INTRODUCTION:
- Pipeline is an implementation technique in which multiple instructions are overlapped in execution
- Today’s processors are fast because of pipelining
- A pipeline is like an assembly line: each step completes one piece of the whole job. Assembly line does not reduce the time it takes to complete an individual job; it increases the amount job being built simultaneously and the rate (eg. workers, working on a car in a factory)
- Pipe stage (pipe segment) small piece of the pipeline instruction
- Therefore, pipelining improves instruction throughput rather than individual instruction execution time. The throughput of the instruction pipeline is determined by how often an instruction exists in pipeline
- The goal of designers is to balance the length of each stage; otherwise there will idle time during a stage. If the stages are perfectly balanced then the time between instructions on the pipelined machine is = Time between instructions (no pipelined) / Number of pipe stages

A RISC processor pipeline operates in much the same way, although the stages in the pipeline are different. While different processors have different numbers of stages, they are basically variations of these five, used in the MIPS R3000 processor:

1. Fetch instructions from memory (IF);
2. Instruction decode and register fetch (ID);
3. Execute the instruction or calculate an address (EX);
4. Access an operand in data memory (MEM);
5. Write the result into a register (WB).

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>RF/ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
</tbody>
</table>

■ Advantages
■ Improves use of functional units even further
■ Dramatically improves performance

■ Disadvantages
■ Increases hardware requirements
■ Introduces several types of “hazard” conditions
Example:

Pipelining Load
- Load instruction takes 5 stages
  - Five independent functional units work on each stage
    - Each functional unit used only once
  - Another load can start as soon as 1st finishes IF stage
  - Each load still takes 5 cycles to complete
  - The *throughput*, however, is much higher

<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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>RF/ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1st lw

<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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>RF/ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2nd lw

<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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>RF/ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3rd lw

Pipelining R-Type
- R-Type instruction takes 4 stages
  - Four independent functional units work on each stage
    - Each functional unit used only once
  - Another can start as soon as 1st finishes IF stage
  - Each still takes 4 cycles to complete
  - Same throughput enhancement as with load

<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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>RF/ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R-type

<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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>RF/ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R-type

<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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>RF/ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R-type
HAZARDS

- There are cases, though, where the next instruction cannot begin executing immediately

- These limits to pipelining are known as *hazards*
  - **Structural Hazard**
    - Resource conflict when the hardware cannot execute two instructions in simultaneous overlapped execution
  - **Data Hazard**
    - Instruction depends on result of instruction still in the pipeline
  - **Control Hazard**
    - Control decision determines execution path, such as when the instruction changes the PC

STRUCTURAL HAZARDS

- Occur when combinations of instructions cannot be accommodated because of resource conflicts

- Often arise when some functional unit is not fully pipelined
- A functional unit can be used once per instruction

- Each functional unit must be used at the same stage for all instructions
  - Load uses Register File’s Write port during its 5th stage
    
    | 1 | 2 | 3 | 4 | 5 |
    |---|---|---|---|---|
    | Load | IF | RF/ID | EX | MEM | WB |
  - R-type uses Register File’s Write port during the 4th stage
    
    | 1 | 2 | 3 |
    |---|---|---|
    | R-type | IF | RF/ID | EX | WB |
Structural Hazard Example

- Consider a load followed immediately by a store
- Processor only has a single write port

Solutions

- Delay instruction until functional unit is ready
  - Hardware inserts a pipeline stall or a bubble that delays execution of all instructions that follow (previous instructions continue)
  - Increases CPI from the ideal value of 1
- Build more sophisticated functional units so that all combinations of instructions can be accommodated
  - Example: Allow two simultaneous writes to the register file

Write Back Stall Solution: Delay R-type register write by one cycle
Data Hazards
- Occur when pipeline changes the order of read/write access to operands so that the order differs from the order seen by sequentially executing instructions
- Caused by several different types of dependencies

Data Hazard Example: Dependencies backwards in time are hazards

Data Dependency Types
- Three classifications of data dependencies for instruction $i$ following instruction $j$
  - Read after Write (RAW)
    - Instr. $j$ tries to read before instr. $i$ tries to write it
  - Write after Write (WAW)
    - Instr. $j$ tries to write an operand before $i$ writes its value
    - Since register writes only occur in WB, the pipeline we have been discussing does not have this type of dependency
  - Write after Read (WAR)
    - Instr. $j$ tries to write a destination before it is read by $i$
    - This also does not occur in this pipeline we have been discussing since all reads happen early in the ID/RF stage and all writes are late in the WB stage
Data Hazard Solution

- **Stalls:** Delay next instruction until ready

```
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11
```

- Or Register file writes on first half and reads on second half
- Or, “Forward” the data to the appropriate unit
Forwarding Limitations

- Forwarding is useful for solving many Read after Write (RAW) hazards
- It does not, however, solve all data hazards
- Solution
  - Stall the pipeline until the values are available to be forwarded

Data Hazard with Forwarding

Solution: Hardware Stall, A *pipeline interlock* checks and stops the instruction issue
**Control Hazard**

- Control hazards are caused by a control dependency such as a branch.
- If a branch is not taken then control simply continues with PC + 4.
- If the branch is taken, then control changes and the PC jumps to a new address.
- Causes a greater performance problem than data hazards.

```
beq r1, r2, L
sub r4, r1, r3
and r6, r2, r7
or r8, r7, r9
L: add r1, r2, r1
```