INTRODUCTION IN PIPELINING

Pipelining

- Pipeline processing is an implementation technique where arithmetic sub-operations or the phases of a computer instruction cycle overlap in execution.
- Pipelining exploits parallelism among instructions by overlapping them - called Instruction Level Parallelism (ILP).
- Pipelining is a technique of decomposing a sequential process into sub-operations, with each sub-process being executed in a special dedicated segment that operates concurrently with all other segments.

Pipelining

- The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.
- The name "pipeline" implies a flow of information analogous to an industrial assembly line.
- It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time.

Introduction to pipelining

- The overlapping of computation in distinct segments is made possible by associating a register with each segment in the pipeline.
- The registers provide isolation between each segment so that each can operate on distinct data simultaneously.
- If there are k stages each of equal delay, then the processing throughput (# of data/instruction processed per second) increases roughly by a factor of k.
- There are two areas of computer design where the pipeline organization is applicable:
  - An arithmetic pipeline divides an arithmetic operation into sub-operations for execution in the pipeline segments.
  - An instruction pipeline operates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle.
Four-segment pipeline

- The simplest way of viewing the pipeline structure is to imagine that each segment consists of an input register followed by a combinational circuit.
- The register holds the data and the combinational circuit performs the sub-operation in the particular segment.
- A clock is applied to all registers after enough time has elapsed to perform all segment activity. In this way the information flows through the pipeline one step at a time.

Example of an arithmetic pipeline

- Suppose that we want to perform the combined multiply and add operations with a stream of numbers:
  \[ A_i \times B_i + C_i \quad \text{for } i = 1, 2, 3, \ldots, 7 \]
- Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit.
- We shall use registers (R1 through R5) that receive new data with every clock pulse.
- The multiplier and adder are combinational circuits.
- The suboperations performed in each segment of the pipeline are as follows:
  - R1 ← Ai, R2 ← Bi: Input Ai and Bi
  - R3 ← R1*R2, R4 ← Ci: Multiply and input Ci
  - R5 ← R3 + R4: Add Ci to product
- The five registers are loaded with new data every clock pulse.
- The first clock pulse transfers A1 and B1 into R1 and R2.

Content of Registers in Pipeline example

<table>
<thead>
<tr>
<th>Clock Pulse Number</th>
<th>Segment 1</th>
<th>Segment 2</th>
<th>Segment 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
</tr>
<tr>
<td>1</td>
<td>A1</td>
<td>B1</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A2</td>
<td>B2</td>
<td>A1*B1</td>
</tr>
<tr>
<td>3</td>
<td>A3</td>
<td>B3</td>
<td>A2*B2</td>
</tr>
<tr>
<td>4</td>
<td>A4</td>
<td>B4</td>
<td>A3*B3</td>
</tr>
<tr>
<td>5</td>
<td>A5</td>
<td>B5</td>
<td>A4*B4</td>
</tr>
<tr>
<td>6</td>
<td>A6</td>
<td>B6</td>
<td>A5*B5</td>
</tr>
<tr>
<td>7</td>
<td>A7</td>
<td>B7</td>
<td>A6*B6</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>A7*B7</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Arithmetic Pipeline

- Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems.
- A pipeline multiplier is essentially an array multiplier with special adders designed to minimize the carry propagation time through the partial products.
- Floating-point operations are easily decomposed into sub-operations, as in the following example of a pipeline unit for floating-point addition and subtraction.

### Pipeline for floating-point addition and subtraction

- The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.
  
  $X = A \times 2^a$
  
  $Y = B \times 2^b$

- $A$ and $B$ are two fractions that represent the mantissas and $a$ and $b$ are the exponents.
- The suboperations that are performed in the four segments of pipelined unit are:
  1. Compare the exponents.
  2. Align the mantissas.
  3. Add or subtract the mantissas.
  4. Normalize the result.

Instruction Pipeline

- Instruction execution involves several operations which are executed successively. This implies a large amount of hardware, but only one part of this hardware works at a given moment.
- Pipelining is an implementation technique whereby multiple instructions are overlapped in execution - different parts of the hardware work for different instructions at the same time.
- The pipeline organization of a CPU is similar to an assembly line: the work to be done in an instruction is broken into smaller steps (pieces), each of which takes a fraction of the time needed to complete the entire instruction.
**Example: Four-Segment Instruction Pipeline**

- **Fetch instruction from memory**
- **Decode instruction and calculate effective address**
- **Branch?**
  - Yes
  - **Fetch operand from memory**
  - No
- **Fetch operand from memory**
- **Execute instruction**
- **Interrupt handling**
- **Interrupt?**
  - Yes
  - Update PC
  - Empty pipe
  - No

**Microprocessors**

**Pipeline**

- The behavior of a pipeline can be illustrated with a space-time diagram. This is a diagram that shows the segment utilization as a function of time.
- The space-time diagram of a four-segment pipeline is represented below. The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number.
- The diagram shows six tasks T1 through T6 executed in four segments.
- No matter how many tasks are in the system, once the pipeline is full, it takes only one clock period to obtain an output.

<table>
<thead>
<tr>
<th>Segment</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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pipeline Speedup**

- Now consider the case where a k-segment pipeline with a clock cycle time $t_p$ is used to execute n tasks.
  - The first task T1 requires a time equal to $k t_p$ to complete its operation since there are k segments in the pipe.
  - The remaining $n - 1$ tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to $(n - 1)t_p$.
  - Therefore, to complete n tasks using a k-segment pipeline requires $k + (n - 1)$ clock cycles.
  - For example, for a four segments and six tasks the time required to complete all the operations is $4 + (6 - 1) = 9$ clock cycles.
  - Next consider a non-pipeline unit that performs the same operation and takes a time equal to $t_n$ to complete each task. The total time required for n tasks is $n t_n$. The speedup of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio:

\[ S = \frac{n \cdot t_n}{(k + n - 1) \cdot t_p} \]

**Pipeline Speedup**

- As the number of tasks increases, $n$ becomes much larger than $k - 1$, and $k + n - 1$ approaches the value of $n$. Under this condition, the speedup becomes:

\[ S = \frac{t_n}{t_p} \]

- If we assume that the time it takes to process a task is the same in the pipeline and nonpipeline circuits, we will have $t_n = k t_p$. Including this assumption, the speedup reduces to

\[ S = \frac{k t_p}{t_p} = k \]
Pipeline Speedup

\[ S = \frac{kt}{t_p} = k \]

- This equation shows that the theoretical maximum speedup that a pipeline can provide is \( k \), where \( k \) is the number of segments in the pipeline.
- However:
  - a greater number of stages increases the overhead in moving information between stages and synchronization between stages.
  - with the number of stages the complexity of the CPU grows.
  - it is difficult to keep a large pipeline at maximum rate because of pipeline hazards.

Instruction Pipelining

- An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations.
- In the most general case, the computer needs to process each instruction with the following sequence of steps:
  - Fetch the instruction from memory.
  - Decode the instruction.
  - Calculate the effective address.
  - Fetch the operands from memory.
  - Execute the instruction.
  - Store the result in the proper place.

Instruction pipelining - Examples

- In general, there are three major difficulties (conflicts or hazards) that cause the instruction pipeline to deviate from its normal operation.
- Pipeline hazards are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycle. The instruction is said to be stalled.
- When an instruction is stalled, all instructions later in the pipeline than the stalled instruction are also stalled.
- Typical conflicts:
  - Resource conflicts (structural hazards) occur when a certain resource (memory, functional unit) is requested by more than one instruction at the same time.
  - Data dependency (data hazards) conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
  - Branch difficulties (control hazards) arise from branch and other instructions that change the value of Program Counter.

- In the following examples we shall use an inversion of the space-time diagram: the horizontal axis displays the time in clock cycles and the vertical axis represents the executed instructions with shifted pipeline contents.
- We assume a 6 stages pipeline:
  - FI = Fetch instruction
  - DI = Decode instruction
  - CO = Calculate operand address
  - FO = Fetch operand
  - EI = Execute instruction
  - WO = Write back operand
Handling Data Hazards

Pipelined computers deal with such conflicts between data dependencies in a variety of ways:

- **Inserting hardware interlocks.** An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict.

- **Operand forwarding (bypassing)** uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments. For ex., instead of transferring an ALU result into a destination register, if it is needed as a source in the next instruction, it passes the result directly into the ALU input, bypassing the register file.

- **Compiler that control reordering of the instructions.** The method is to give the responsibility for solving data conflicts problems to the compiler that translates the high-level programming language into a machine language program. The compiler detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no-operation instructions.

**Branch Hazard (Control Hazard)**

- Hazard for unconditional branch

After the FO stage of the branch instruction the address of the target is known and it can be fetched.

The instruction following the branch is fetched; before the DI is finished it is not known that a branch is executed. Later the fetched instruction is discarded.
Hazard for conditional branch

\[ \text{ADD } R1, R2 \]
\[ \text{BEZ TARGET} \]
\[ R1 \leftarrow R1 + R2 \]
\[ \text{branch if zero} \]

If branch is taken
At this moment, both the condition (set by ADD) and the target address are known.

If branch is not taken
At this moment the condition is known and instr+1 can go on.

Branch Hazard Alternatives

- Stall until branch direction is clear
- Prefetch the target instruction(s) in addition to the instruction(s) following the branch.
- Predict Branch
  - Use of a branch target buffer (BTB) or a loop buffer.
  - Branch prediction. Additional logic to guess the outcome of a conditional branch instruction before it is executed.
- Delayed Branch
  - Define branch to take place AFTER a following instruction branch instruction
    \[
    \begin{align*}
    \text{sequential successor } f & \quad \text{Branch delay of} \\
    \text{sequential successor } n & \quad \text{length } n
    \end{align*}
    \]

References