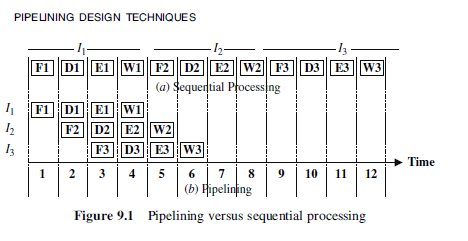
**Pipelining Design Techniques**

There exist two basic techniques to increase the instruction execution rate of a processor. These are to increase the clock rate, thus decreasing the instruction execution time, or alternatively to increase the number of instructions that can be executed simultaneously. Pipelining and instruction-level parallelism are examples of the latter technique.

Pipelining refers to the technique in which a given task is divided into a number of subtasks that need to be performed in sequence. Each subtask is performed by a given functional unit.

shows an illustration of the basic difference between executing four subtasks of a given instruction (in this case fetching F, decoding D, execution E, and writing the results W) using pipelining and sequential processing.



**Performance Measure in Pipelining**

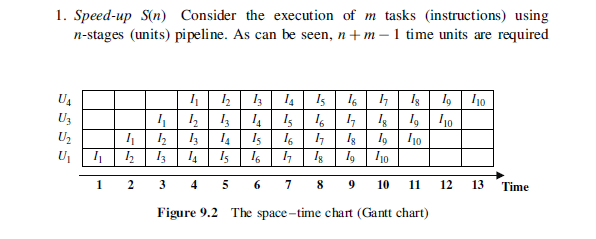
There are 3 basic performance measure for the goodness of pipelining

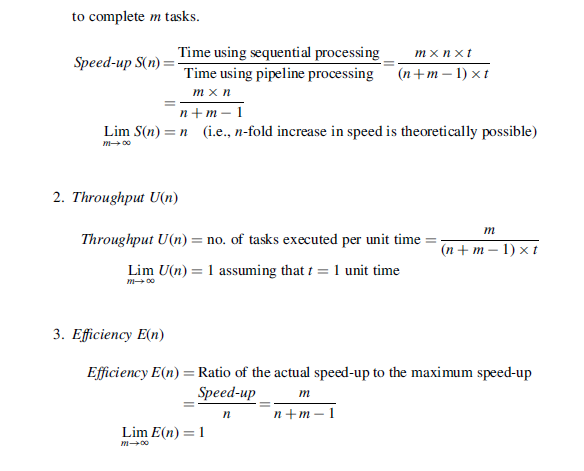
1. Speed Up S(n)
2. Through put U(n)
3. Efficiency E(n)

**Speed Up S(n):** Speed up in pipelining is about increasing the speed of the process, the process should take less time to be completed under pipelining.

* Through put U(n): this is the production rate or the success rate of the pipelining and if it gives same result at a shortest time.
* Efficiency E(n): this talks about saving time, resource and energy and still produce a successful result.

**PIPELINING IN COMPUTER ARCHITECTURE**





* Consider a pipeline having 4 phases with duration 60, 50, 90 and 80 ns. Given latch delay is 10 ns. Calculate-
* Pipeline cycle time
* Non-pipeline execution time
* Speed up ratio
* Pipeline time for 1000 tasks
* Sequential time for 1000 tasks
* Throughpu
* Given-
* Four stage pipeline is used
* Delay of stages = 60, 50, 90 and 80 ns
* Latch delay or delay due to each register = 10 ns

**Part-01: Pipeline Cycle Time-**

* Cycle time
* = Maximum delay due to any stage + Delay due to its register
* = Max { 60, 50, 90, 80 } + 10 ns
* = 90 ns + 10 ns
* = 100 ns

**Non-Pipeline Execution Time-**

Non-pipeline execution time for one instruction

= 60 ns + 50 ns + 90 ns + 80 ns

= 280 ns

**Speed Up Ratio-**

Speed up

= Non-pipeline execution time / Pipeline execution time

= 280 ns / Cycle time

= 280 ns / 100 ns

= 2.8

**Pipeline Time For 1000 Tasks-**

Pipeline time for 1000 tasks

= Time taken for 1st task + Time taken for remaining 999 tasks

= 1 x 4 clock cycles + 999 x 1 clock cycle

= 4 x cycle time + 999 x cycle time

= 4 x 100 ns + 999 x 100 ns

= 400 ns + 99900 ns

= 100300 ns

**Sequential Time For 1000 Tasks-**

Non-pipeline time for 1000 tasks

= 1000 x Time taken for one task

= 1000 x 280 ns

= 280000 ns

**Throughput-**

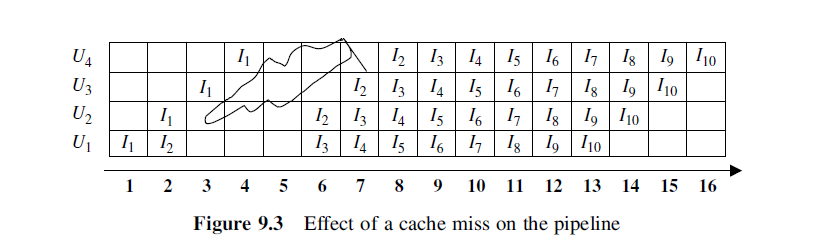
Throughput for pipelined execution

= Number of instructions executed per unit time

= 1000 tasks / 100300 ns

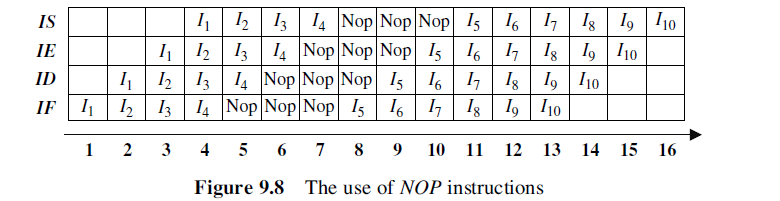
**INSTRUCTION PIPELINE Hazards**

The simple analysis made in above ignores an important aspect that can affect the performance of a pipeline, that is, pipeline stall. A pipeline operation is said to have been stalled if one unit (stage) requires more time to perform its function, thus forcing other stages to become idle.



**Methods Used to Prevent Fetching the Wrong Instruction or Operand**

Use of NOP (No Operation) This method can be used in order to prevent the fetching of the wrong instruction, in case of instruction dependency, or fetching the wrong operand, in case of data dependency. Recall Example 1. In that example, the execution of a sequence of ten instructions I1–I10 on a pipeline consisting of four pipeline stages: IF, ID, IE, and IS were considered. In order to show the execution of these instructions in the pipeline, we have assumed that when the branch instruction is fetched, the pipeline stalls until the result of executing the branch instruction is stored. This assumption was needed in order to prevent fetching the wrong instruction after fetching the branch instruction. In real-life situations, a mechanism is needed to guarantee fetching the appropriate instruction at the appropriate time. Insertion of “NOP” instructions will help carrying out this task. A “NOP” is an instruction that has no effect on the status of the processor.



**Unconditional Branch Instructions**

In order to be able to reduce the pipeline stall due to unconditional branches, it is necessary to identify the unconditional branches as early as possible and before fetching the wrong instruction. It may also be possible to reduce the stall by reordering the instruction sequence. These methods are explained below.

**REORDERING OF INSTRUCTIONS**

In this case, the sequence of instructions are reordered such that correct instructions are brought to the pipeline while guaranteeing the correctness of the final results produced by the reordered set of instructions. Consider,

* For example, the execution of the following group of instructions I1, I2, I3, I4, I5., ., Ij, Ijþ1, ., . on a pipeline consisting of three pipeline stages:

**REORDERING OF INSTRUCTIONS**

