A Lecture on Pipeline

By

Prof. Dr. Atal Chaudhury

Department Of Computer Science & Engineering

Jadavpur University

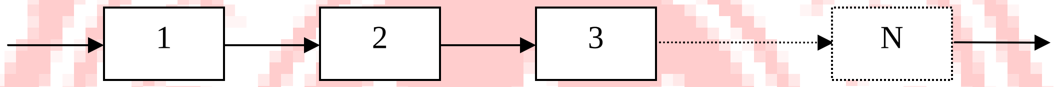
<u>PIPELINING</u>

The basic idea behind pipeline design is quite natural; it is not specific to computer technology.

In fact the name pipeline stems from the analogy with petroleum pipelines in which a sequence of hydrocarbon products is pumped through a pipeline. The last product might well be entering the pipeline before the first product has been removed from the terminus.

The key contribution of the pipelining is that it provides a way to start a new task before an old one has been completed. Pipeline processing has led to the tremendous improvement of system throughput in the modern digital computers.

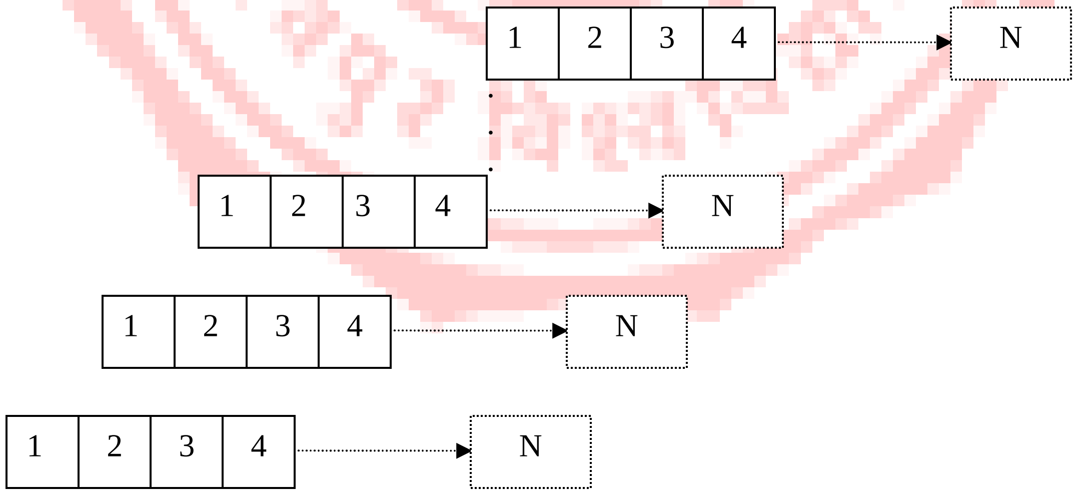
Consider the figure below, which shows a sequential process being done step by step over a period of time.



An N-step sequential process

Total time required to complete this process is N units, assuming that each step takes one unit time. (In the figure above each box denotes the execution of a single step, and the label in the box denotes the number of the step being executed.)

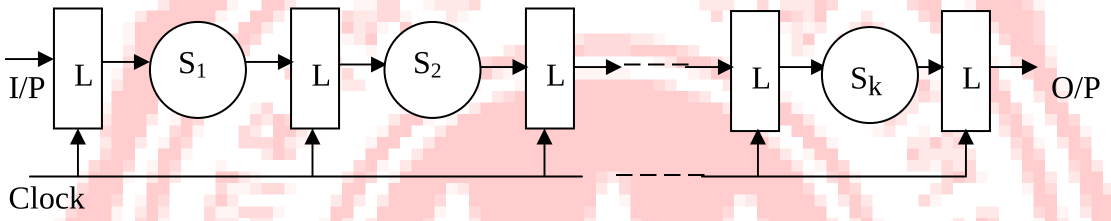
To perform the same process using pipeline technique assume we have N independent stations to perform the sequence of N steps of the process.



Space-time diagram for Pipelined execution of an N-step sequence

Here we have a continuous stream of jobs going through the N-sequential steps of the process. Each horizontal row of boxes represents the time history of one job and each vertical column of boxes represents the activity at a specific time. (Here each job will take individually N unit of time but to be noted that after the first job completed after N units of time we will get one job completed after each unit of time.)

A basic linear-pipeline processor is depicted below. The pipeline consists of a cascade of processing stages. The stages are pure combinational circuits performing arithmetic or logical operations over the data stream flowing through the pipe. The stages are separated by high-speed interface latches. The latches are high speed registers for holding the intermediate results between the stages. Information flows between adjacent stages are under the control of a common clock applied to all the latches simultaneously.



L → Latch; S_i → Processing Stages

<u>Clock Period: -</u>

The logic circuitry in each stage S_i has a time delay denoted by τ; and τ_l be the time delay of each interface latch.

Then clock period of a linear pipeline = $\tau = \max \{ \tau_i \}_1^k + \tau_l = \tau_{max} + \tau_l$ and frequency of pipeline processor = $f = 1/ \tau$.

<u>Speed up: -</u>

Ideally a linear pipeline with k stages can process n tasks in $T_k = k + (n-1)$ clock periods, where k cycles are used to fill up the pipeline or to complete execution of the first task and (n-1) cycles are needed to complete the remaining (n-1) tasks.

The same number of tasks can be executed in a non-pipeline processor with an equivalent function in $T_l = n.k$ clock periods.

We define the speed up of a k-stage linear-pipeline processor over a non-pipeline processor as $S_k = T_l / T_k = n.k / k + (n-1)$.

As the number of tasks increases n becomes much larger than k-1 that is as n>>k so $k+(n-1) \rightarrow n$, $S_k \rightarrow k$ that is the maximum speed up that a linear pipeline can provide is k, where k is the number of stages in the pipe.

(The maximum speed is never fully achievable because of data dependencies between instructions, interrupts, program branching and other factors. Many pipeline cycles may be wasted on a waiting state caused by out-of-sequence instruction execution.)

To understand the operational principles of pipeline computation, let us illustrate the design of a pipeline floating-point adder. The pipeline is linearly constructed with four functional stages and the inputs to this pipeline are two normalized floating-point numbers.

$A = a \times 2^p$; $B = b \times 2^q$; where a, b are two fractions and p and q are their exponents respectively. (For simplicity base 2 is assumed).
We should compute the sum: -
$C = A+B = C \times 2^r = d \times 2^s$ where $r = \max(p, q)$ and $0.5 <= d < 1$.

<u>Operations performed in the four stages are as follows</u>: -

1. Compare the two exponents p and q to reveal the larger exponent $r = \max(p, q)$ and to determine their difference $t = | p-q |$. Shift right the function associated with the smaller exponent by t bits to equalize the two exponents before fraction addition.
2. Add the pre-shifted fraction with the other fraction to produce the intermediate sum fraction C, where $0 <= C <1$.
3. Count the number of leading zeros, say u, in fraction C and shift left C by u bits to produce the normalized fraction sum $d = c \times 2^{u}$ with leading bit 1.
4. Update the large exponent s by subtracting $S = r - u$ to produce the output exponent.

After defining clock period and speed up we need to introduce two related measures of the performance of a linear pipeline processor. The product (area) of a time interval and a storage space in the space time diagram is called a time-space span. A given time-space span can be in either a busy state or an idle state, but not both. We use this concept to measure the performance of a pipeline.

Efficiency: - The efficiency of a linear pipeline is measured by the percentage of busy time-space spans over the total time-space span (which equals the sum of all busy and idle time-space spans).
Let n = no. of tasks; k = no. of stages; τ = clock period.
Pipeline efficiency is defined by: - $\eta = n. k. \tau / k. [k \tau + (n - 1) \tau] = n / k + (n-1)$.
Note that $\eta \rightarrow 1$ as $n \rightarrow \alpha$; this implies larger the number of tasks flowing through the pipeline, better is the efficiency.

Throughput: - Number of tasks completed per unit time period is called the throughput.
Time required to complete n tasks = $k. \tau + (n - 1) \tau$.
So throughput $w = n / k. \tau + (n - 1) \tau = \eta / \tau$.
Note that when $\eta \rightarrow 1$, $w \rightarrow 1/ \tau = f$; that is maximum throughput of a linear pipeline is equal to its frequency, which corresponds to one task per clock.

Classification of Pipeline Processor: -

1. Arithmetic Pipeline: - The arithmetic logic unit of a computer can be segmentized for pipeline operations (The floating point adder shown before).
   Examples: -- 4-stage pipe used in Star-100.
   8-stage pipe used in TI-ASC.
   14-stage pipe used in Cray-1.
   26-stage pipe used in Cyber-205.

2. Instruction Pipeline: - The execution of a stream of instructions can be pipelined by overlapping the execution of the current instruction with the fetch, decode, and operand fetch of subsequent instructions (also known as instruction look-ahead). Almost all high-performance computers are now equipped with instruction-execution pipelines.

3. Processor Pipeline: - Same data stream passes through a cascade of processors, each of which processes a specific task. The data stream passes the first processor with results stored in a memory block accessible to second processor and then the refined result passes to the third one and so on. The pipelining of multiple processors is not yet well acceptable as a common practice.

According to pipeline configurations and control strategies, Ramamoorthy and Live proposed three pipeline classification schemes: -

1. Uni-function vs. Multi-function Pipelines: - A pipeline unit with a fixed and dedicated function (e.g. floating point adder) is called uni-functional. Cray -1 has 12 uni-functional pipeline units. A multi-function pipe may perform different functions, either at different times or at the same time, by interconnecting different subsets of stages in the pipeline. TI – ASC has four multi-function pipeline processors.

2. Static vs. Dynamic Pipelines: - A static pipeline may assume only one functional configuration at a time. Static pipelines can be either uni-functional or multi-functional. The function performed by a static pipeline should not change frequently; otherwise its performance may be very low.

   A dynamic pipeline processor permits several functional configurations to exist simultaneously. (In this sense dynamic pipeline must be multi-functional and uni-functional pipe must be static). The dynamic configuration needs much more elaborate control and sequencing mechanisms than these for static pipelines.

3. Scalar vs. Vector Pipelines: - A scalar pipeline processes a sequence of scalar operands under the control of a DO loop. Instructions in a small DO loop are often pre-fetched into the instruction buffer. The required scalar operands for repeated scalar instructions are moved into a data cache in order to continuously supply the pipeline with operands.

   Vector pipelines are specially designed to handle vector instructions over vector operands. The handling of vector operands in vector pipelines is under firmware and hardware controls rather than under software control as in scalar pipelines.

4. Arithmetic & Instruction Pipelining: - Pipeline processing can occur not only in the data stream (called arithmetic pipelining, e.g. floating point addition) but in the instruction stream as well. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. One possible case that may degrade the performance of the system is when a branch instruction is executed. In this case all the instructions that were fetched have to be flashed out, and a new set of instructions are fetched.

Typical instruction-execution sequence is: -

1. Instruction fetch: - obtain a copy of the instruction from the memory.
2. Instruction decode: - examine the instruction and prepare to initialize the control signals required to execute the instruction in subsequent steps.
3. Address generation: - compute the effective address of the operands by performing indirection or indexing as specified by the instruction.
4. Operand fetch: - for READ operations, obtain the operand from central memory.
5. Instruction execution: - execute the instruction in the processor.
6. Operand store: - for WRITE operations, return the resulting data to central memory.
7. Update program counter: - generate the address of the next instruction.

Some inherent difficulties prevent the instruction pipeline from executing if at its maximum rate.

- ➢ Different segments may take different times to operate on the incoming information.
- ➢ Some segments are skipped for some operations. (For example immediate addressing or register addressing does not need effective address calculations).
- ➢ Two or more segments might require the access of memory at the same time. (For example instruction fetch and operand fetch may require memory access simultaneously. Normally operand fetch gets the priority).

The design of the pipeline thus would be most efficient, if the segments are divided, such that each segment takes almost equal time to complete the task given to it.

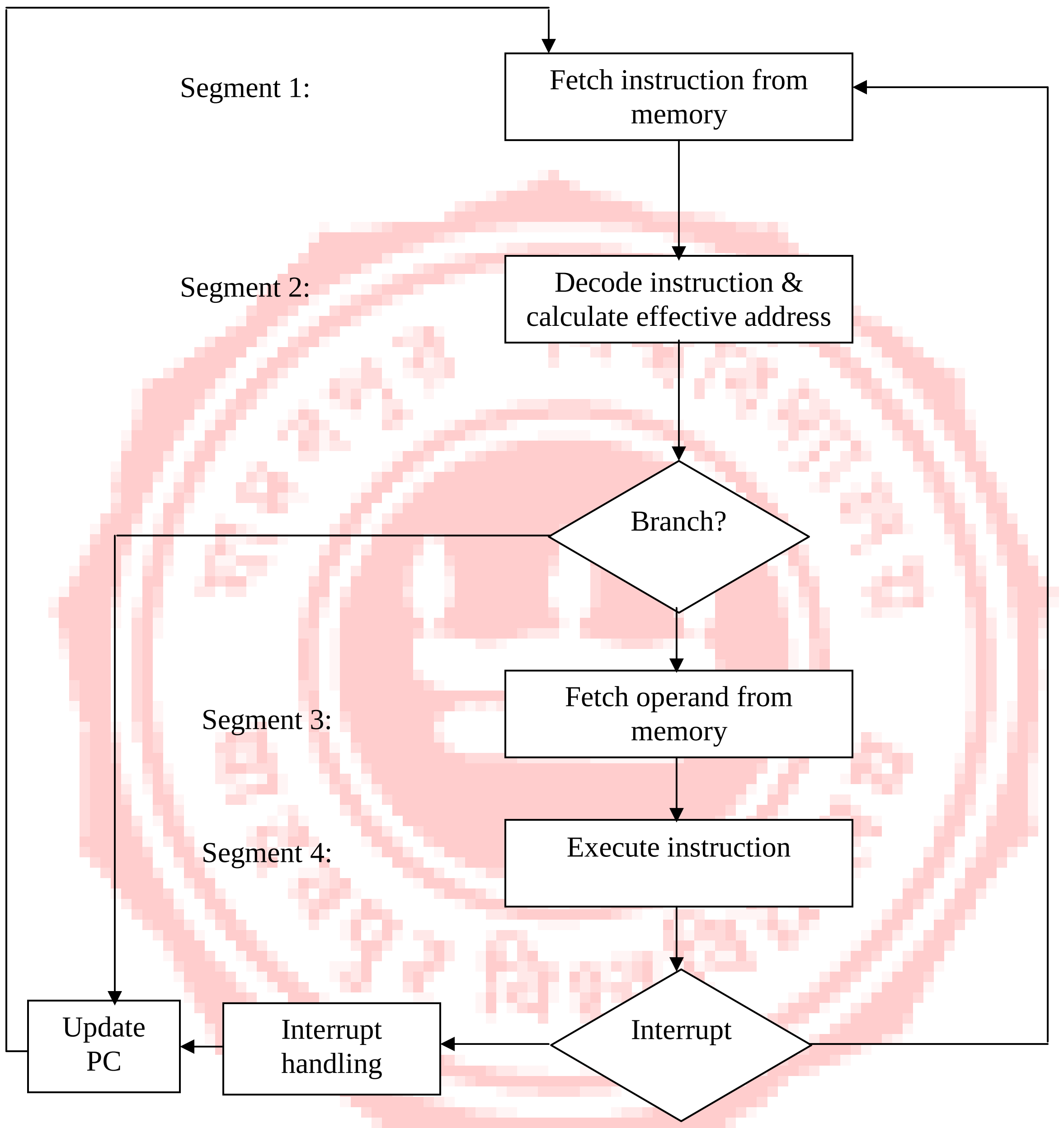
Assuming that the decoding of the instruction can be combined with the calculation of the effective address into one segment.

Assuming further that most of the instructions places the result into a processor register so that the instruction execution and storing of the result can be combined into one segment. This reduces the instruction pipeline into four segments.

Thus up-to four instructions can be processed in pipeline.

1. FI: - instruction fetch segment.
2. DA: - decoder and the effective address calculation segment.
3. FO: - operand fetch segment.
4. EX: - execute segment.

Four-Segment Instruction Pipelining: -


Segment 1:
Fetch instruction from memory
Segment 2:
Decode instruction & calculate effective address
Branch?
Segment 3:
Fetch operand from memory
Segment 4:
Execute instruction
Interrupt
Interrupt handling
Update PC

It is assumed that the processor has the separate instruction and the data memories so that FI and FO segment can proceed together. In other case i.e. in Von-Neumann architecture operand fetch is given higher precedence over instruction fetch.

Pipeline Hazards: -

The pipeline processor discussed above has four stages, which means that the expected rate of instruction processing is four times that of sequential operation. However this speed up would be achieved only if pipelined operation could be sustained without interruption through program execution. Unfortunately this is not the case.

Let us consider an instruction pipelining having four stages as Fetch, Decode, Execute and Write. For variety of reasons, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted. Stage E (execute) is responsible for arithmetic and logical operations and one clock cycle is assigned for the task. Although this may be sufficient for most operations but operation like divide may require more time to complete.

Clock	1	2	3	4	5	6	7	8	9	10
Instruction										
I_1	F_1	D_1	E_1	W_1						
I_2		F_2	D_2	$\longleftarrow$	E_2	$\longrightarrow$	W_2			
I_3			F_3	D_3	X	X	E_3	W_3		
I_4				F_4	X	X	D_4	E_4	W_4	
I_5					X	X	F_5	D_5	E_5	W_5

In the previous example the E stage of instruction I_2 requires 3 cycles to complete instead of one, from cycle 4 through cycle 6, thus in cycle 5 and 6. W-stage has nothing to do as no data to work with. Meanwhile the information in buffer prior to E-stage remain blocked till E-stage has completed its operation, this means that stage D and in turn stage F are blocked from accepting new instructions. Thus steps E_3, D_4 & F_5 all are postponed.

Pipelined operation is said to have been stalled for two clock cycles and normal operation resumes in cycle 7. Any condition that causes the pipeline to stall is called a <u>HAZARD</u>. The example we have just treated is known as data hazard.

<u>Data Hazard</u> is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline, as a result some operation has to be delayed and the pipeline stalls.

The pipeline may also be stalled because of delay in the availability of an instruction. This may be result of a miss in cache, requiring the instruction to be fetched from main memory, branch instruction e.t.c. Such hazards are often called <u>Instruction Hazard</u> or <u>Control Hazard</u>.

Clock	1	2	3	4	5	6	7	8	9 $\longrightarrow$
Instruction									
I_1	F_1	D_1	E_1	W_1					
I_2		$\longleftarrow$	F_2	$\longrightarrow$		D_2	E_2	W_2	
I_3						F_3	D_3	E_3	W_3

<u>Alternatively</u>: -

Clock	1	2	3	4	5	6	7	8	9
Stage									
F	F_1	F_2	F_2	F_2	F_2	F_3			
D		D_1	idle	idle	idle	D_2	D_3		
E			E_1	idle	idle	idle	E_2	E_3	
W				W_1	idle	idle	idle	W_2	W_3

The effect of cache miss on pipelined operation is illustrated above. Instruction I_1 is fetched from cache in cycle 1 and its execution proceeds normally. However the fetch operation for instruction I_2 which started in cycle 2 results in a cache miss. The fetch unit remains suspended till I_2 arrives. We assumed I_2 is received and loaded at the end of cycle 5 and the pipeline resumes its normal operation from that point.

The idle cycles are sometimes called <u>stalls</u> or <u>bubbles</u>.

The third type of hazard that may be encountered in pipelined operation is known as <u>Structural Hazard</u>. This is the situation when two instructions require the use of a given hardware resource at the same time. The most common case arises with simultaneous access to memory. One instruction may need to access memory as part of execute or write while another instruction is being fetched. (Many processor use separate instruction cache and data cache to avoid this hazard).

It is important to understand that pipeline does not result in individual instructions being executed faster; rather it is the throughput that increases.

An important goal in designing processors is to identify all hazards that may cause pipeline to stall and to find ways to minimize their impact.

Pipeline Hazards and their Solutions: -

In general there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

1. Data dependency conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available and is known as Data Hazard.
2. Branch difficulties arise from branch and other instructions that change the value of PC or cache miss and are known as Instruction Hazard.
3. Resource conflicts caused by access to memory by two segments at the same time and is known as Structural Hazard. (Most of these conflicts can be resolved by using separate instruction cache and data cache).

Solutions towards Data Hazards: -

1. Hardware Interlocks: - An interlock is a circuit that detects instructions whose source operands are destinations of instruction 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. This approach maintains the program sequence by using hardware to insert required delays.

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

3. Delayed Load: - In some computers the responsibility of solving data conflicts is given to the compiler that translates the high-level programming language into a machine language program. In such cases compilers detect the data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no-operation instructions.

Solutions towards Instruction Hazards: -

1. Pre-fetch target instruction: - One way of handling a conditional branch is to pre-fetch the target instruction in addition to the instruction following the branch, both are saved until the branch is executed. If the branch condition is successful, the pipeline continues from the branch target instruction. An extension of this procedure is to continue fetching instructions from both places until the branch decision is made.

2. Branch target buffer: - Branch target buffer (BTB) is an associative memory included in the fetch segment of the pipeline. Each entry in the BTB consists of the address of a previously executed branch instruction and the target instruction for that branch. It also stores nest few instructions starting from the branch target instruction. When the pipeline decodes a branch instruction, it searches the BTB memory for the address of the instruction. If it is in BTB, the instruction is available directly and pre-fetch continues from the new path. If the instruction is not in the BTB the pipeline shifts to a new instruction stream and stores the target instruction in the BTB as a new entry. The advantage of this scheme is that branch instructions that have occurred previously are readily available in the pipeline without interruption.

3. Loop Buffer: - This is a small very high speed register file maintained by the instruction fetch segment of the pipeline. When a program loop is detected in the program, it is stored in the loop buffer in it's entirely, including all branches. The program loop can be executed directly from the high speed registers without having access to main memory.

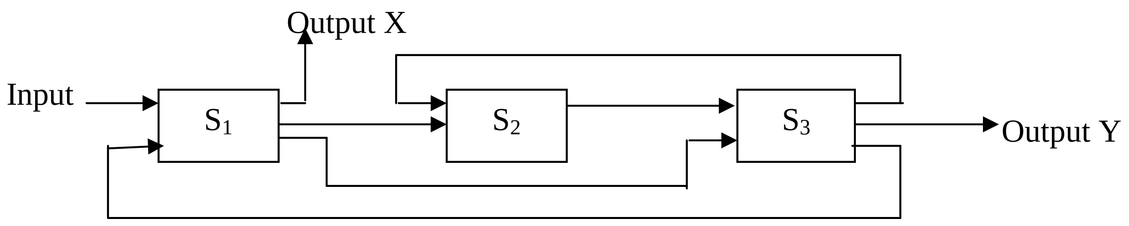
4. Branch Prediction: - A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed. The pipeline then begins pre-fetching the instruction stream from the predicted path. A correct prediction eliminates the wasted time caused by branch penalties.

5. Delayed Branch: - A technique employed in most RISC processors. Here the compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without interruptions. An example of delayed branch is the insertion of a no-operation instruction after a branch instruction. This causes the computer to fetch the target instruction during the execution of no-operation instructions.

Dynamic/Nonlinear Pipeline: -

A dynamic pipeline can be reconfigured to perform variable functions at different times. (The traditional linear pipelines are static pipelines because they are used to perform fixed functions).

A dynamic pipeline allows feed-forward and feedback connections in addition to the streamline connections. For this reason, some authors call such a structure a nonlinear pipeline.



Feed-forward and feedback connections make the scheduling of successive events into the pipeline a nontrivial task. The output of the pipeline is not necessarily from the last stage. In fact, following different dataflow patterns, one can use the same pipeline to evaluate different functions.

Reservation Table: -

The utilization pattern of successive stages in a synchronous pipeline is specified by a reservation table. This table is essentially a space-time diagram depicting the precedence relationship in using the pipeline stages. For linear pipeline the utilization follows diagonal streamline pattern.

Stages	1	2	3	4 → Clock Cycles
S_1	X			
S_2		X		
S_3			X	
S_4				X

A four stage linear pipeline

Reservation table for non-linear pipeline is not trivial. Given a pipeline configuration, multiple reservation tables can be generated for the evaluation of different functions.

Two reservation tables are given below corresponding to two functions X and Y respectively.

Reservation table for function X: -

	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X				
S_3			X		X		X	

Reservation table for function Y: -

	1	2	3	4	5	6
S_1	Y				Y	
S_2			Y			
S_3		Y		Y		Y

Each reservation table displays the time-space flow of data through the pipeline for one function evaluation. A number of pipeline configurations may be represented by the same reservation table. There is a many-to-many mapping between various pipeline configurations and different reservation tables.

The number of columns in a reservation table is called the evaluation *time* of a given function. (For example, the function X requires 8 clock cycles to evaluate, and function Y requires 6 clock cycles).

A pipeline <u>initiation table</u> corresponds to each function evaluation. All initiations to a static pipeline use the same reservation table. On the other hand, a dynamic pipeline may allow different initiations to follow a mix of reservation tables.

The checkmarks in each row of the reservation table correspond to the time instants (cycles) that a particular stage will be used. Multiple checkmarks in a row simply imply usage of the same stage. Contiguous checkmarks in a row simply imply the extended usage of a stage over more than one cycle. Multiple checkmarks in a column mean that multiple stages are used in parallel during a particular clock cycle.

<u>Latency Analysis</u>: -

The number of time units (clock cycles) between two initiations of a pipeline is *the latency* between them. (A latency of *k* means that two initiations are separated by *k* clock cycles).
Any attempt by two or more initiations to use the same pipeline stage at the same time will cause a collision i.e. resource conflict and thus collision should be avoided. Latencies that cause collisions are called <u>forbidden latencies</u>. (In evaluating X, latencies 2 and 5 are forbidden).

	1	2	3	4	5	6	7	8	9	10	11	
S_1	**X_1**		**X_2**		**X_3**	**X_1**	**X_4**	**X_1 X_2**	**X_5**	**X_2 X_3**		
S_2		**X_1**		**X_1 X_2**		**X_2 X_3**		**X_3 X_4**		**X_4**		....
S_3			**X_1**		**X_1 X_2**		**X_1 X_2 X_3**		**X_2 X_3 X_4**			

Collision with scheduling latency 2
(Similarly there will be collision with latency 5 as well)

To detect a forbidden latency, one needs simply to check the distance between any two checkmarks in the same row of the reservation table. Thus 2, 4, 5 and 7 are forbidden latencies for X. 2 and 4 are forbidden latencies for Y.

A <u>latency cycle</u> is a latency sequence which repeats the same subsequence (cycle) indefinitely without collision. (For X the latency cycle 1, 8 can go-on indefinitely without any collision).

A <u>constant cycle</u> is a latency cycle which contains only one latency value. (For k the constant cycle could be 3 or 6).

The average latency of a latency cycle is obtained by dividing the sum of all latencies along the cycle. (Thus latency cycle 1, 8 has an average latency of (1+8)/2 = 4.5). The average latency of a constant cycle is simply the latency itself.

Collision-Free Scheduling: -

Objective is to obtain the shortest average latency between initiations without causing collisions.

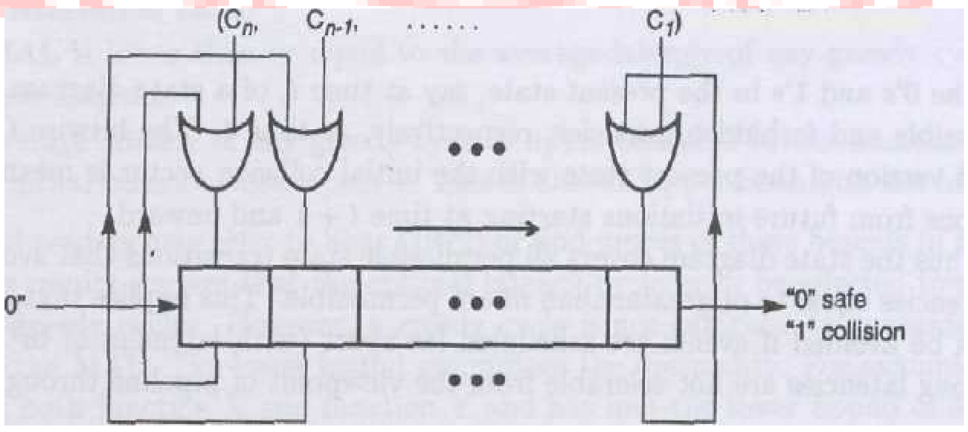
Collision Vector: -

By examining the reservation table, one can distinguish the set of permissible latencies from the set of forbidden latencies. For a reservation table with n columns, the maximum forbidden latency $m \le$ **n-** 1. The permissible latency p should be as small as possible and the choice is made in the range $1 \le p \le m - 1$ (as latency m is always forbidden).

The combined set of permissible and forbidden latencies can be easily displayed by a collision vector, which is an m-bit binary vector $C = (C_m C_{m-1} \ldots C_2 C_1)$ where $C_i = 1$ for forbidden latency and $C_i = 0$ for permissible latency. (Collision vector for X is $C_x = 1011010$ and $C_y = 1010$).

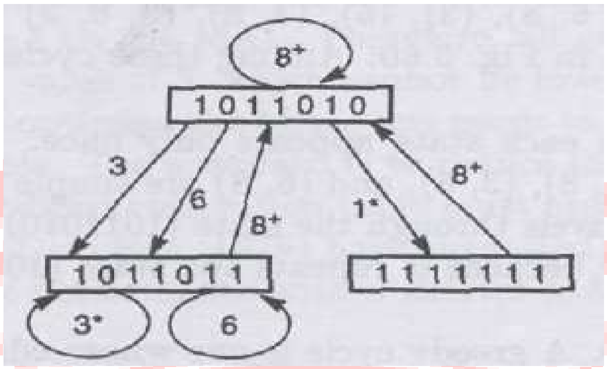
State Diagrams: -

From collision vector, state diagram is drawn specifying the permissible state transitions among successive initiations corresponding to permissible latencies. (The collision vectors Cx or C_y above are the initial collision vectors). Let p be a permissible latency within the range $1 \le p \le m - 1$.
The next state of the pipeline at time t + p is obtained using a m-bit right shift register as follows.

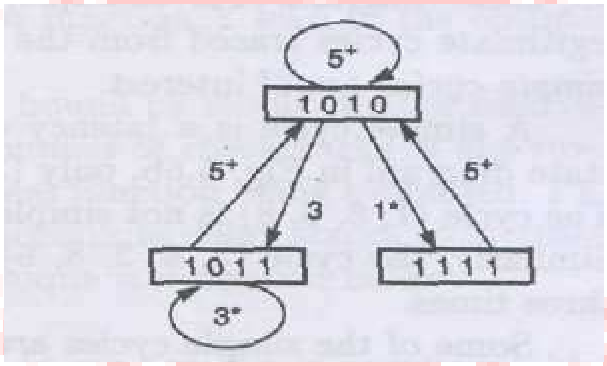


$(C_n\ C_{n-1} \ldots\ldots\ C_1) \rightarrow$ Initial collision vector.

The initial collision vector C is initially loaded into the register. The register is then shifted to the right. Each 1-bit shift corresponds to an increase in the latency by 1. When a 0 bit emerges from the right end after p shifts, it means p is a permissible latency. Logical 0 enters from the left end of the shift register. The next state after p shifts is thus obtained by bitwise-ORing the initial collision vector with the shifted register contents.



State diagram for X



State diagram for Y

For function X, from the initial state (1011010), only three outgoing transitions are possible, corresponding to the three permissible latencies 6, 3, and 1 in the initial collision vector. (Again from state (1011011), one reaches the same state after either three shifts or six shifts).

When the number of shifts is m+1 or greater, all transitions are redirected back to the initial state. (For example, after eight or more (denoted as 8^{+}) shifts, the next state must be the initial state, regardless of which state the transition starts from). Once the initial collision vector is determined, the corresponding state diagram is uniquely determined.

The O's and l's in the present state, say at time *t*, of a state diagram indicate the permissible and forbidden latencies, respectively, at time *t*. The bitwise ORing of the shifted version of the present state with the initial collision vector is meant to prevent collisions from future initiations starting at time *t* + 1 and onward. Thus the state diagram covers all permissible state transitions that avoid collisions.

All latencies equal to or greater than m are permissible. This implies that collisions can always be avoided if events are scheduled far apart (with latencies of m^+). However, such long latencies are not tolerable from the viewpoint of pipeline throughput.

Greedy Cycles: -

There are infinitely many latency cycles one can trace from the state diagram. (For example, (1, 8), (1, 8, 6, 8), (3), (6), (3, 8), (3, 6, 3) ..., are legitimate latency cycles). Among these cycles, only simple cycles are of interest. A simple cycle is a latency cycle in which each state appears only once.

((For X (3), (6), (8), (1, 8), (3, 8), and (6, 8) are simple cycles. The cycle (1, 8, 6, 8) is not simple because it travels through the state (1011010) twice. Similarly, the cycle (3, 6, 3, 8, 6) is not simple because it repeats the state (1011011) three times).

Some of the simple cycles are greedy cycles. A greedy cycle is one whose edges are all made with minimum latencies from their respective starting states. (For X (1, 8) and (3) are greedy cycles and for Y (1, 5) and (3) are greedy cycles.

Greedy cycles must first be simple and their average latencies must be lower than these of other simple cycles. (The minimum-latency edges in the state diagrams are marked with asterisks).

At least one of the greedy cycles will lead to the MAL. The collision-free scheduling of pipeline events is thus reduced to finding greedy cycles from the set of simple cycles. The greedy cycle yielding the MAL is the final choice.

For X MAL (minimum average latency) is 3 from greedy cycle (3) and for Y MAL is also 3 from both greedy cycles (1, 5) and (3).

Pipeline Schedule Optimization: -

The idea is to insert non-compute delay stages into the original pipeline. This will modify the reservation table, resulting in a new collision vector and an improved state diagram. The purpose is to yield an optimal latency cycle, which is absolutely the shortest.

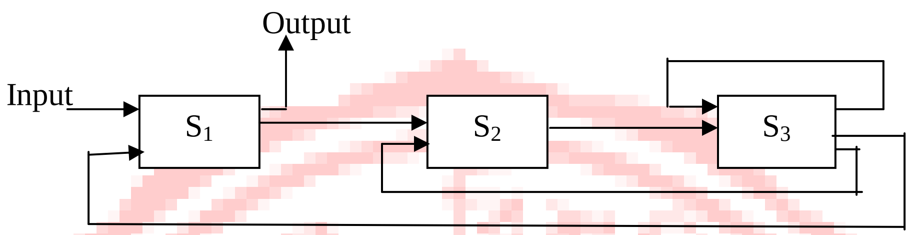
Bounds on the MAL: -

(1) The MAL is lower-bounded by the maximum number of checkmarks in any row of the reservation table.

(2) The MAL is lower than or equal to the average latency of any greedy cycle in the state diagram.

(3) The average latency of any greedy cycle is upper-bounded by the number of l's in the initial collision vector plus 1. This is also an upper bound on the MAL.

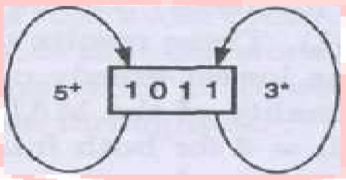
Let us consider the 3-stage pipeline along with the reservation table below: -



3-Stage Pipeline

	1	2	3	4	5
S_1	X				X
S_2		X		X	
S_3			X	X	

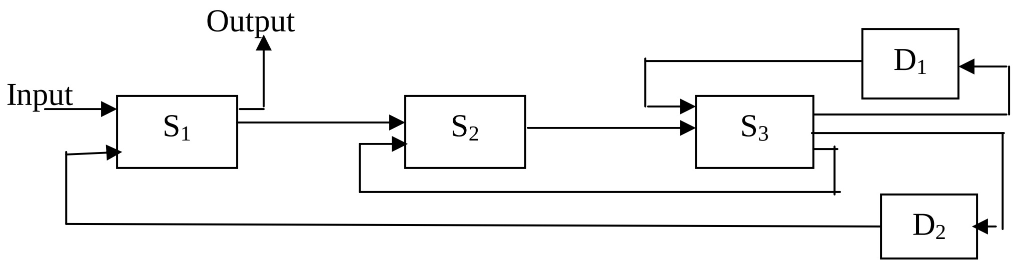
Reservation Table

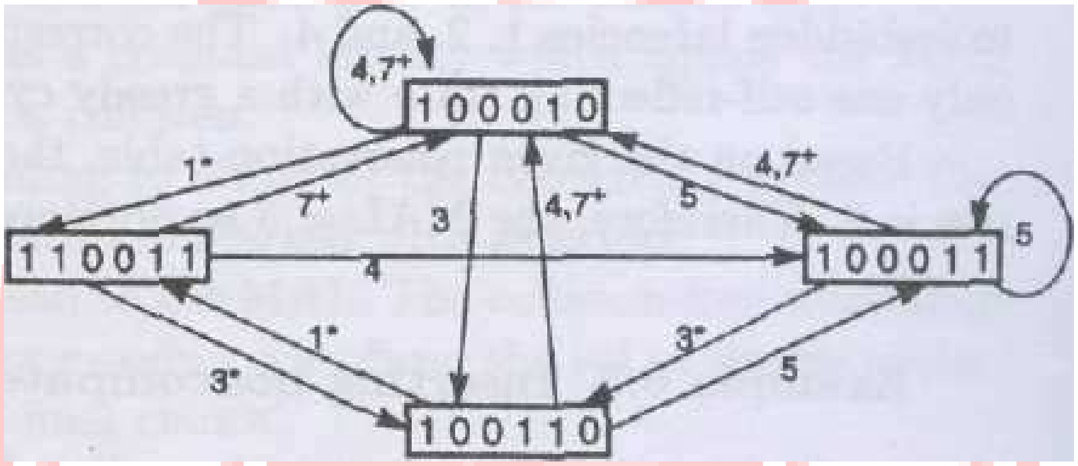


State Diagram

The corresponding state diagram contains only one greedy cycle of latency 3 equal to the MAL. But based on the given reservation table, the maximum number of checkmarks in any row is 2. So, the MAL = 3 so obtained is not optimal.

Now let us insert a non-compute stage D_1 after stage S_3 will delay both X_1 and X_2 operations one cycle beyond time 4. Yet another non-compute stage D_2 after the second usage of S_1 will delay the operation X_2 by another cycle. Resulting new pipeline configuration and reservation tables are as follows.





	1	2	3	4	5	6	7	8
S_1	X				$\rightarrow$	$\rightarrow$	X_2	
S_2		X		X				
S_3			X	$\rightarrow$	X_1			
S_4				D_1				
S_5						D_2		

In total, the operation X_1 has been delayed one cycle from time 4 to time 5 and the operation X_2 has been delayed two cycles from time 5 to time 7. All remaining operations are unchanged. The new table leads to a new collision vector (100010) and a modified state diagram.

This diagram displays a greedy cycle (1,3), resulting in a reduced MAL = (l + 3)/2 = 2.