branch penalty
     • Other machines: branch target known before outcome
#4: Delayed Branch

- Define branch to take place \textit{AFTER} a following instruction

\[
\text{branch instruction} \quad \text{sequential successor}_1 \quad \text{sequential successor}_2 \quad \ldots \quad \text{sequential successor}_n \quad \text{branch target if taken}
\]

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this
Scheduling Branch Delay Slots (Fig A.14)

A. From before branch

```plaintext
add $1,$2,$3
if $2=0$ then
  delay slot
becomes
if $2=0$ then
  add $1,$2,$3
```

B. From branch target

```plaintext
sub $4,$5,$6
if $1=0$ then
  delay slot
add $1,$2,$3
if $1=0$ then
  delay slot
sub $4,$5,$6
```

C. From fall through

```plaintext
add $1,$2,$3
if $1=0$ then
  delay slot
sub $4,$5,$6
```

• A is the best choice, fills delay slot & reduces instruction count (IC)
• In B, the `sub` instruction may need to be copied, increasing IC
• In B and C, must be okay to execute `sub` when branch fails
Delayed 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

• Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
  – Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  – Growth in available transistors has made dynamic approaches relatively cheaper
Delayed Branch

• Where to get instructions to fill branch delay slot?
  – Before branch instruction
  – From the target address: only valuable when branch taken
  – From fall through: only valuable when branch not taken
  – Canceling branches allow more slots to be filled

• 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

• Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)
Problems with Pipelining

- **Exception**: An unusual event happens to an instruction during its execution
  - Examples: divide by zero, undefined opcode

- **Interrupt**: Hardware signal to switch the processor to a new instruction stream
  - Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)

- **Problem**: It must appear that the exception or interrupt must appear between 2 instructions ($I_i$ and $I_{i+1}$)
  - The effect of all instructions up to and including $I_i$ is totalling complete
  - No effect of any instruction after $I_i$ can take place

- **The interrupt (exception) handler either aborts program or restarts at instruction $I_{i+1}$**
Key observation: architected state only change in memory and register write stages.
Recall: Speed Up Equation for Pipelining

\[ \text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst} \]

\[
\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}} \]

For simple RISC pipeline, CPI = 1:

\[
\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}} \]
Example: Evaluating Branch Alternatives

Pipeline speedup = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}

Assume:
Conditional & Unconditional = 14\%, 65\% change PC

<table>
<thead>
<tr>
<th>Scheduling scheme</th>
<th>Branch penalty</th>
<th>CPI</th>
<th>speedup v. stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline</td>
<td>3</td>
<td>1.42</td>
<td>1.0</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td>1.14</td>
<td>1.26</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>1</td>
<td>1.09</td>
<td>1.29</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.07</td>
<td>1.31</td>
</tr>
</tbody>
</table>
The Memory Abstraction

- Association of \(<\text{name}, \text{value}>\) pairs
  - typically named as byte addresses
  - often values aligned on multiples of size
- Sequence of Reads and Writes
- Write binds a value to an address
- Read of addr returns most recently written value bound to that address

[Diagram showing command (R/W), address (name), data (W), data (R), done]
Relationship of Caches and Pipeline

Memory

I-$

Next PC

4

Address

Next

SEQ

PC

Data

Memory

D-$

MEM/WB

EX/MEM

MEM/MB

Address

Next

Sign

Extend

RD

RD

RD

WB Data

Adder

IF/ID

ALU

Reg File

MUX

Data

Memory

I-$

D-$
Example: Dual-port vs. Single-port

- Machine A: Dual ported memory ("Harvard Architecture")
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

\[
\text{SpeedUp}_A = \frac{\text{Pipeline Depth}}{1 + 0} \times \left(\frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{pipe}}}\right) = \text{Pipeline Depth}
\]

\[
\text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{1 + 0.4 \times 1} \times \left(\frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{unpipe}} / 1.05}\right) = \left(\frac{\text{Pipeline Depth}}{1.4}\right) \times 1.05 = 0.75 \times \text{Pipeline Depth}
\]

\[
\text{SpeedUp}_A / \text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{0.75 \times \text{Pipeline Depth}} = 1.33
\]

- Machine A is 1.33 times faster