**Multi-cycle Datapath**

Single-Cycle implementation has poor performance
- Cycle time longer than necessary for all but slowest instruction

Solution: break the instruction into smaller steps
- Execute each step in one clock cycle
- Cycle time: time it takes to execute the longest step
- Design all the steps to have similar length

Advantages of the multiple cycle processor
- Cycle time is much shorter
- Functional units can be used > once/instruction (less HW)

Disadvantages of the multiple cycle processor
- More timing paths to analyze and tune
- Additional registers to store intermediate data values

---

**A Simple Datapath**

Instruction Fetch (IF) → Decode (RD) → Execute (EX) → Memory (MEM) → Writeback (WB)
**Execution Time**

**Time taken for 1 instruction**
- Add up the execution times of each phase
- Each phase may take different amounts of time
- One instruction executes at a time

**Example**
- Pick some execution times out of the air
  - \( t_{\text{fetch}} = 60\, \text{ns} \), \( t_{\text{decode}} = 30\, \text{ns} \), \( t_{\text{exec}} = 50\, \text{ns} \), \( t_{\text{mem}} = 80\, \text{ns} \), \( t_{\text{wb}} = 20\, \text{ns} \)
- Total execution time per instruction = 240 ns

**Pipelining**

We can execute multiple instructions at the same time!
Each instruction will be in a different phase of execution
Throughput will increase by the number of pipeline stages

Overlap different steps for consecutive instructions
- Steps are called *pipeline stages*
- Need latches after each stage to hold control/data for later stages

A new instruction enters the pipeline at IF on each clock
- Takes 5 clocks to complete execution and leave the pipeline
- Potential throughput of 1 CPI
Pipeline Diagram

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock Cycle =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>IF&lt;sub&gt;I&lt;/sub&gt;</td>
</tr>
<tr>
<td>I+1</td>
<td>IF&lt;sub&gt;I+1&lt;/sub&gt;</td>
</tr>
<tr>
<td>I+2</td>
<td>IF&lt;sub&gt;I+2&lt;/sub&gt;</td>
</tr>
<tr>
<td>I+3</td>
<td>IF&lt;sub&gt;I+3&lt;/sub&gt;</td>
</tr>
<tr>
<td>I+4</td>
<td>IF&lt;sub&gt;I+4&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Like assembly lines in manufacturing

Execution Time

Pipeline stages execute in parallel
- Must wait for slowest one to finish

Pipeline overhead
- Introducing pipelining registers adds latency
- Let’s assume the overhead is 5ns

Our example
- \( t_{\text{fetch}} = 60\text{ns}, \ t_{\text{decode}} = 30\text{ns}, \ t_{\text{exec}} = 50\text{ns}, \ t_{\text{mem}} = 80\text{ns}, \ t_{\text{wb}} = 20\text{ns} \)
- Longest state is 80ns + 5ns = 85ns
- Instruction executes in 5*85 = 425ns
- But, we execute different parts of 5 instructions at same time!

At peak throughput, 1 instruction every 85ns
A Pipelined Datapath

Instruction Fetch (IF) \nDecode (RD) \nExecute (EX) \nMemory (MEM) \nWriteback (WB)

The major hurdle of pipelining
- Situations where next instruction cannot execute
- Reduce the performance of pipelining

Speedup = Pipeline depth/(1 + pipeline stalls/inst)

Want incredibly long pipelines, with no pipeline stalls

Good luck!

Long pipes increase likelihood of hazards
- Let's look at pipeline resources used by instruction class

Pipeline Hazards

CDA 4150 – Pipelining
### Types of Hazards

Three classes of hazards

- **Data hazards**
  - One instruction has a source operand that is the result of a previous instruction in the pipeline (Read-After Write: RAW)
  - There are other types of data hazards (later)

- **Control hazards**
  - The execution of an instruction depends on the resolution of a previous branch instruction in the pipeline
  - Becomes a big problem with deep pipelines

- **Structural hazards**
  - Two or more instructions in the pipeline require the same hardware resource to progress
  - Most common instance is non-pipelined FU (multiplier)
Data Hazards

In MIPS R3000 pipeline, a data dependency occurs when an instruction’s source register is the destination register for either of the 2 prior instructions

- The simplest way to handle this is to stall the dependent instruction at RD until the required register has been written back
- This would cause a 2-clock delay when the instructions are consecutive

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock Cycle =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $r3,$r1,$r2</td>
<td>IF RD EX MEM WB</td>
</tr>
<tr>
<td>sub $r5,$r3,$r4</td>
<td>IF RD RD RD EX MEM WB</td>
</tr>
</tbody>
</table>

Data Hazards

Instruction Fetch (IF) | Decode (RD) | Execute (EX) | Memory (MEM) | Writeback (WB) | add r1,r2,r3 | pc | IS | is | 11 | 4 | 16 | 16 | 32 | =? | alu | smd | DS | Imcl | CDA 4150 – Pipelining
Data Hazards

Instruction Fetch (IF)  
Decode (RD)  
Execute (EX)  
Memory (MEM)  
Writeback (WB)

add r1, r2, r3  
add r4, r1, r5  
add r6, r4, r1  
...

CDA 4150 Pipelining

Data Hazards

Instruction Fetch (IF)  
Decode (RD)  
Execute (EX)  
Memory (MEM)  
Writeback (WB)

add r1, r2, r3  
add r4, r1, r5  
add r6, r4, r1  
...

CDA 4150 Pipelining
**Bypassing**

Performance can be improved by **forwarding (bypassing)** a result from a later stage to an earlier stage.

- The result of an ALU instruction is known at the end of EX.
- The result of a Load instruction is known at the end of MEM.

There is no delay when an ALU instruction executes.

There is 1 clock delay when a Load instruction is directly followed by a dependent instruction.

- The Load instruction is said to have a **latency** of 2 clocks.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock Cycle =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add $t3, $t1, $t2</code></td>
<td>IF RD EX MEMWB</td>
</tr>
<tr>
<td><code>sub $t5, $t3, $t4</code></td>
<td>IF RD EX MEMWB</td>
</tr>
<tr>
<td><code>lw $s1, 0($t3)</code></td>
<td>IF RD EX MEMWB</td>
</tr>
<tr>
<td><code>addi $s2, $s1, 1</code></td>
<td>IF RD RD EX MEMWB</td>
</tr>
</tbody>
</table>

**ALU Bypass**

![ALU Bypass Diagram]

Instruction Fetch (IF) → Decode (RD) → Execute (EX) → Memory (MEM) → Writeback (WB)
When a branch instruction is executed, execution of subsequent instructions depends on whether the branch is taken and the location of the destination.

A simple, but effective approach is to assume the branch is not taken and follow the sequential path.

The branch is resolved at the end of EX:

- If taken, cancel instructions in the sequential path and start fetching from the destination on the next clock
  - this results in a 2-clock delay for taken branches
- If not taken, continue sequentially

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock Cycle =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>I₁</td>
<td>IF</td>
</tr>
<tr>
<td>beq $t0, $t1, L1</td>
<td>IF</td>
</tr>
<tr>
<td>I₃</td>
<td>IF</td>
</tr>
<tr>
<td>L1: I₃</td>
<td>IF</td>
</tr>
<tr>
<td>I₄</td>
<td>IF</td>
</tr>
</tbody>
</table>

CDA 4150 – Pipelining
**ISA Considerations with Pipelining**

Load Delay
- Explicit 1-instruction delay in MIPS ISA
  - If no instruction can be scheduled following the load, nop required
  - MIPS == “Microprocessor without Interlocked Pipeline Stages
  - But other implementations may have different load delays!

Branch Delay
- Explicit 1-instruction delay in MIPS, HP-PA, SPARC
  - For MIPS, if no instruction can be scheduled, NOP required
  - Scheduled instruction must be safe to execute whether or not branch is taken (assembler schedules)
  - For HP-PA/SPARC the instruction following the branch is conditionally executed or squashed

---

**Delayed Branches**

![Diagram of pipeline stages: Instruction Fetch (IF), Decode (RD), Execute (EX), Memory (MEM), Writeback (WB)]

- **PC**: Program Counter
- **IS**: Instruction Store
- **Regs**: Register Files
- **alu**: Arithmetic Logic Unit
- **DS**: Data Store
- **Mdr**: Memory Data Register
- **Ird**: Instruction Register
- **Sbad**: Status and Branch Address Register
- **Cdb**: Control and Delay Register

CDA 4150 – Pipelining
Delayed Branches

Instruction Fetch (IF) → Decode (RD) → Execute (EX) → Memory (MEM) → Writeback (WB)

- pc: Program Counter
- IS: Instruction Set
- regs: Register Set
- alu: Arithmetic Logic Unit
- DS: Data Store
- instr: Instruction
- bgez: Branch on Greater than or Equal to Zero
- offset: Offset Value
- mem: Memory
- wb: Writeback

CDA 450 - Pipelining
**Structural Hazards**

Non-pipelined, multi-cycle functional units
- Integer multiply, divide

Can also have structural hazards on data cache
- Loads access tags/data in MEM
- Stores access tags in MEM, data in WB
- What if a load follows a store?

Structural hazards are detected in decode and stalled there
Only way to remove them is to add functional units
- Or pipeline them
- Or dual port them (caches)

---

**Next Time**

More complicated (deeper) pipelines
Data hazards revisited
Code scheduling for pipelines

What makes pipelining hard
- Interrupts
- Precise exceptions
- Branches and long pipes