4th Semester CE/IT/CSE

CE0404 - Computer Organization & Architecture

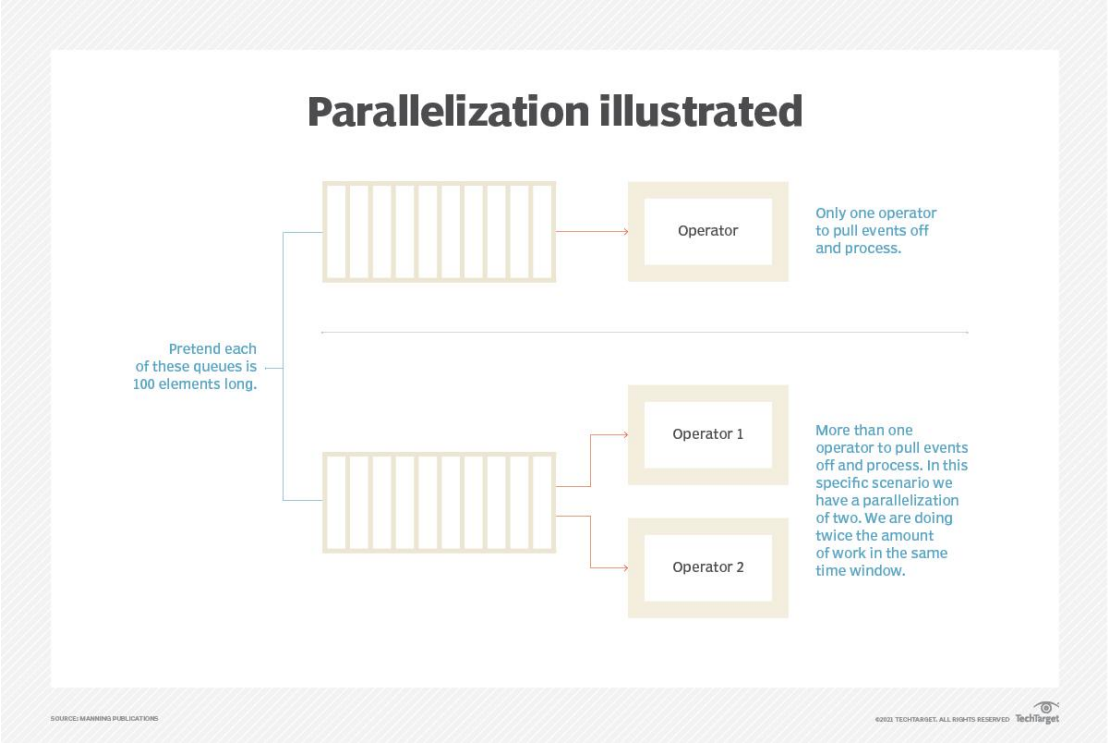
Unit-3 Pipeline

1. Pipeline

Pipelining is the process of storing and prioritizing computer instructions that the processor executes. The pipeline is a "logical pipeline" that lets the processor perform an instruction in multiple steps. The processing happens in a continuous, orderly, somewhat overlapped manner.

In computing, pipelining is also known as pipeline processing. It is sometimes compared to a manufacturing assembly line in which different parts of a product are assembled simultaneously, even though some parts may have to be assembled before others. Even if there is some sequential dependency, many operations can proceed concurrently, which facilitates overall time savings.

Pipelining creates and organizes a pipeline of instructions the processor can execute in parallel.



The pipeline is divided into logical stages connected to each other to form a pipelike structure. Instructions enter from one end and exit from the other. Pipelining is an ongoing, continuous process in which new instructions, or tasks, are added to the pipeline and completed tasks are removed at a specified time after processing

completes. The processor executes all the tasks in the pipeline in parallel, giving them the appropriate time based on their complexity and priority. Any tasks or instructions that require processor time or power due to their size or complexity can be added to the pipeline to speed up processing.

1.1. How pipelining works

Without a pipeline, the processor would get the first instruction from memory and perform the operation it calls for. It would then get the next instruction from memory and so on. While fetching the instruction, the arithmetic part of the processor is idle, which means it must wait until it gets the next instruction. This delays processing and introduces latency.

With pipelining, the next instructions can be fetched even while the processor is performing arithmetic operations. These instructions are held in a buffer close to the processor until the operation for each instruction is performed. This staging of instruction fetching happens continuously, increasing the number of instructions that can be performed in a given period.

Within the pipeline, each task is subdivided into multiple successive subtasks. A pipeline phase is defined for each subtask to execute its operations. Like a manufacturing assembly line, each stage or segment receives its input from the previous stage and then transfers its output to the next stage. The process continues until the processor has executed all the instructions and all subtasks are completed.

In the pipeline, each segment consists of an input register that holds data and a combinational circuit that performs operations. The output of the circuit is then applied to the input register of the next segment of the pipeline. Here are the steps in the process:

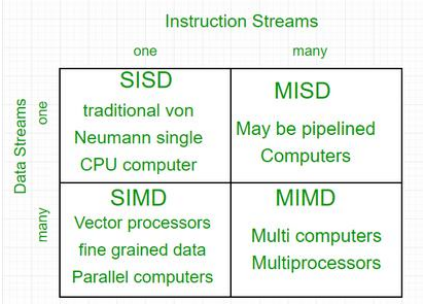
1. Fetch instructions from memory.
2. Read the input register, and decode the instruction.
3. Execute the instruction.
4. Access an operand in data memory.
5. Write the result of the operation into the input register of the next segment.

2. Flynn's taxonomy for Parallel Processing

Parallel computing is a computing where the jobs are broken into discrete parts that can be executed concurrently. Each part is further broken down to a series of instructions. Instructions from each part execute simultaneously on different CPUs.

Parallel systems deal with the simultaneous use of multiple computer resources that can include a single computer with multiple processors, a number of computers connected by a network to form a parallel processing cluster or a combination of both. Parallel systems are more difficult to program than computers with a single processor because the architecture of parallel computers varies accordingly and the processes of multiple CPUs must be coordinated and synchronized.

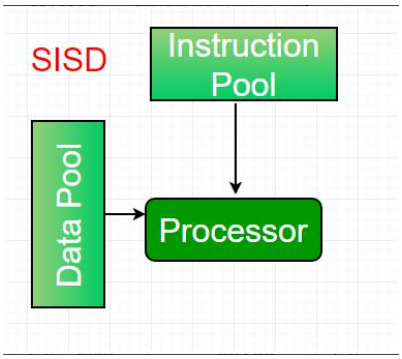
The crux of parallel processing are CPUs. Based on the number of instruction and data streams that can be processed simultaneously, computing systems are classified into four major categories:



Flynn's classification –

2.1. Single-instruction, single-data (SISD) systems –

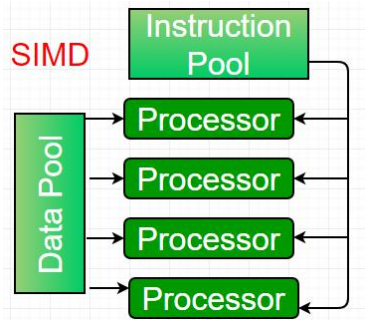
An SISD computing system is a uniprocessor machine which is capable of executing a single instruction, operating on a single data stream. In SISD, machine instructions are processed in a sequential manner and computers adopting this model are popularly called sequential computers. Most conventional computers have SISD architecture. All the instructions and data to be processed have to be stored in primary memory.



The speed of the processing element in the SISD model is limited(dependent) by the rate at which the computer can transfer information internally. Dominant representative SISD systems are IBM PC, workstations.

2.2. Single-instruction, multiple-data (SIMD) systems –

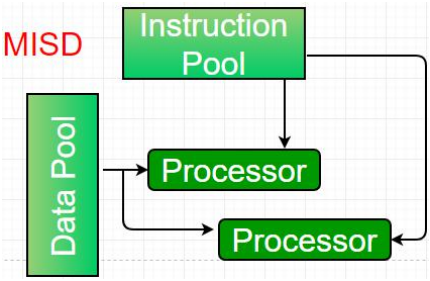
An SIMD system is a multiprocessor machine capable of executing the same instruction on all the CPUs but operating on different data streams. Machines based on an SIMD model are well suited to scientific computing since they involve lots of vector and matrix operations. So that the information can be passed to all the processing elements (PEs) organized data elements of vectors can be divided into multiple sets(N-sets for N PE systems) and each PE can process one data set.



Dominant representative SIMD systems is Cray's vector processing machine.

2.3. Multiple-instruction, single-data (MISD) systems –

An MISD computing system is a multiprocessor machine capable of executing different instructions on different PEs but all of them operating on the same dataset .

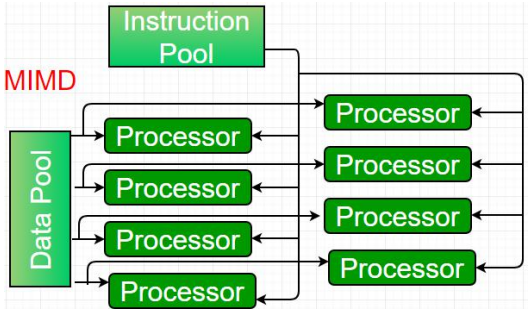


Example Z = sin(x)+cos(x)+tan(x)

The system performs different operations on the same data set. Machines built using the MISD model are not useful in most of the application, a few machines are built, but none of them are available commercially.

2.4. Multiple-instruction, multiple-data (MIMD) systems –

An MIMD system is a multiprocessor machine which is capable of executing multiple instructions on multiple data sets. Each PE in the MIMD model has separate instruction and data streams; therefore machines built using this model are capable to any kind of application. Unlike SIMD and MISD machines, PEs in MIMD machines work asynchronously.



MIMD machines are broadly categorized into **shared-memory MIMD** and **distributed-memory MIMD** based on the way PEs are coupled to the main memory.

In the **shared memory MIMD** model (tightly coupled multiprocessor systems), all the PEs are connected to a single global memory and they all have access to it. The communication between PEs in this model takes place through the shared memory, modification of the data stored in the global memory by one PE is visible to all other PEs. Dominant representative shared memory MIMD systems are Silicon Graphics machines and Sun/IBM's SMP (Symmetric Multi-Processing).

In **Distributed memory MIMD** machines (loosely coupled multiprocessor systems) all PEs have a local memory. The communication between PEs in this model takes place through the interconnection network (the inter process communication channel, or IPC). The network connecting PEs can be configured to tree, mesh or in accordance with the requirement.

The shared-memory MIMD architecture is easier to program but is less tolerant to failures and harder to extend with respect to the distributed memory MIMD model. Failures in a shared-memory MIMD affect the entire system, whereas this is not the case of the distributed model, in which each of the PEs can be easily isolated. Moreover, shared memory MIMD architectures are less likely to scale because the addition of more PEs leads to memory contention. This is a situation that does not happen in the case of distributed memory, in which each PE has its own memory. As a result of practical outcomes and user's requirement , distributed memory MIMD architecture is superior to the other existing models.

3. Types of pipelines

There are two types of pipelines in computer processing.

3.1. Instruction pipeline

The instruction pipeline represents the stages in which an instruction is moved through the various segments of the processor, starting from fetching and then

buffering, decoding and executing. One segment reads instructions from the memory, while, simultaneously, previous instructions are executed in other segments. Since these processes happen in an overlapping manner, the throughput of the entire system increases. The pipeline's efficiency can be further increased by dividing the instruction cycle into equal-duration segments.

An instruction pipeline receives sequential instructions from memory while prior instructions are implemented in other portions. Pipeline processing can be seen in both the data and instruction streams.

Pipeline processing can happen not only in the data stream but also in the instruction stream. To perform tasks such as fetching, decoding and execution of instructions, most digital computers with complicated instructions would require an instruction pipeline.

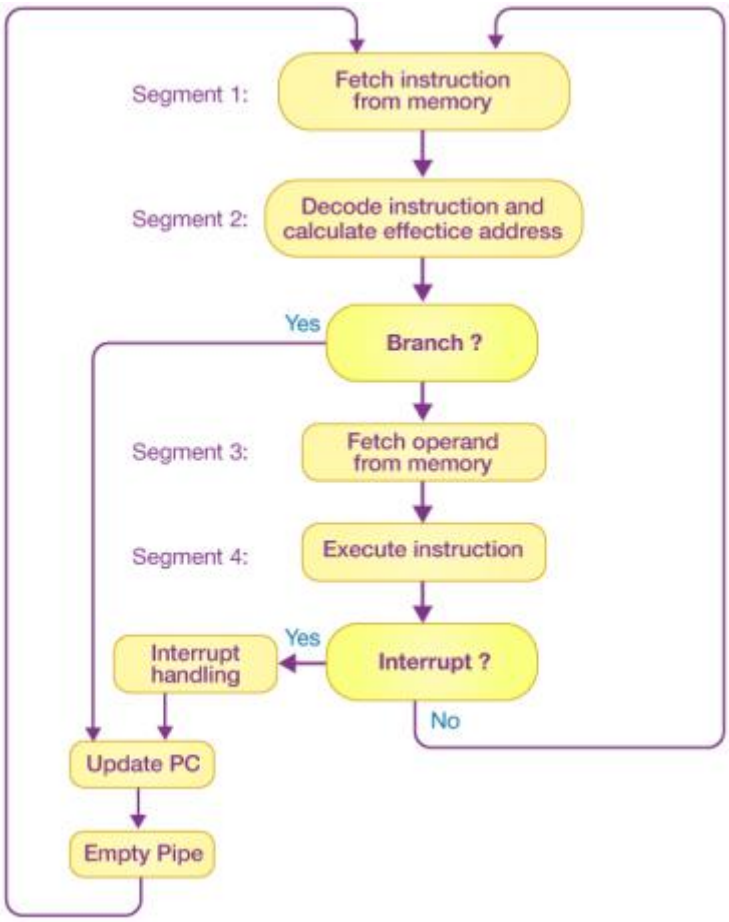
In general, each and every instruction must be processed by the computer in the following order:

1. Fetching the instruction from memory
2. Decoding the obtained instruction
3. Calculating the effective address
4. Fetching the operands from the given memory
5. Execution of the instruction
6. Storing the result in a proper place

Each step is carried out in its own segment, and various segments may take different amounts of time to process the incoming data. Furthermore, there are occasions when multiple segments request memory access at the very same time, requiring one segment to wait unless and until the memory access of another is completed.

If the instruction cycle is separated into equal-length segments, the organization of an instruction pipeline will become much more efficient. A four-segment type of instruction pipeline refers to one of the most common instances of this style of organization.

A four-segment instruction pipeline unifies two or more distinct segments into a single unit. For example, the decoding of the instruction and the calculation of the effective address can be merged into a single segment.



A four-segment instruction pipeline is illustrated in the block diagram given above. The instructional cycle is divided into four parts:

- **Segment 1:** The implementation of the instruction fetch segment can be done using the FIFO or first-in, first-out buffer.
- **Segment 2:** In the second segment, the memory instruction is decoded, and the effective address is then determined in a separate arithmetic circuit.
- **Segment 3:** In the third segment, some operands would be fetched from memory.
- **Segment 4:** The instructions would finally be executed in the very last segment of a pipeline organization.

3.2. Arithmetic pipeline

The arithmetic pipeline represents the parts of an arithmetic operation that can be broken down and overlapped as they are performed. It can be used for used for arithmetic operations, such as floating-point operations, multiplication of fixed-point numbers, etc. Registers are used to store any intermediate results that are then passed on to the next stage for further processing.

Arithmetic Pipelines are commonly used in various high-performance computers. They are used in order to implement floating-point operations, fixed-point

multiplication, and other similar kinds of calculations that come up in scientific situations.

Let's look at an example to better understand the ideas of an arithmetic pipeline. We perform addition and subtraction of floating points on a unit of the pipeline here.

The inputs in the floating-point adder pipeline refer to two different normalized floating-point binary numbers. These are defined as follows:

$A = X * 2^x = 0.9504 * 10^3$

$B = Y * 2^y = 0.8200 * 10^2$

Where x and y refer to the exponents and X and Y refer to two fractions representing the mantissa.

The floating-point addition and subtraction process is broken into four pieces. The matching sub-operation to be executed in the specified pipeline is contained in each segment. The four segments depict the following sub-operations:

1. Comparing the exponents using subtraction
2. Aligning the mantissa
3. Adding or subtracting the mantissa
4. Normalizing the result

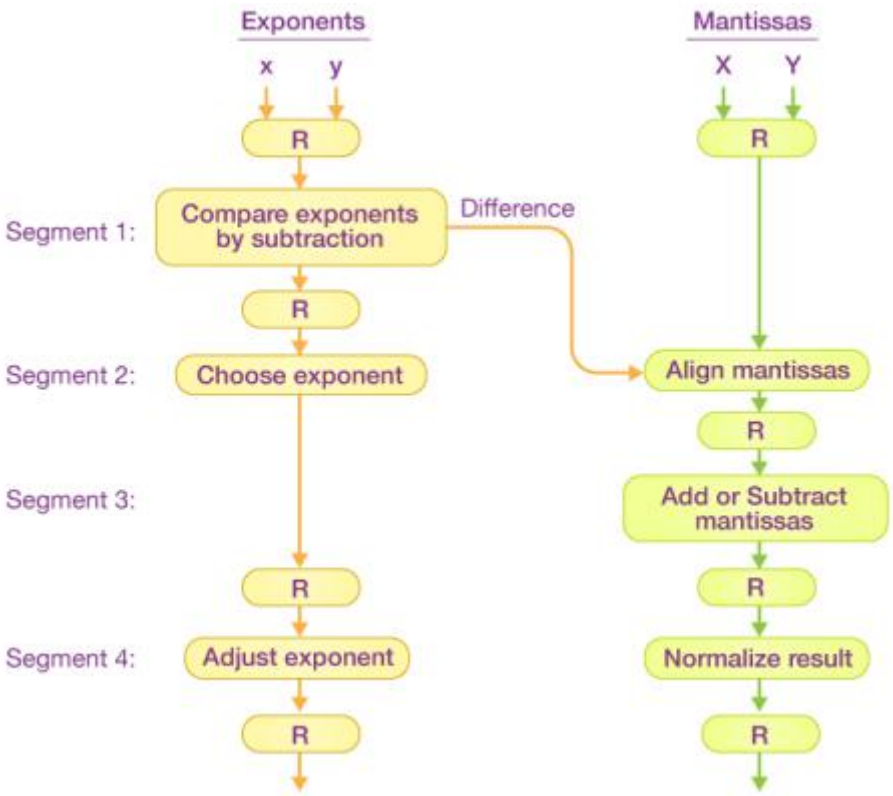


Figure 3.1. Arithmetic Pipeline Flowchart

A. Comparing Exponents by Subtraction

The difference between the exponents is calculated by subtracting them. The result's exponent is chosen to be the larger exponent.

The exponent difference, 3 – 2 = 1, defines the total number of times the mantissa associated with the lesser exponent should be shifted to the right.

B. Aligning the Mantissa

As per the difference of exponents calculated in segment one, the mantissa corresponding with the smaller exponent would be moved.

$A = 0.9504 * 10^3$

$B = 0.08200 * 10^3$

C. Adding the Mantissa

Both the mantissa would be added in the third segment.

$C = A + B = 1.0324 * 10^3$

D. Normalizing the Result

After the process of normalization, the result would be written as follows:

$C = 0.1324 * 10^4$

4. Measuring performance of pipeline

The ability to overlap stages of a sequential process for different input tasks (data or operations) results in an overall theoretical completion time of

$$\mathbf{T_{pipe}=m.P+(n-1).P}$$

Here n is the number of input tasks, m is the number of stages in the pipeline, and P is the clock.

The three basic performance measures for the pipeline are as follows:

A. Speed up:

K-stage pipeline processes n tasks in k + (n-1) clock cycles: k cycles for the first task and n-1 cycles for the remaining n-1 tasks

Total time to process n tasks $T_k = [k + (n-1)]$

For the non-pipelined processor $T_1 = n\ k$

Speed up factor is

speedup=(Timetakenbynonpipelineimplemention)/(Timetakenbypipelinedimplementation)

Speedup=nky/(K+(n−1))y=nk/(k+(n−1))

n=α

speedup=nk/n=k

As the K stage reaches n and the value of n is approximated to infinity. The speedup factor is equivalent to K

B. Through-put:

Throughput is the outputs produced per clock cycle and that throughput will be equal to 1, in case of ideal situation that means, when the pipeline is producing one output per clock cycle.

Formula is U(n)= m*f / n+(m-1)

C. Efficiency:

The efficiency of n stages in a pipeline is defined as ratio of the actual speedup to he maximum speed.

Formula is E(n)= m / n+m-1

5. Types of Hazard

As we all know, the CPU's speed is limited by memory. There's one more case to consider, i.e. a few instructions are at some stage of execution in a pipelined design. There is a chance that these sets of instructions will become dependent on one another, reducing the pipeline's pace. Dependencies arise for a variety of reasons, which we will examine shortly. The dependencies in the pipeline are referred to as hazards since they put the execution at risk.

We can swap the terms, dependencies and hazards since they are used interchangeably in computer architecture. A hazard, in essence, prevents an instruction present in the pipe from being performed during the specified clock cycle. Since each of the instructions may be in a separate machine cycle, we use the term clock cycle.

The three different types of hazards in computer architecture are:

1. Structural
2. Data
3. Control

Dependencies can be addressed in a variety of ways. The easiest is to introduce a bubble into the pipeline, which stalls it and limits throughput. The bubble forces the next instruction to wait until the previous one is completed.

- **Structural Hazard:** Hardware resource conflicts among the instructions in the pipeline cause structural hazards. Memory, a GPR Register, or an ALU might all be used as resources here. When more than one instruction in the pipe requires access to the very same resource in the same clock cycle, a resource conflict is said to arise. In an overlapping pipe-lined execution, this is a circumstance where the hardware cannot handle all potential combinations. Know more about structural hazards here.
- **Data Hazards:** Data hazards in pipelining emerge when the execution of one instruction is dependent on the results of another instruction that is still being processed in the pipeline. The order of the READ or WRITE operations on the register is used to classify data threats into three groups. Know more about data hazards here.
- **Control Hazards:** Branch hazards are caused by branch instructions and are known as control hazards in computer architecture. The flow of program/instruction execution is controlled by branch instructions. Remember that conditional statements are used in higher-level languages for iterative loops and condition testing (correlate with while, for, and if case statements). These are converted into one of the BRANCH instruction variations. As a result, when the decision to execute one instruction is reliant on the result of another instruction, such as a conditional branch, which examines the condition's consequent value, a conditional hazard develops. Know more about control hazards in pipelining here.

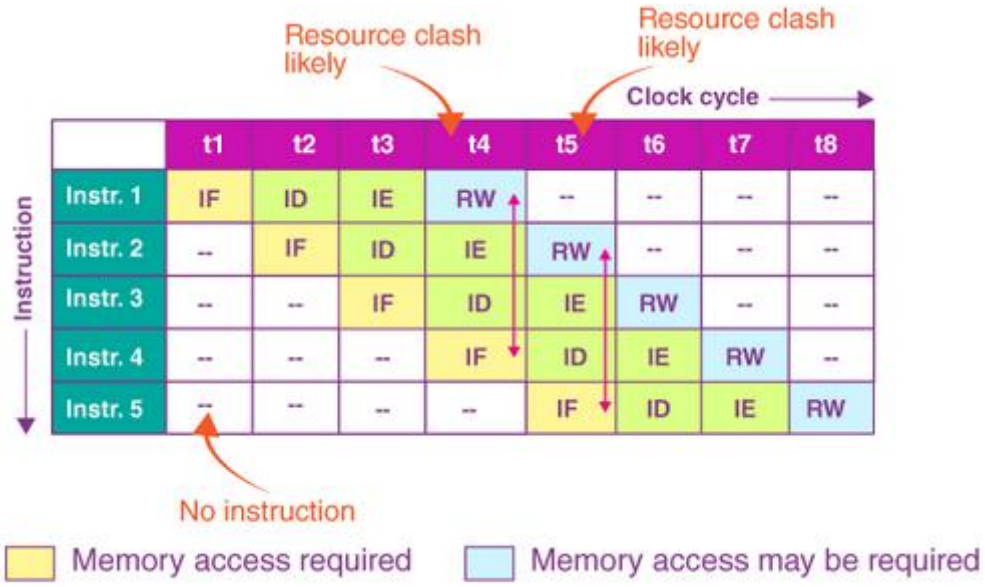
6. Structural Hazards

When two (or more) instructions in the pipeline require the same resource, a structural hazard occurs. As a result, for a portion of the pipeline, instructions must be performed in series rather than parallel. Occasionally, structural dangers are considered to be resource hazards.

6.1. What are Structural Hazards?

Hardware resource conflicts among the instructions in the pipeline cause structural hazards. Memory, a GPR Register, or an ALU might all be used as resources here.

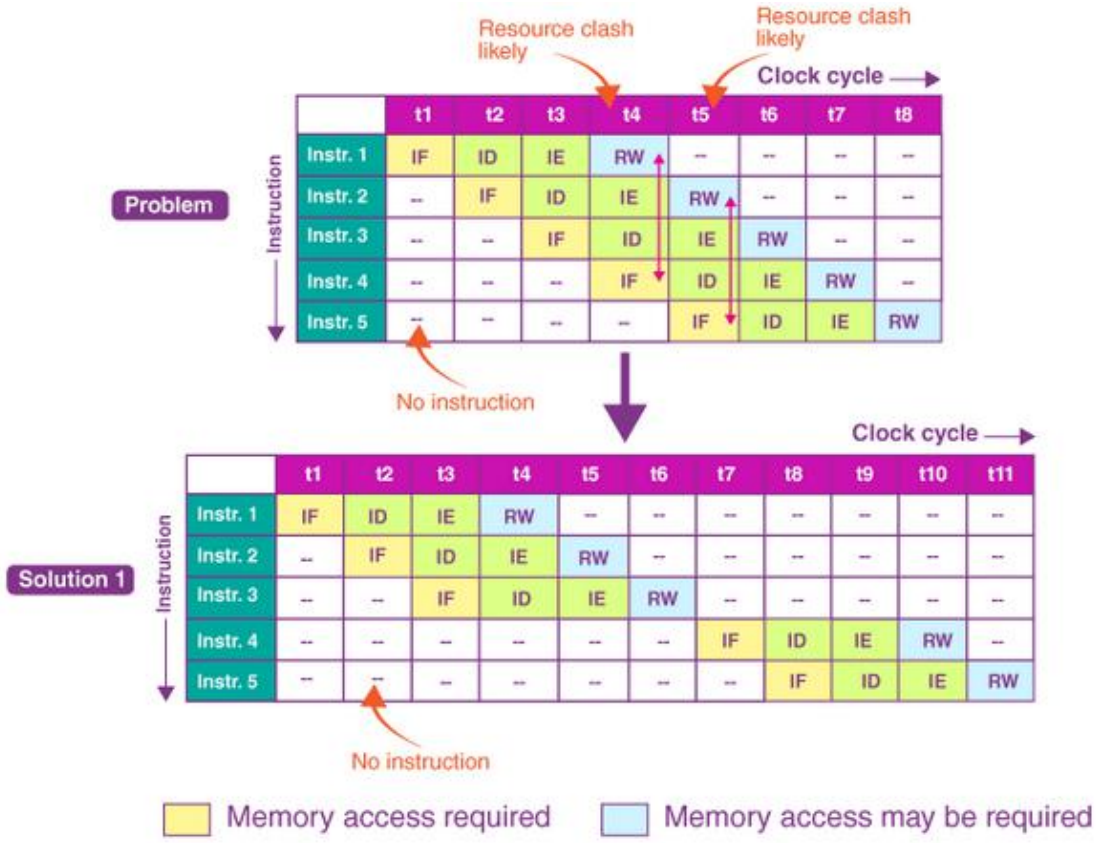
When more than one instruction in the pipe requires access to the very same resource in the same clock cycle, a resource conflict is said to arise. In an overlapping pipe-lined execution, this is a circumstance where the hardware cannot handle all potential combinations.



Take a look at the illustration above. In an IF machine cycle, instructions are retrieved from memory in any system. Depending on the instruction, Result Writing (RW) in the 4-stage pipeline may access memory or one General Purpose Register. Instruction-1(I1) is in the RW stage at t4, while Instruction-4(I4) is in the IF stage. Alas! If I1 is a STORE instruction, both I4 and I1 are accessing the same resource, which is memory. In a timed condition, how can it be possible to access memory with two commands from the same CPU? Impossible. This is referred to as structural dependency. What would be the solution?

6.2. Solution

The figure above shows a bubble that blocks the pipeline. I4 isn't allowed to proceed at t4 and is instead delayed. It might have been allowed in t5, but there would have been a conflict with I2 RW. I4 is not allowed in t6 for much the same reason. Finally, only at t7 could I4 be allowed to progress (stalled) in the pipe.



This delay is passed on to all future commands as well. As a result, while the ideal 4-stage system would require 8 timing states to execute 5 instructions, it instead takes 11 timing states due to structural dependency. This isn't it. You've probably figured out that this threat will occur in every fourth instruction. This is not a good solution for a large CPU load. Is there a better way to do things? Yes!

Here, a better solution would be to boost the system's structural resources using one of the options listed below:

- The pipeline can be expanded to five or more stages, with the functionality of the stages appropriately redefined and the clock frequency adjusted. This resolves the hazard at every fourth instruction in the four-stage pipeline.
- Instruction memory and Data Memory are two types of memory that can be physically separated. Instead of dealing with Main memory, it would be better to build Cache memory in the CPU. Instruction memory is used in IF, and Data Memory is used in Result writing. These two resources become independent of one another, eliminating reliance.
- Multiple levels of cache in the CPU are also possible.
- There is a chance that ALU will become resource-dependent. Instructions in the IE machine cycle may require ALU, whereas another instruction in the IF stage

may require ALU to calculate Effective Address dependent on addressing mode. Either stalling or having an exclusive ALU for the calculation of address would be the solution.

- GPRs are replaced with registered files. Register files feature exclusive read and write ports and multiport access. This allows access to one write register, and one read register at the same time.

7. Data Hazards

Data hazard, structural hazard, and control hazard are three categories of common hazards. Hazards in the domain of central processing unit (CPU) design are problems with the instruction pipeline in CPU micro-architectures when the next instruction cannot execute in the next clock cycle, which might possibly result in inaccurate calculation results.

In recent CPUs, the last two approaches have been implemented. If dependency develops beyond this, the only choice is to stall. Keep in mind that acquiring more resources comes at a higher price. As a result, the trade-off is a designer's decision.

Data hazard arises if an instruction accesses a register that a preceding instruction overwrites in a future cycle. Unless we decrease data hazard, pipelines will produce erroneous outputs. In this article, we shall dig further into Data Hazards.

7.1. What is a Data hazard?

Data hazard in pipelining arises when one instruction is dependent on the results of a preceding instruction and that result has not yet been calculated. Whenever two distinct instructions make use of the same storage. The location must seem to be run sequentially.

In other words, a Data hazard in computer architecture occurs when instructions with data dependency change data at several stages of a pipeline. Ignoring possible data hazards might lead to race conditions (also termed race hazards).

7.2. Types of Data Hazard in Pipelining

The data hazard or dependency exists in the instructions in the three types that are as follows:

- Read after Write (RAW)
- Write after Read (WAR)
- Write after Write (WAW)

7.2.1. Read after Write (RAW)

Read after Write(RAW) is also known as True dependency or Flow dependency. A read-after-write (RAW) data hazard is when an instruction refers to a result that has not yet been computed or retrieved. This can happen because, even when an instruction is executed after another, the previous instruction has only been processed partially through the pipeline.

For example, consider the two instructions:

- **R2 <- R5 + R3**
- **R4 <- R2 + R3**

The first instruction computes a value to be saved in register R2, and the second uses this value to compute a result for register R4. However, in a pipeline, when the operands for the second operation are retrieved, the results of the first operation have not yet been stored, resulting in a data dependence.

Instruction I2 has a data dependence since it is dependent on the execution of instruction I1.

7.2.2. Write after Read (WAR)

Write after Read(WAR) is also known as anti-dependency. These data hazards arise when an instruction's output register is utilized immediately after being read by a previous instruction.

As an example, consider the two instructions:

- **R4 <- R1 + R5**
- **R5 <- R1 + R2**

When there is a chance that I2 will end before I1 (i.e., when there is concurrent execution), it must be assured that the result of register R5 is not saved before I1 has had a chance to obtain the operands.

7.2.3. Write after Write (WAW)

Write after Write(WAW) is also known as output dependency. These data hazards arise when the output register of instruction is utilized for write after the previous instruction has been written.

As an example, consider the two instructions:

- R2 <- R4 + R7
- R2 <- R1 + R3

The write-back (WB) of I2 must be postponed until I1 has completed its execution.

7.3. Handling Data Hazard

There are various methods to handle the data hazard in computer architecture that occur in the program. Some of the methods to handle the data hazard are as follows:

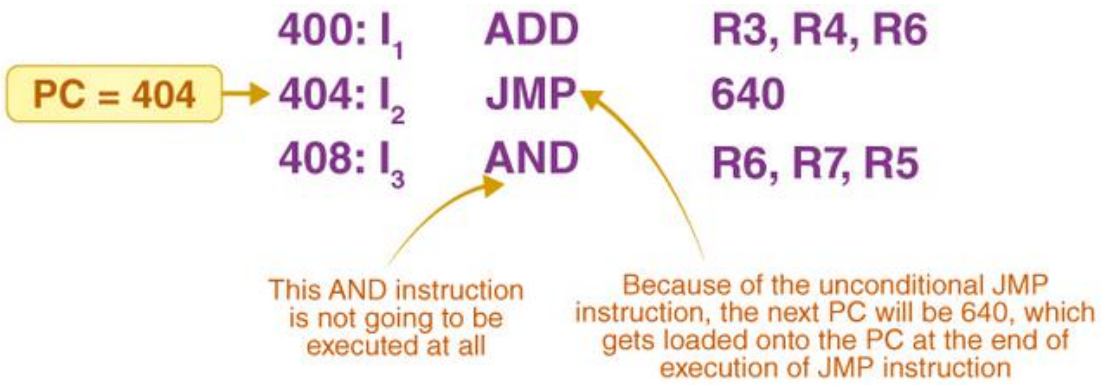
- Forwarding is the addition of specific circuitry to the pipeline. This approach works because the needed values take less time to travel via a wire than it does for a pipeline segment to compute its result.
- Code reordering requires the use of specialized software. This sort of software is known as a hardware-dependent compiler.
- Stall Insertion inserts one or more installs(no-op instructions) into the pipeline, delaying execution of the current instruction until the needed operand is written to the register file; unfortunately, this method reduces pipeline efficiency and throughput.

8. Control Hazard

Branch hazards are caused by branch instructions and are known as control hazards. The flow of program/instruction execution is controlled by branch instructions. In higher-level languages, conditional statements are used for repetitive loops or condition testing (correlate with while, for, if, case statements). These are converted into one of the BRANCH instruction variations. To understand the programme flow, you must know the value of the condition being tested. It's a difficult situation.

As a result, when the decision to execute one instruction is reliant on the result of another instruction, such as a conditional branch, which examines the condition's consequent value, a conditional hazard develops.

The Program Counter (PC) is loaded with the appropriate place for the branch and jump instructions, which determines the programme flow. The next instruction that is to be fetched and executed by the given CPU is stored in the PC. Take a look at the instructions that follow:



In this scenario, fetching the I3 is pointless. What is the status of the pipeline? The I3 fetch must be terminated while in I2. This can only be determined once I2 has been decoded as JMP. As a result, the pipeline cannot continue at its current rate, resulting in a Control Dependency (hazard). If I3 is fetched in the meantime, it is not just unnecessary work, but it is also possible that some data in registers has been changed and needs to be reversed.

Identical scenarios arise in the case of conditional JMP or BRANCH.

9. Method for Avoiding Hazard or Conditional Hazards Solutions

9.1. Stall

Stall the given pipeline as soon as any branch instructions are decoded. Just don't allow IF anymore. Stalling reduces throughput as it always does. According to statistics, at least 30% of the instructions in a program are BRANCH. With Stalling, the pipeline is effectively operating at 50% capacity.

9.2. Prediction

Consider a for or a while loop that is repeated 100 times. We know the programme would run 100 times without the given branch condition being met. The program only exits the loop for the 101st time. As a result, it's better to let the pipeline run its course and then flush/undo when the branch condition is met. This has less of an impact on the pipeline's throttle and stalling.

9.3. Dynamic Branch Prediction

With the help of Branch Table Buffer, a historical record is kept (BTB). The BTB is a type of cache that contains a series of entries containing the branch instruction's PC address and the effective branch address. This is done for each branch instruction that is encountered. When a conditional branch instruction is encountered, the BTB is queried for the matching branch instruction address. If the target branch address is hit, the next instruction is fetched from the associated target branch address. Dynamic branch prediction is the term for this.

Branch Instruction Address	Target Branch Address Taken

Branch Table Buffer

This method works to the extent that the program's temporal locality of reference allows it. When the prediction fails, flushing is required.

9.4. Reordering Instructions

Delayed branching entails reordering the instructions to move the branch instruction later in the sequence, allowing safe and beneficial instructions that are unaffected by the result of a branch to be brought in earlier in the sequence, delaying the fetch of the branch instruction. If such instructions are not available, NOP is used. The Compiler is used to implement this delayed branch.

Last but not least, the Control unit in a pipelined design is intended to handle the following scenarios:

- Dependence requiring stall
- No dependence
- Dependence solution by forwarding
- Out of order execution
- Dependence with access in order
- Branch prediction table and more

10. Vector Processing

Vector processing is a central processing unit that can perform the complete vector input in individual instruction. It is a complete unit of hardware resources that implements a sequential set of similar data elements in the memory using individual instruction.

The scientific and research computations involve many computations which require extensive and high-power computers. These computations when run in a conventional computer may take days or weeks to complete. The science and engineering problems can be specified in methods of vectors and matrices using vector processing.

Features of Vector Processing

There are various features of Vector Processing which are as follows −

- A vector is a structured set of elements. The elements in a vector are scalar quantities. A vector operand includes an ordered set of n elements, where n is known as the length of the vector.
- Each clock period processes two successive pairs of elements. During one single clock period, the dual vector pipes and the dual sets of vector functional units allow the processing of two pairs of elements.

- As the completion of each pair of operations takes place, the results are delivered to appropriate elements of the result register. The operation continues just before the various elements processed are similar to the count particularized by the vector length register.
- In parallel vector processing, more than two results are generated per clock cycle. The parallel vector operations are automatically started under the following two circumstances −
  - When successive vector instructions facilitate different functional units and multiple vector registers.
  - When successive vector instructions use the resulting flow from one vector register as the operand of another operation utilizing a different functional unit. This phase is known as chaining.
- A vector processor implements better with higher vectors because of the foundation delay in a pipeline.
- Vector processing decrease the overhead related to maintenance of the loop-control variables which creates it more efficient than scalar processing.

11. Array Processors

A processor that performs computations on a vast array of data is known as an array processor. Multiprocessors and vector processors are other terms for array processors. It only executes one instruction at a time on an array of data. They work with massive data sets to perform computations. Hence, they are used to enhance the computer's performance.

11.1. Classification of Array Processors

Array processors can be divided into two categories:

1. Attached Array Processors
2. SIMD(Single Instruction Stream, Multiple Data Stream) Array Processors

11.1.1. Attached Array Processor

The attached array processor is the auxiliary processor connected to a general-purpose computer to enhance and improve the machine's performance in numerical computational tasks. It provides excellent performance by using numerous functional units in parallel processing.

The attached array processor includes a common processor with an input/output interface and a local memory interface.

The main memory and the local memory are linked.

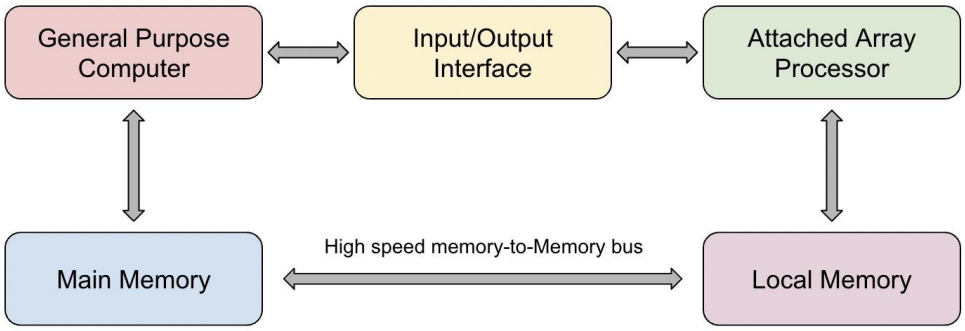


Fig- The interconnection of Attached Array Processor to Host computer

The attached array processor intends to improve the performance of the host computer in specific numeric computations.

11.1.2. SIMD Array Processor

SIMD refers to the organization of a single computer with multiple parallel processors. The processing units are designed to work together under the supervision of a single control unit, resulting in a single instruction stream and multiple data streams.

An array processor's general block diagram is given below. It comprises several identical processing elements (PEs), each with its local memory M. An ALU and registers are included in each processor element. The master control unit controls the processing elements' actions. It also decodes instructions and determines how they should be carried out.

The program is stored in the main memory. The control unit retrieves the instructions. Vector instructions are sent to all PEs simultaneously, and the results are stored in memory.

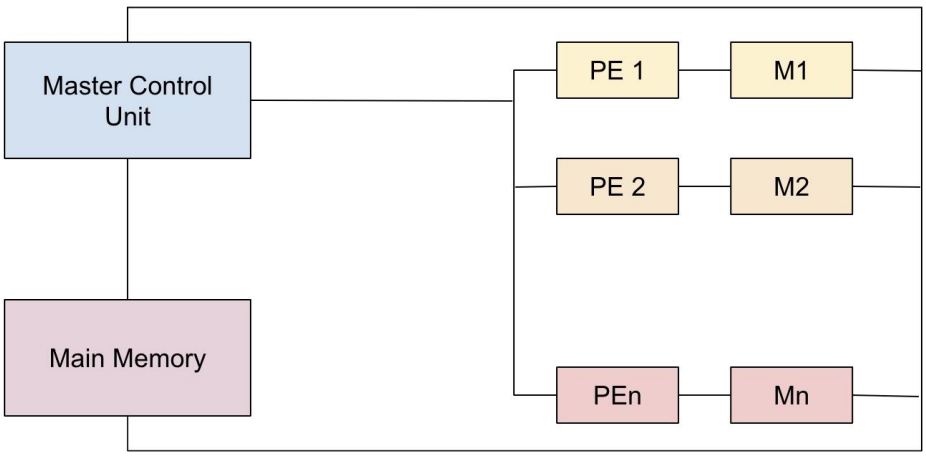


Fig-Array Processor Organisation in SIMD

The ILLIAC IV computer, manufactured by the Burroughs Corporation, is the most well-known SIMD array processor. Single Instruction Multiple Data (SIMD) processors are highly specialized computers. They're only good for numerical issues that can be stated as vectors or matrices; they're not suitable for other kinds of computations.

11.1.2.1. Configurations of SIMD

1. Array processors that use RAM(Random Access Memory) are also known as Dedicated Memory Organization.
   - ILLIAC-IV
   - CM-2
   - MP-1
2. Associative processor that uses content accessible memory is known as Global Memory Organization.
   - BSP

11.1.2.2. Usage of Array Processors

- Array processors enhance the total speed of instruction processing.
- Most array processors' design optimizes its performance for repetitive arithmetic operations, making it faster at vector arithmetic than the host CPU. Since most Array processors run asynchronously from the host CPU, the system's overall capacity is thus improved.
- Array Processors have their own local memory, providing additional extra memory to systems with limited memory. This is an essential consideration for the systems with a limited physical memory or address space.

11.1.2.3. Applications of Array Processors

Array processing is used at various places, including:-

- Radar Systems
- Sonar Systems
- Anti-jamming
- Seismic Exploration
- Wireless communication
- Medical applications
- Used for Speech Enhancement
- Used in Astronomy applications

Array processors are extremely useful for dealing with problems that require a lot of parallelisms. However, they do require a change in programming methodology. Converting conventional (sequential) programs to support array processors is complex, and different (parallel) algorithms may be needed to match the parallel approach.

12. Parallel Processing

Parallel processing can be described as a class of techniques which enables the system to achieve simultaneous data-processing tasks to increase the computational speed of a computer system.

A parallel processing system can carry out simultaneous data-processing to achieve faster execution time. For instance, while an instruction is being processed in the ALU component of the CPU, the next instruction can be read from memory.

The primary purpose of parallel processing is to enhance the computer processing capability and increase its throughput, i.e. the amount of processing that can be accomplished during a given interval of time.

A parallel processing system can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. The data can be distributed among various multiple functional units.

The following diagram shows one possible way of separating the execution unit into eight functional units operating in parallel.

The operation performed in each functional unit is indicated in each block if the diagram:

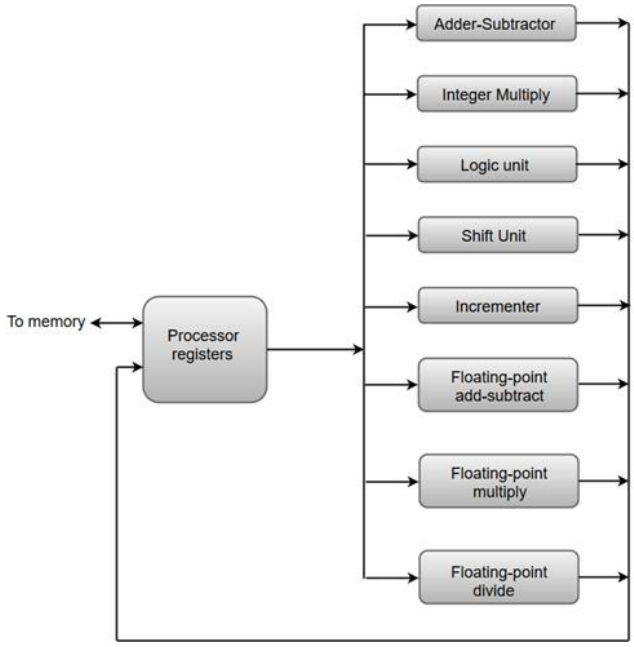


Figure 12.1. Parallel Processing with Data from Processor Register

- The adder and integer multiplier performs the arithmetic operation with integer numbers.
- The floating-point operations are separated into three circuits operating in parallel.
- The logic, shift, and increment operations can be performed concurrently on different data. All units are independent of each other, so one number can be shifted while another number is being incremented.