UNIT – 5

PART – 1 : PIPELINE AND VECTOR PROCESSING

Flynn's Taxonomy:

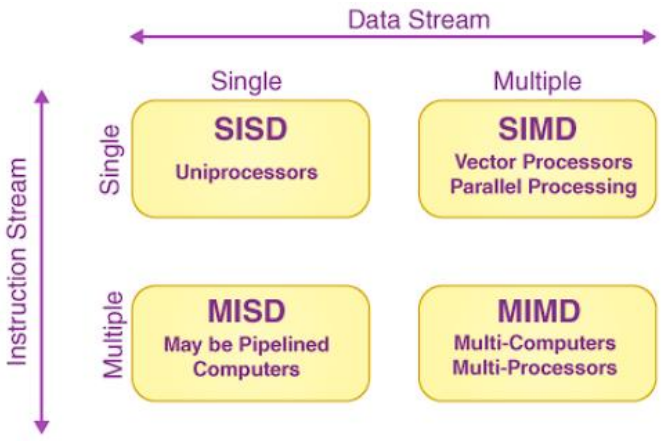
Flynn's taxonomy is a classification scheme for computer architectures proposed by Michael Flynn in 1966. The taxonomy is based on the number of instruction streams and data streams that can be processed simultaneously by a computer architecture.

An instruction stream is a collection of instructions read from memory. A data stream is the result of the actions done on the data in the processor.

The term 'stream' refers to the flow of data or instructions.

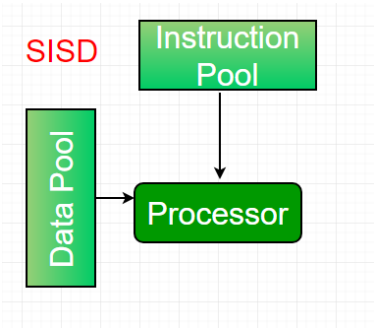
Parallel processing can happen in the data stream, the instruction stream, or both.

Flynn's Classification:



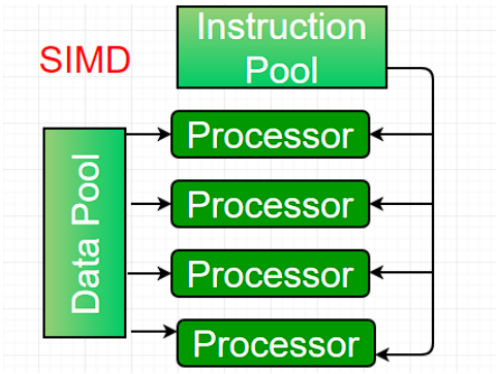
Flynn's Classification of Computers

1. **Single-instruction, single-data (SISD) systems –** An SISD computing system is a uniprocessor machine that 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.



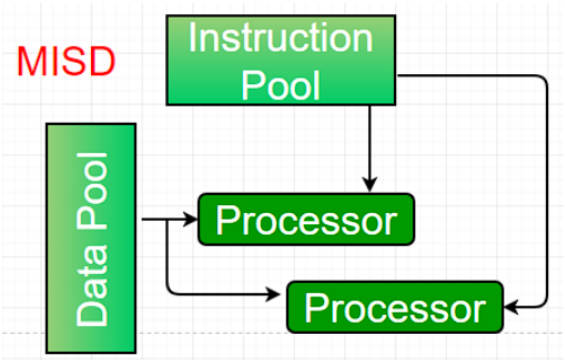
The speed of the processing element in the SISD model is limited(dependent) by the rate at which the computer can transfer information internally. Dominant representative SISD systems are IBM PC, and workstations.

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 a 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.



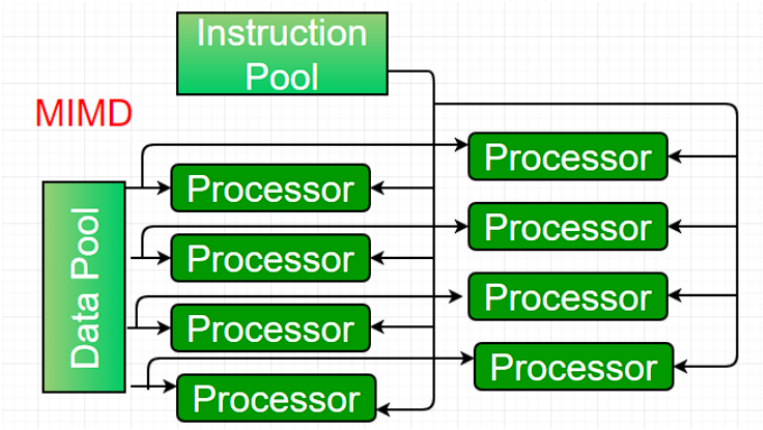
Dominant representative SIMD systems are Cray's vector processing machines.

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 operate 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 applications, a few machines are built, but none of them are available commercially.

4. **Multiple-instruction, multiple-data (MIMD) systems –** An MIMD system is a multiprocessor machine that 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 of any 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. The 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 in 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 requirements, distributed memory MIMD architecture is superior to the other existing models.

Flynn's taxonomy itself does not have any inherent advantages or disadvantages. It is simply a classification scheme for computer architectures based on the number of instruction streams and data streams that can be processed simultaneously.

However, the different types of computer architectures that fall under Flynn's taxonomy have their own advantages and disadvantages. Here are some examples:

1. SISD architecture: This is the simplest and most common type of computer architecture. It is easy to program and debug and can handle a wide range of applications. However, it does not offer significant performance gains over traditional computing systems.
2. SIMD architecture: This type of architecture is highly parallel and can offer significant performance gains for applications that can be parallelized. However, it requires specialized hardware and software and is not well-suited for applications that cannot be parallelized.
3. MISD architecture: This type of architecture is not commonly used in practice, as it is difficult to find applications that can be decomposed into independent instruction streams.
4. MIMD architecture: This type of architecture is highly parallel and can offer significant performance gains for applications that can be parallelized. It is well-suited for distributed computing, parallel processing, and other high-performance computing applications. However, it requires specialized hardware and software and can be challenging to program and debug.

Overall, the advantages and disadvantages of different types of computer architectures depend on the specific application and the level of parallelism that can be exploited. Flynn's taxonomy is a useful tool for understanding the different types of computer architectures and their potential uses, but ultimately the choice of architecture depends on the specific needs of the application.

Some additional features of Flynn's taxonomy include:

Concurrency: Flynn's taxonomy provides a way to classify computer architectures based on their concurrency, which refers to the number of tasks that can be executed simultaneously.

Performance: Different types of architectures have different performance characteristics, and Flynn's taxonomy provides a way to compare their performance based on the number of concurrent instructions and data streams.

Parallelism: Flynn's taxonomy highlights the importance of parallelism in computer architecture and provides a framework for designing and analyzing parallel processing systems.

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.

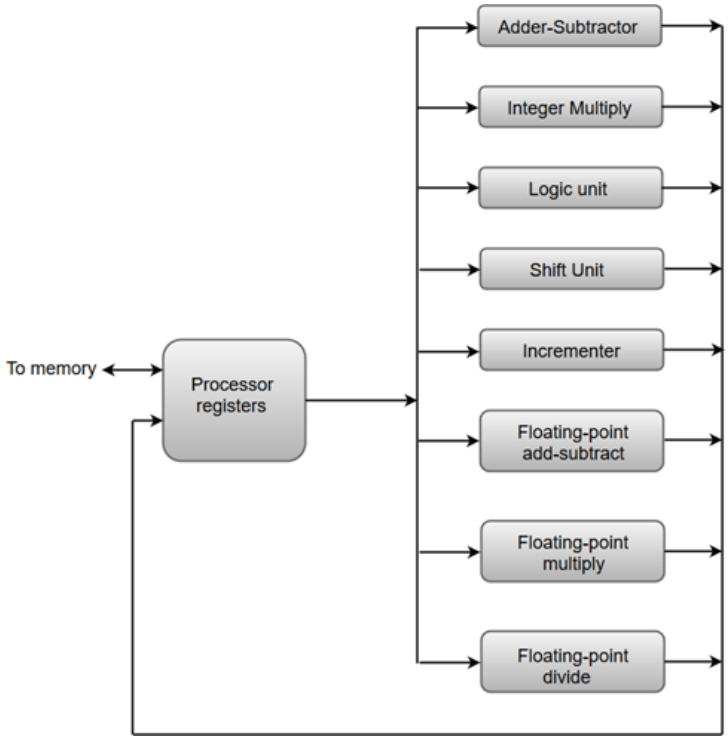
A parallel processing system can carry out simultaneous data-processing to achieve faster execution time. For instance, while an instruction is being processed in the ALU component of the CPU, the next instruction can be read from memory.

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:



- The adder and integer multiplier performs the arithmetic operation with integer numbers.
- The floating-point operations are separated into three circuits operating in parallel.

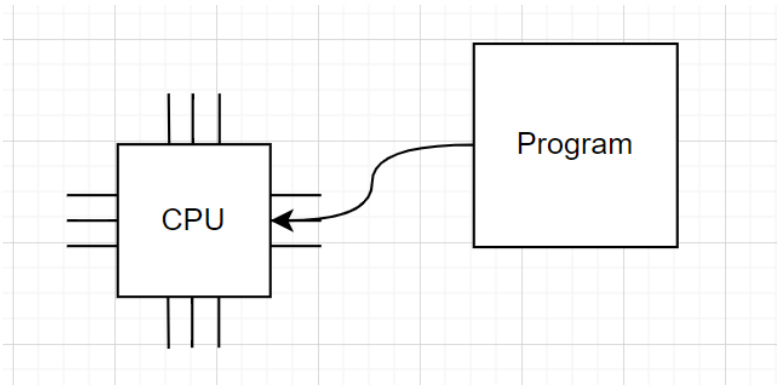
- The logic, shift, and increment operations can be performed concurrently on different data. All units are independent of each other, so one number can be shifted while another number is being incremented.

All these units will work concurrently and produce the required output.

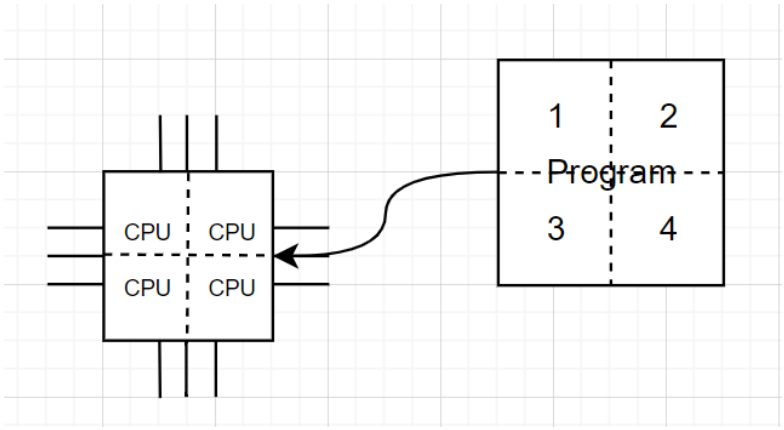
For Example:

1. The system may have two or more ALUs and can execute multiple instructions simultaneously.
2. The system may have multiple processors operating simultaneously.
3. The following instruction can be fetched while the current instruction is being executed in ALU.

The primary purpose of parallel processing is to increase the computer processing capability and its throughput. We can achieve parallel processing by using multiple functional units that can perform identical or different operations concurrently to distribute the data among these functional units.



With parallel processing



without parallel processing

Need of Parallel Processing:

1. **Application demands:** With increasing technology, modern applications require more computing cycles. Examples include videos, high graphic games, databases, science models, etc.
2. **Technology Trends:** With the increasing number of transistors on a chip, clock rates are expected to go up but slowly.
3. **Economics:** Instead of costly components used in traditional supercomputers, today's microprocessors offer high performance and have multiprocessor support.

Levels of Parallelism:

- **Instruction-Level Parallelism (ILP):** In ILP, a single task is divided into multiple instructions, and these instructions are executed simultaneously or in an overlapping fashion. Techniques like pipelining and superscalar architectures exploit ILP within a single processor.
- **Task-Level Parallelism (TLP):** TLP involves breaking down a program into multiple independent tasks that can be executed concurrently. Multiple processors or cores are used to perform these tasks simultaneously. This can be achieved through multiprocessing or multi-core architectures.
- **Data-Level Parallelism (DLP):** DLP involves processing multiple data elements concurrently. SIMD (Single Instruction, Multiple Data) architectures, vector processors, and GPU (Graphics Processing Unit) programming are examples of DLP.

Advantages of Parallel Processing:

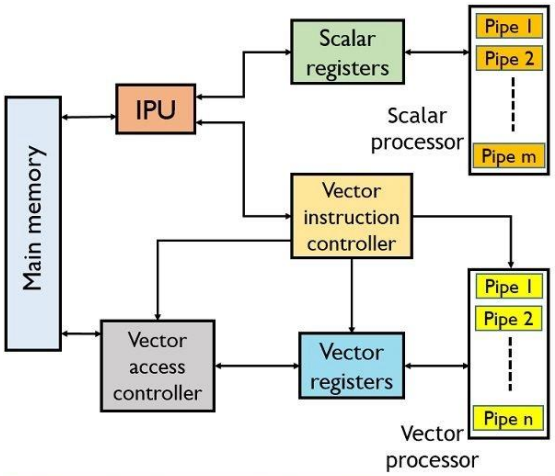
1. It increases the speed and efficiency of computers. Sequential computing forces fast processors to do things inefficiently.
2. Computers can be used to solve more complex and more extensive problems. A single web app may have to process millions of requests every second with so much data.
3. For a system that has to support billions of operations (for example, bank software), parallel processing makes things cheaper.
4. Parallel computing is more suited for hardware since serial computing wastes the computing power of processors.

Disadvantages of Parallel Processing:

1. Increases the cost of computers since more hardware is required.
2. Multicore architectures consume higher power.
3. Parallel architectures are difficult to achieve.
4. A parallel computing system needs different code tweaking depending on the target architecture.
5. It increases the overhead cost due to synchronization and data transfers.

Vector Processing:

To avoid the overhead of the processing loop, vector processing operates on all elements of the entire array in one operation, i.e. in parallel. But vector processing is possible only if operations performed in parallel are independent of each other.



Functional Diagram of Vector Computer

Characteristic of Vector Processing:

A **vector** is defined as an ordered set of a one-dimensional array of data items. A vector V of length n can be represented as a row vector by V = [V1 V2 V3 · · · Vn]. If the data items are listed in a column, it may be represented as a column vector

For a processor with multiple ALUs, it is possible to operate on multiple data elements in parallel using a single instruction. Such instructions are called single-instruction multiple-data (SIMD) instructions. They are also called **vector instructions.**

VectorAdd.S Vi, Vj, Vk

The above Vector instruction computes L sums using the elements in vector registers Vj and Vk, and places the resulting sums in vector register Vi. Similar instructions are used to perform other arithmetic operations.

VectorLoad.S Vi, X(Rj)

The above Vector instruction is to transfer multiple data elements between a vector register and the memory.

A computer capable of vector processing eliminates the overhead associated with the time it takes to fetch and execute the instructions in the program loop. It allows operations to be specified with a single vector instruction of the form

C(1: 100) = A(1: 100) + B(1: 100).

The vector instruction includes the initial address of the operands, the length of the vectors, and the operation to be performed, all in one composite instruction.

The **instruction format** of the vector processor is

Operation code	Base address source1	Base address source2	Base address destination	Vector length

This is a **three-address instruction** with three fields specifying the length of the data items in the vectors and the base address of the operands and an additional field. This assumes that the vector operands reside in memory. It is also possible to design the processor with a large number of registers and store all operands in registers prior to the addition operation. In that case, the base address and length in the vector instruction specify a group of CPU registers.

In a source program written in a high-level language, if the operations performed in each pass are independent of the other passes, loops that operate on arrays of integers or floating-point numbers are **vectorizable.**

A **vectorizing compiler** can recognize such loops and generate vector instruction if they are not too complex.

Using vector instructions reduces the number of instructions that need to be executed and enables the operations to be performed in parallel on multiple ALUs.

Based on where from where operands are retrieved, vector processors can be classified into two classes - Memory to memory and Register to register.

- **Memory to memory Vector processor**: In this type of vector processor, the operands for instruction, intermediate result, and final result all are retrieved from the main memory. Such processors are Cyber-205, TI-ASC.
- **Register to register Vector processor**: In this type of vector processor, the operands for instruction, intermediate result, and final result all are retrieved from scalar or vector registers. Such processors are Cray-1, Fujitsu VP-200.

How to Improve Vector Processing:

For improving the performance of vector processing, we need to reduce the overhead on the vector processor. In the following way we can ensure better efficiency of vector processor:

- **Efficient algorithm**: We need to choose an efficient algorithm that would work faster for vector pipelined processing.
- **Integrating scalar instruction**: We can reduce the overhead of reconfiguring pipelines by integrating scalar instruction of the same type.
- **Improving vector instruction**: Vector instruction can be improved by reducing memory access and utilizing resources.

Pipelining:

The term Pipelining refers to a technique of decomposing a sequential process into sub-operations, with each sub-operation being executed in a dedicated segment that operates concurrently with all other segments.

The most important characteristic of a pipeline technique is that several computations can be in progress in distinct segments at the same time. The overlapping of computation 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.

The structure of a pipeline organization can be represented simply by including an input register for each segment followed by a combinational circuit.

Let us consider an example of combined multiplication and addition operation to get a better understanding of the pipeline organization.

The combined multiplication and addition operation is done with a stream of numbers such as:

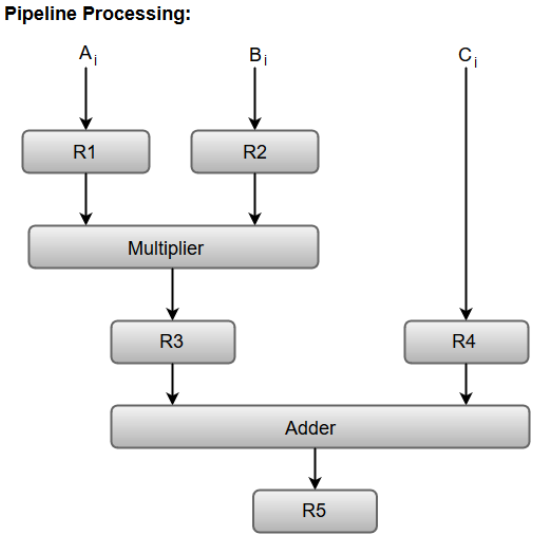
$A_i * B_i + C_i$ for i = 1, 2, 3, ..., 7

The operation to be performed on the numbers is decomposed into sub-operations with each sub-operation to be implemented in a segment within a pipeline.

The sub-operations performed in each segment of the pipeline are defined as:

$R1 \leftarrow A_i$, $R2 \leftarrow B_i$	Input A_i, and B_i
$R3 \leftarrow R1 * R2$, $R4 \leftarrow C_i$	Multiply, and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

The following block diagram represents the combined as well as the sub-operations performed in each segment of the pipeline.



Registers R1, R2, R3, and R4 hold the data and the combinational circuits operate in a particular segment.

The output generated by the combinational circuit in a given segment is applied as an input register of the next segment. For instance, from the block diagram, we can see that the register R3 is used as one of the input registers for the combinational adder circuit.

Pipeline Stages:

RISC processor has 5 stage instruction pipeline to execute all the instructions in the RISC instruction set. Following are the 5 stages of the RISC pipeline with their respective operations:

- **Stage 1 (Instruction Fetch):** In this stage the CPU reads instructions from the address in the memory whose value is present in the program counter.
- **Stage 2 (Instruction Decode):** In this stage, instruction is decoded and the register file is accessed to get the values from the registers used in the instruction.
- **Stage 3 (Instruction Execute):** In this stage, ALU operations are performed.
- **Stage 4 (Memory Access):** In this stage, memory operands are read and written from/to the memory that is present in the instruction.
- **Stage 5 (Write Back):** In this stage, computed/fetched value is written back to the register present in the instructions.

The pipelining concept assumes that different stages of the pipeline can operate concurrently. Once a stage completes its work on an instruction, it passes the instruction to the next stage, and the freed-up stage can start working on the next instruction in the pipeline.

Without pipelining, one instruction will get processed at a time, but we can process multiple instructions simultaneously with pipelining. That means while one instruction is decoded, the next instruction will be fetched simultaneously, and so on.

However, this isn't as simple as that. Some instructions may depend on other instructions. For example, instruction A updates a register, and instruction B uses the value stored in that register. So, we can't directly decode instruction B until instruction A has completed written back. There are several ways to handle such scenarios like stalling, data forwarding, etc.

Pipeline Design:

1. The entire process is divided into several stages.
2. We use buffers to hold intermediate output between two stages.
3. The output of one stage is connected to the next stage's input.
4. A common clock controls all the stages.

Pipelined Execution:

Consider that there are two processes - P1, P2, and there are 4 stages of execution - S1, S2, S3, and S4. The execution without pipelining will look like:

Cycle ->	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
S1	P1				P2			
S2		P1				P2		
S3			P1				P2	
S4				P1				P2

The execution with pipelining will look like:

Cycle ->	**1**	**2**	**3**	**4**	**5**
S1	P1	P2			
S2		P1	P2		
S3			P1	P2	
S4				P1	P2

We can see that without pipelining, it took 8 clock cycles to execute 3 instructions, while in pipelined execution, it took just 5 clock cycles.

Performance of a pipelined processor:

Consider a 'k' segment pipeline with clock cycle time as 'Tp'. Let there be 'n' tasks to be completed in the pipelined processor. Now, the first instruction is going to take 'k' cycles to come out of the pipeline but the other 'n – 1' instructions will take only '1'

cycle each, i.e, a total of ‘n – 1’ cycles. So, time taken to execute ‘n’ instructions in a pipelined processor:

$ET_{pipeline}$ = k + n – 1 cycles = (k + n – 1) Tp

In the same case, for a non-pipelined processor, the execution time of ‘n’ instructions will be:

$ET_{non\text{-}pipeline}$ = n * k * Tp

So, speedup (S) of the pipelined processor over the non-pipelined processor, when ‘n’ tasks are executed on the same processor is:

S = Performance of non-pipelined processor / Performance of pipelined processor

As the performance of a processor is inversely proportional to the execution time, we have,

S = ETnon-pipeline / ETpipeline

=> S = [n * k * Tp] / [(k + n – 1) * Tp]

S = [n * k] / [k + n – 1]

When the number of tasks ‘n’ is significantly larger than k, that is, n >> k

S = n * k / n

S = k

where ‘k’ are the number of stages in the pipeline.

Also, **Efficiency** = Given speed up / Max speed up = S / S_{max}

We know that S_{max} = k

So, **Efficiency** = S / k

Throughput = Number of instructions / Total time to complete the instructions

So, **Throughput** = **n / (k + n – 1) * Tp**

Performance of pipeline is measured using two main metrices as Throughput and latency.

Throughput:

- It measure number of instruction completed per unit time.
- It represents overall processing speed of pipeline.

- Higher throughput indicate processing speed of pipeline.
- Calculated as, throughput= number of instruction executed/ execution time.
- It can be affected by pipeline length, clock frequency. efficiency of instruction execution and presence of pipeline hazards or stalls.

Latency:

- It measure time taken for a single instruction to complete its execution.
- It represents delay or time it takes for an instruction to pass through pipeline stages.
- Lower latency indicates better performance .
- It is calculated as, Latency= Execution time/ Number of instruction executed.
- It in influenced by pipeline length, depth, clock cycle time, instruction dependencies and pipeline hazards.

Pipeline Hazards:

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.

Types of Pipeline Hazards in Computer Architecture

The three different types of hazards in computer architecture are:

1. Structural

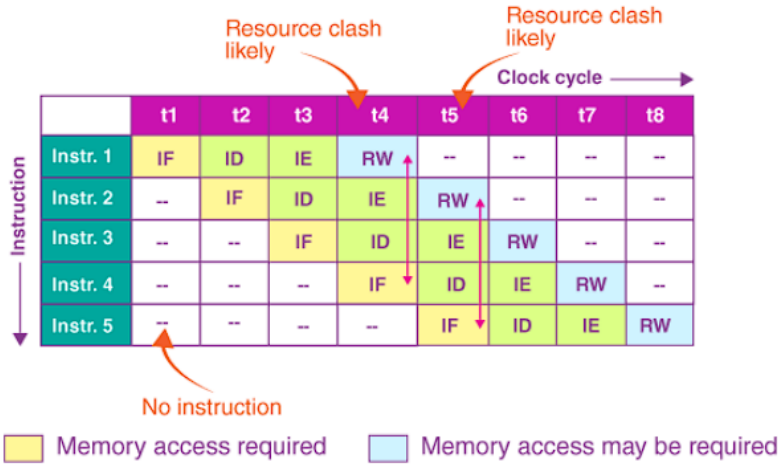
2. Data

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.

1. Structural Hazards:

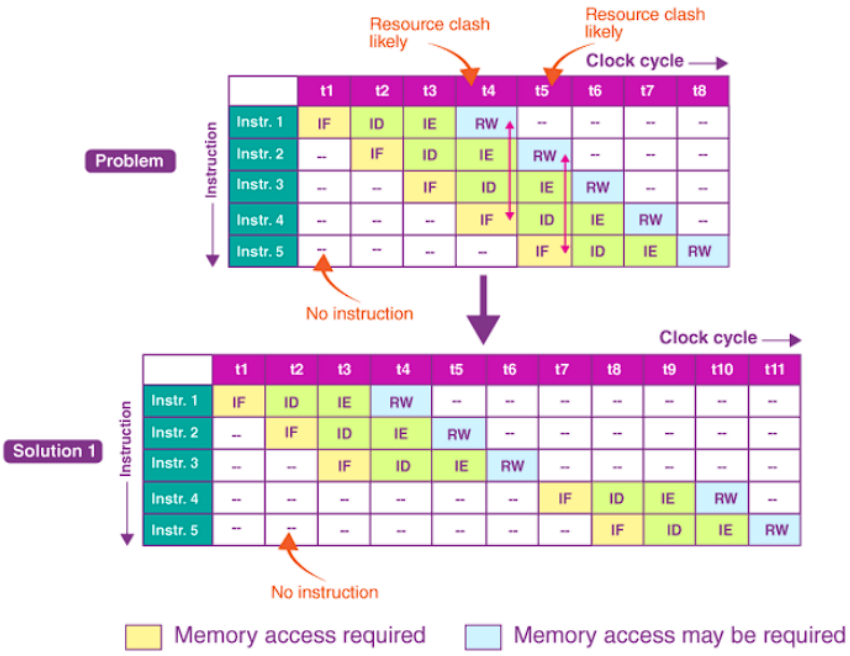
Hardware resource conflicts among the instructions in the pipeline cause structural hazards. Memory, a GPR Register, or an ALU might all be used as resources here. When more than one instruction in the pipe requires access to the very same resource in the same clock cycle, a resource conflict is said to arise. In an overlapping pipelined 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?

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.

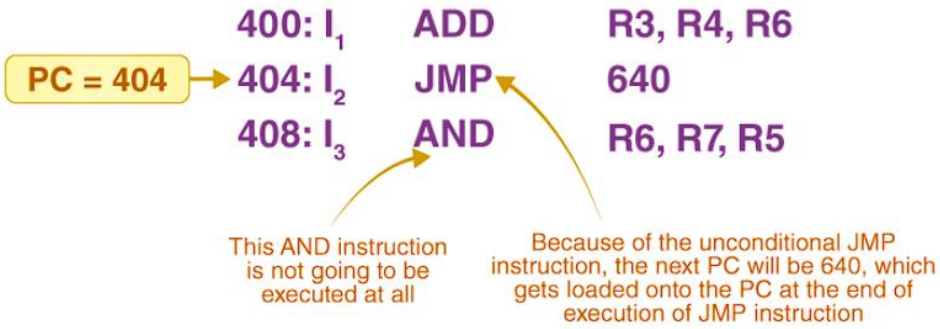
In recent CPUs, the last two approaches have been implemented. If dependency develops beyond this, the only choice is to stall. Keep in mind that acquiring more resources comes at a higher price. As a result, the trade-off is a designer's decision.

2. Control Hazards:

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.

Conditional Hazards Solutions:

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.

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.

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.

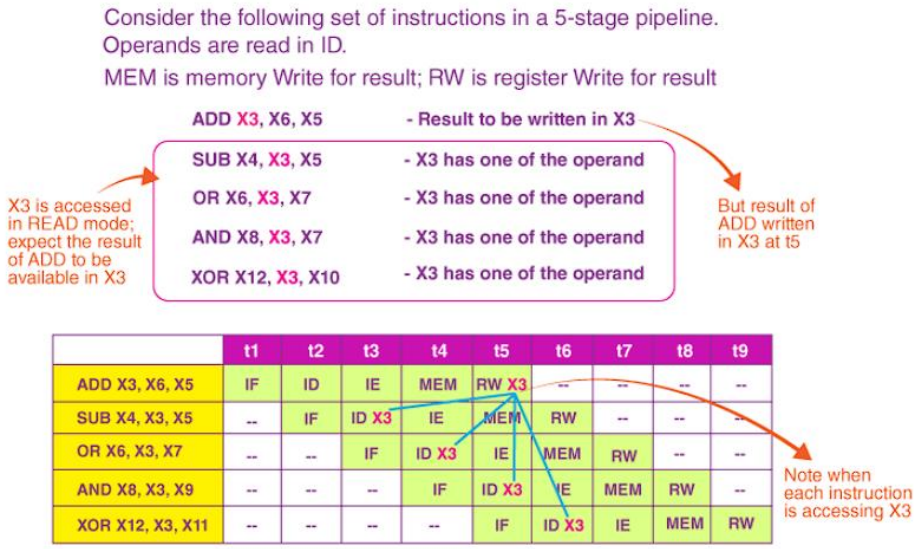
4. Reordering Instructions: Delayed branching entails reordering the instructions to move the branch instruction later in the sequence, allowing safe and beneficial instructions that are unaffected by the result of a branch to be brought in earlier in the sequence, delaying the fetch of the branch instruction. 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

3. Data Hazards:

When the execution of an instruction is dependent on the results of a prior instruction that's still being processed in a pipeline, data hazards occur. Consider the following scenario.



	t1	t2	t3	t4	t5	t6	t7	t8	t9
ADD X3, X6, X5	IF	ID	IE	MEM	RW X3	--	--	--	--
SUB X4, X3, X5	--	IF	ID X3	IE	MEM	RW	--	--	--
OR X6, X3, X7	--	--	IF	ID X3	IE	MEM	RW	--	--
AND X8, X3, X9	--	--	--	IF	ID X3	IE	MEM	RW	--
XOR X12, X3, X11	--	--	--	--	IF	ID X3	IE	MEM	RW

The result of the ADD instruction is written into the register X3 at t5 in the example above. If bubbles aren't used to postpone the following SUB instruction, all the three operations will use data from X3, that is earlier in the ADD process. The program has gone haywire!

Solutions to Data Hazard:

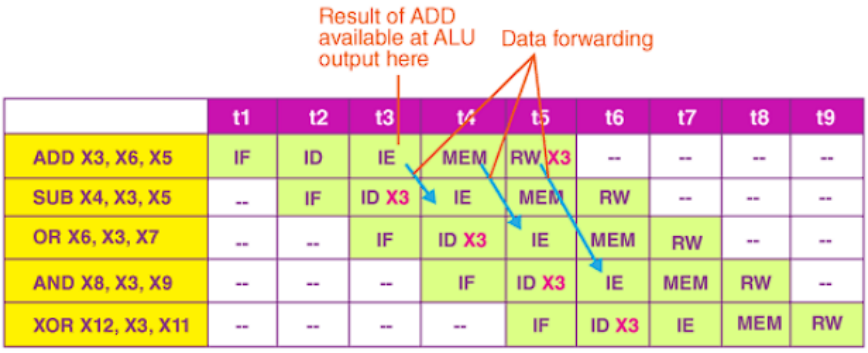
The following are some of the probable solutions:

Solution 1: At the IF stage of the SUB instruction, add three bubbles. This will make it easier for SUB – ID to work at t6. As a result, all subsequent instructions in the pipe are similarly delayed.

Solution 2:

Forwarding of Data – Data forwarding is the process of sending a result straight to that functional unit which needs it: a result is transferred from one unit's output to another's input. The goal is to have the solution ready for the next instruction as soon as possible.

In this scenario, the ADD result can be found at the ALU output in ADD –IE, that is the t3 end. In case the control unit can control and forward this to the SUB-IE stage at t4 just before writing to the output register X3, the pipeline will proceed without halting. This necessitates additional processing to detect and respond to this data hazard. It's worth noting that, though Operand Fetch normally occurs in the ID stage, it's only utilised in the IE stage. As a result, the IE stage receives forwarding as an input. OR and AND instructions can also be used to forward data in a similar way.



	t1	t2	t3	t4	t5	t6	t7	t8	t9
ADD X3, X6, X5	IF	ID	IE	MEM	RW X3	--	--	--	--
SUB X4, X3, X5	--	IF	ID X3	IE	MEM	RW	--	--	--
OR X6, X3, X7	--	--	IF	ID X3	IE	MEM	RW	--	--
AND X8, X3, X9	--	--	--	IF	ID X3	IE	MEM	RW	--
XOR X12, X3, X11	--	--	--	--	IF	ID X3	IE	MEM	RW

Solution 3:

When generating executable code, the compiler can recognise data dependencies and reorganise (resequence) the instructions appropriately. This will make the device easier to use.

Solution 4:

If the reordering described above is not possible, the compiler can detect and insert a no operation (or NOP) instruction(s). NOP refers to a software-generated dummy instruction equivalent bubble.

During the code optimization stage of the compilation process, the compiler examines data dependencies.

Classification of Data Hazards:

Data hazards are divided into three types according to the order in which READ or WRITE operations are performed on the register:

Flow/True Data Dependency [RAW (or Read after Write)]: This is when one instruction makes use of data from a previous instruction.

Example,

ADD X0, X1, X2

SUB X4, X3, X0

Anti-Data Dependency [WAR (or Write after Read)]: When the second instruction is written to a register before the first instruction is read, this is known as a race condition. In case of a simple structure of a pipeline, this is uncommon. WAR, on the other hand, can occur in some machines having complex and specific instructions.

Example,

ADD X2, X1, X0

SUB X0, X3, X4

Output data dependency [WAW (or Write after Write)]: This is a situation where two simultaneous instructions must write the same register in the same sequence they were issued.

Example,

ADD X0, X1, X2

SUB X0, X4, X5

Only when instructions are parallelly executed or out of sequence may WAW and WAR dangers exist. These arise because the compiler has allotted the very same register numbers, which may have been avoided. This problem can be solved by the compiler renaming one of these registers or waiting for the updating of a given register until the proper value has been generated. Modern CPUs include not only parallel execution with various ALUs, they also include out-of-order instruction issuing and execution, as well as many pipeline stages.

Types of Pipeline:

It is divided into 2 categories:

1. Arithmetic Pipeline
2. Instruction Pipeline

1. Arithmetic Pipeline:

Arithmetic Pipelines are mostly used in high-speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems.

To understand the concepts of arithmetic pipeline in a more convenient way, let us consider an example of a pipeline unit for floating-point addition and subtraction.

The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers defined as:

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

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

Where **A** and **B** are two fractions that represent the mantissa and **a** and **b** are the exponents.

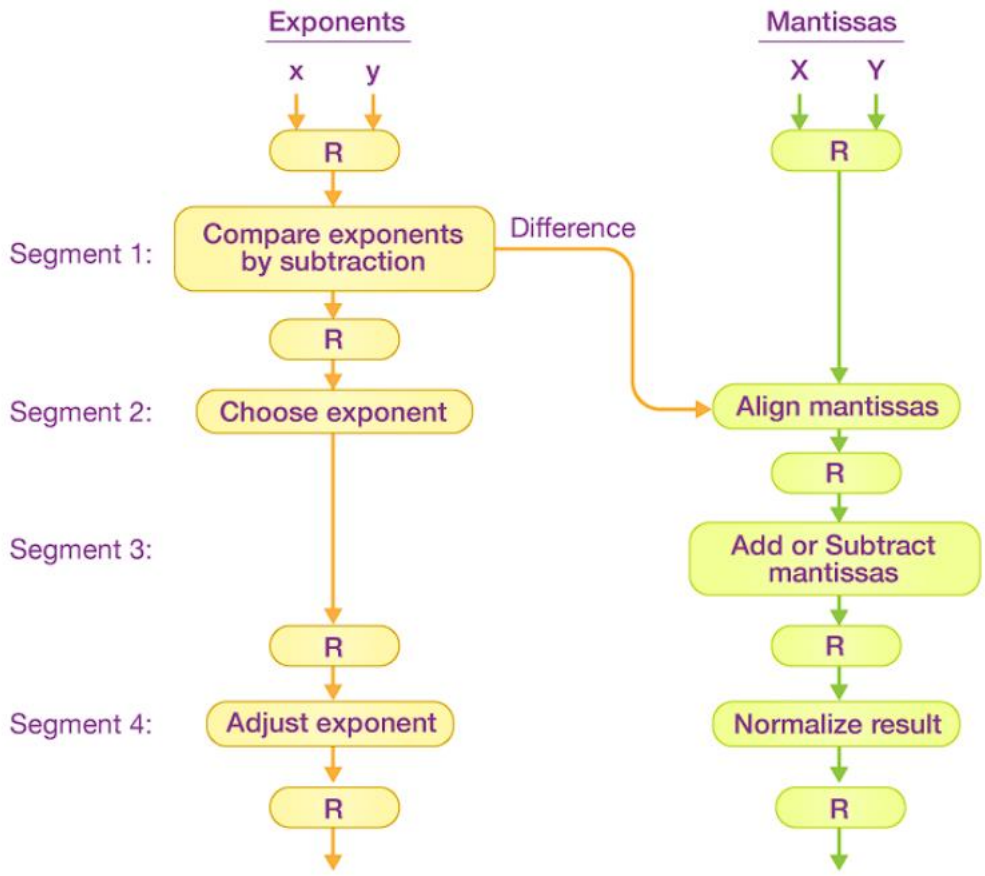
The combined operation of floating-point addition and subtraction is divided into four segments. Each segment contains the corresponding suboperation to be performed in the given pipeline. The suboperations that are shown in the four segments are:

1. Compare the exponents by subtraction.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

We will discuss each suboperation in a more detailed manner later in this section.

The following block diagram represents the suboperations performed in each segment of the pipeline.

Pipeline organization for floating point addition and subtraction:



Note: Registers are placed after each suboperation to store the intermediate results.

1. Compare exponents by subtraction:

The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result.

The difference of the exponents, i.e., **3 - 2 = 1** determines how many times the mantissa associated with the smaller exponent must be shifted to the right.

2. Align the mantissas:

The mantissa associated with the smaller exponent is shifted according to the difference of exponents determined in segment one.

$X = 0.9504 * 10^3$

$Y = 0.08200 * 10^3$

3. Add mantissas:

The two mantissas are added in segment three.

$Z = X + Y = 1.0324 * 10^3$

4. Normalize the result:

After normalization, the result is written as:

$Z = 0.1324 * 10^4$

2. Instruction Pipeline:

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.

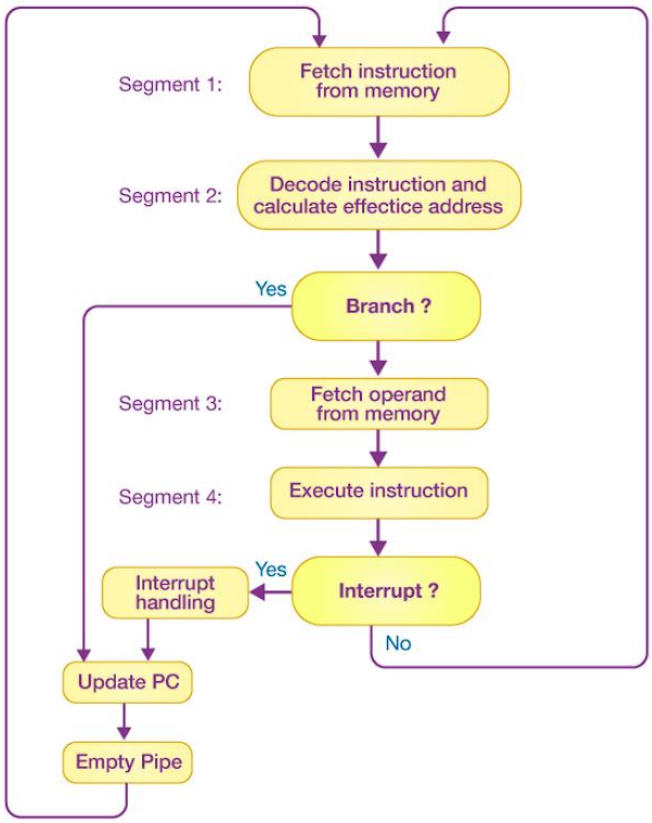
In general, each and every instruction must be processed by the computer in the following order:

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 organisation 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 organisation.

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 organisation.

PART – 2 : COMPUTER ARITHMETIC

Introduction:

Data is manipulated by using the arithmetic instructions in digital computers. Data is manipulated to produce results necessary to give solution for the computation problems. The Addition, subtraction, multiplication and division are the four basic arithmetic operations.

If we want then we can derive other operations by using these four operations. To execute arithmetic operations there is a separate section called arithmetic processing unit in central processing unit. The arithmetic instructions are performed generally on binary or decimal data. Fixed-point numbers are used to represent integers or fractions. We can have signed or unsigned negative numbers. Fixed-point addition is the simplest arithmetic operation.

If we want to solve a problem then we use a sequence of well-defined steps. These steps are collectively called algorithm. To solve various problems, we give algorithms.

In order to solve the computational problems, arithmetic instructions are used in digital computers that manipulate data. These instructions perform arithmetic calculations.

And these instructions perform a great activity in processing data in a digital computer. As we already stated that with the four basic arithmetic operations addition, subtraction, multiplication and division, it is possible to derive other arithmetic operations and solve scientific problems by means of numerical analysis methods.

A processor has an **arithmetic processor** (as a sub part of it) that executes arithmetic operations. The data type, assumed to reside in processor, registers during the execution of an arithmetic instruction. Negative numbers may be in a signed magnitude or signed complement representation. There are three ways of representing negative fixed point - binary numbers signed magnitude, signed 1's complement or signed 2's complement. Most computers use the signed magnitude representation for the mantissa.

Signed Magnitude Binary Numbers:

Signed magnitude is a method of representing both positive and negative numbers in binary form. In signed magnitude representation, the leftmost bit (most significant bit) is used to indicate the sign of the number. The remaining bits represent the magnitude (absolute value) of the number.

Representation:

- **Sign Bit:** The leftmost bit (MSB) is used to indicate the sign of the number.
  - 0 for positive numbers
  - 1 for negative numbers

- **Magnitude Bits:** The remaining bits represent the magnitude of the number in binary form.

Examples:

- For an 8-bit signed magnitude representation:
  - 0 0110101 represents the positive number 53
  - 1 0110101 represents the negative number -53

In the examples above, the leftmost bit (the sign bit) determines whether the number is positive or negative, and the remaining bits represent the magnitude of the number.

Addition and Subtraction:

With Signed –Magnitude Data:

We designate the magnitude of the two numbers by A and B. Where the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to present a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words parentheses should be used for the subtraction algorithm).

TABLE Addition and Subtraction of Signed-Magnitude Numbers

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

When the signs of A and B are same, add the two magnitudes and attach the sign of result is that of A. When the signs of A and B are not same, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A, if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result will be positive.

Hardware Implementation:

Need:

Two Register to hold magnitude of A (Accumulator), B and Result

Two Flip-flop to hold sign of As, Bs and Result

A parallel adder to perform micro-operation A+B

A comparator circuit to establish A>B, A=B and A<B

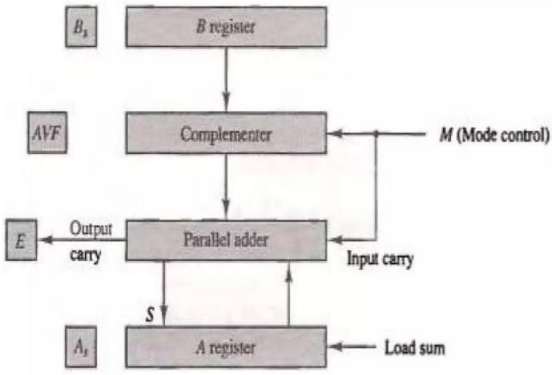
Two Parallel subtractor circuit to perform micro-operation A-B & B-A

An Exclusive OR gate to determine sign of result with input As, Bs.

AVF (Add over-flow flip flop) hold the overflow bit.

When M = 0: the output of B is transferred to the adder, the input carry is 0 and the output of the circuit is equal to A+B.

When M = 1: the 1's Complement of B is applied to adder, the input carry is 1 and output S = A+B+1, this is equal to A plus 2's complement of B



Hardware for signed-magnitude addition and subtraction.

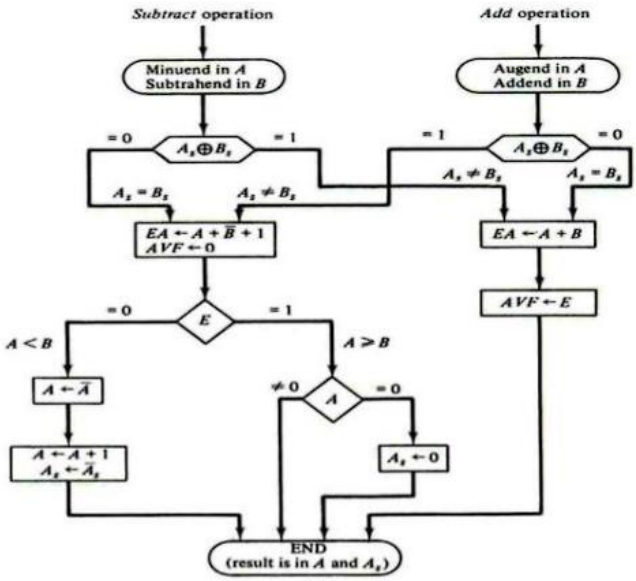


Figure: Flowchart

With Signed –2's Complement Data:

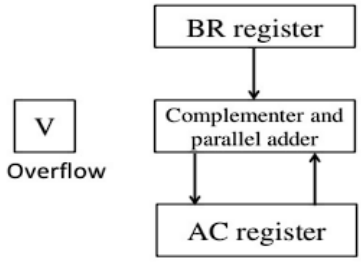


Figure: Hardware for signed-2's complement addition and subtraction.

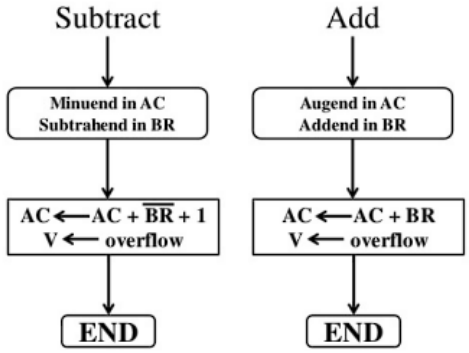


Figure: Algorithm for adding and subtracting numbers in signed-2's complement representation.

Multiplication:

Signed-Magnitude Complement:

If the multiplier bit is 1, the multiplicand is copied down, otherwise zeros are copied down. The number copied down in successive lines are shifted one position left from the previous number. Finally, the numbers are added. If the sign is alike take positive sign otherwise take negative sign.

```
23     10111   Multiplicand
19   x 10011   Multiplier
       10111
      10111
     00000   +
    00000
   10111
437 110110101  Product
```

1. Instead of providing register to store and add simultaneously as many binary number as there are bits in multiplier, it is convent to provide an adder for the summation of only two binary number and successively accumulate the partial product in a register.
2. Instead of shifting the multiplicand to the left, the partial product is shifted to the right, which result in leaving the partial product and the multiplicand in the required relative position.
3. When the corresponding bit of the multiplier is zero, there are no need to add all zeros to partial product since it will not alter its value.

Hardware Implementation:

The hardware for multiplication consists of the equipment given in below Figure. The multiplier is stored in the register and its sign in Qs. The sequence counter SC is initially set bits in the multiplier. When the content of the counter reaches zero, the product is complete and we stop the process.

Figure Hardware for multiply operation.

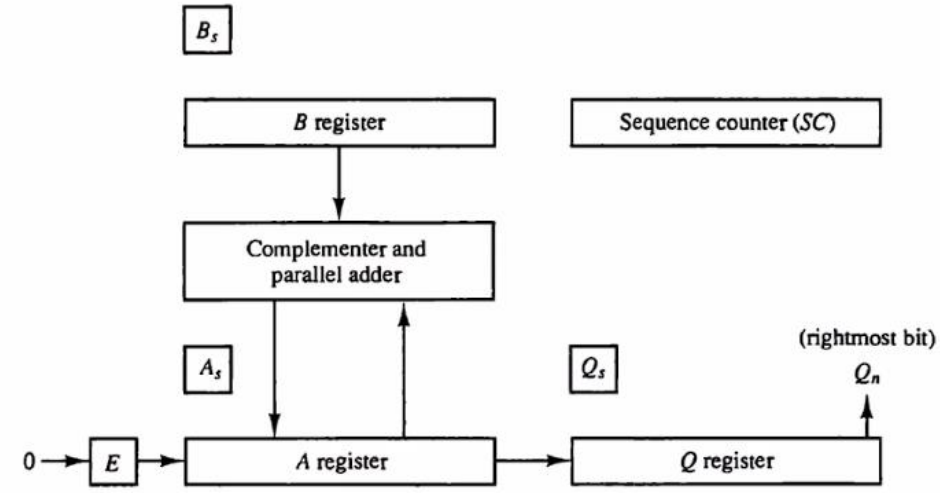


Figure Flowchart for multiply operation.

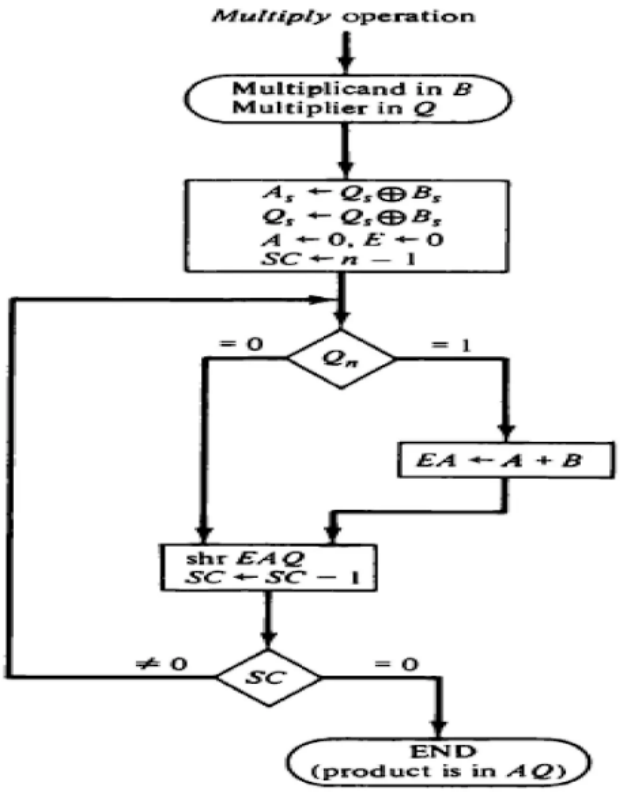


TABLE Numerical Example for Binary Multiplier

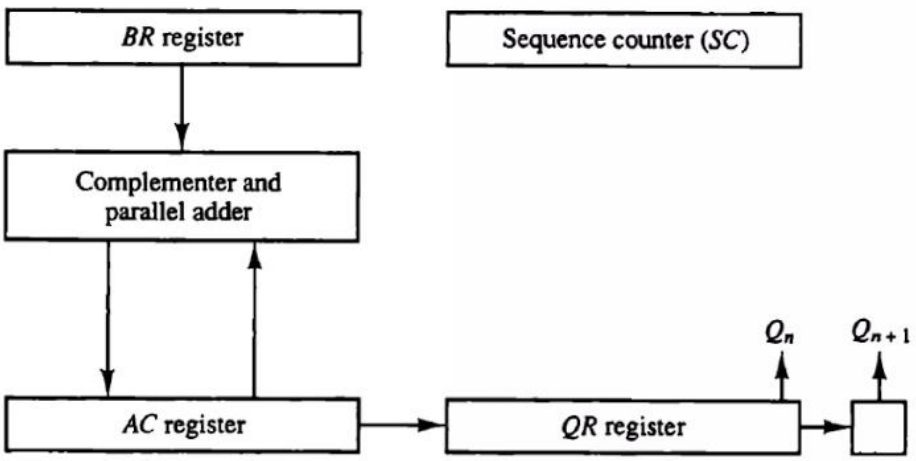
Multiplicand $B = 10111$	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in $AQ = 0110110101$				

Booth's Multiplication Algorithm (Signed-2's Complement):

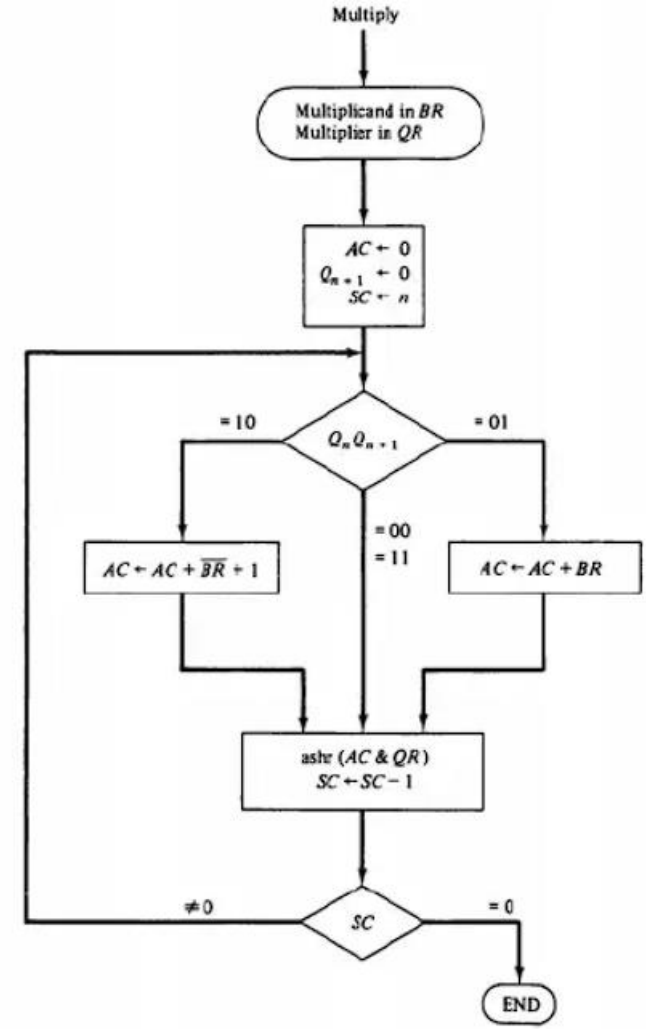
The booth algorithm is a multiplication algorithm that allows us to multiply the two signed binary integers in 2's complement, respectively. It is also used to speed up the performance of the multiplication process. It is very efficient too. It works on the string bits 0's in the multiplier that requires no additional bit only shift the right-most string bits and a string of 1's in a multiplier bit weight 2^k to weight 2^m that can be considered as $\mathbf{2^{k+1} - 2^m}$.

Hardware Implementation of Booths Algorithm – The hardware implementation of the booth algorithm requires the register configuration shown in the figure below.

Figure Hardware for Booth algorithm.



Flowchart: The Flowchart for Signed – 2's Complement as shown below –



The number can be represented as $2^{k+1} – 2^m = 2^4 - 2^0 = 16 - 1 = 15$. Therefore, the multiplication M x 14, where M is the multiplicand and 14 the multiplier may be computed as M x 2^4 - M x 2^1. That is, the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

Booth algorithm needs examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand added to the partial product, subtracted from the partial product, or left unchanged by the following rules:

1. The multiplicand is subtracted from the partial product when we get the first least significant 1 in a string of 1's in the multiplier.

2. The multiplicand is added to the partial product when we get the first Q (provided that there was a previous 1) in a string of 0's in the multiplier.

3. The partial product does not change when the multiplier bit is the same as the previous multiplier bit.

The algorithm applies to both positive and negative multipliers in 2's complement representation. This is because a negative multiplier ends with a string of l's and the last operation will be a subtraction of the appropriate weight. For example, a multiplier equal to -14 is represented in 2's complement as 110010 and is treated as $-2^4 + 2^2 - 2^1 = -14$.

A numerical example of Booth algorithm is given in Table for n = 5. It gives the multiplication of (- 9) x (-13) = +117.

TABLE Example of Multiplication with Booth Algorithm

$Q_n Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

Application of Booth's Algorithm:

1. Chip and computer processors: Corner's Calculation is utilized in the equipment execution of number-crunching rationale units (ALUs) inside microchips and computer chips. These parts are liable for performing number juggling and coherent procedure on twofold information. Proficient duplication is fundamental in different applications, including logical registering, designs handling, and cryptography. Corner's Calculation lessens the quantity of piece movements and augmentations expected to perform duplication, bringing about quicker execution and better in general execution.

2. Digital Signal Processing (DSP): DSP applications frequently include complex numerical tasks, for example, sifting and convolution. Duplicating enormous twofold numbers is a principal activity in these errands. Corner's Calculation permits DSP frameworks to perform duplications all the more productively, empowering ongoing handling of sound, video, and different sorts of signs.

3. Hardware Accelerators: Many particular equipment gas pedals are intended to perform explicit assignments more productively than broadly useful processors. Corner's Calculation can be integrated into these gas pedals to accelerate augmentation activities in applications like picture handling, brain organizations, and AI.

4. Cryptography: Cryptographic calculations, like those utilized in encryption and computerized marks, frequently include particular exponentiation, which requires proficient duplication of huge numbers. Corner's Calculation can be utilized to speed up the measured augmentation step in these calculations, working on the general proficiency of cryptographic tasks.

5. High-Performance Computing (HPC): In logical reenactments and mathematical calculations, enormous scope augmentations are oftentimes experienced. Corner's Calculation can be carried out in equipment or programming to advance these duplication tasks and improve the general exhibition of HPC frameworks.

6. Implanted Frameworks: Inserted frameworks frequently have restricted assets regarding handling power and memory. By utilizing Corner's Calculation, fashioners can upgrade augmentation activities in these frameworks, permitting them to perform all the more proficiently while consuming less energy.

7. Network Parcel Handling: Organization gadgets and switches frequently need to perform estimations on bundle headers and payloads. Augmentation activities are regularly utilized in these estimations, and Corner's Calculation can assist with diminishing handling investment utilization in these gadgets.

8. Advanced Channels and Balancers: Computerized channels and adjusters in applications like sound handling and correspondence frameworks require productive augmentation of coefficients with input tests. Stall's Calculation can be utilized to speed up these increases, prompting quicker and more precise sifting activities.

Array Multiplier:

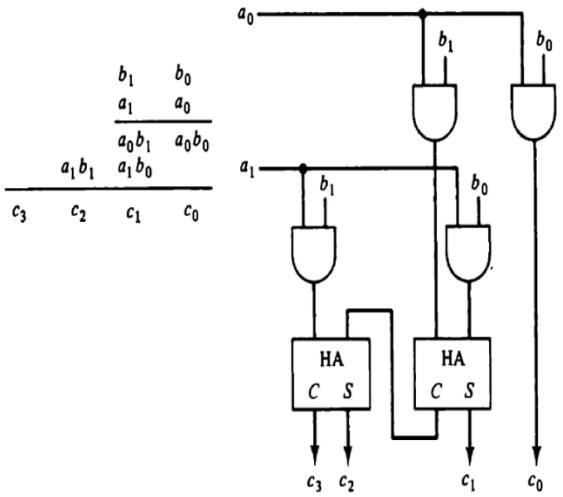
To check the bits of the multiplier one at a time and forming partial products is a sequential operation requiring a sequence of add and shift micro-operations. The multiplication of two binary numbers can be done with one micro-operation by using combinational circuit that forms the product bits all at once.

This is a fast way since all it takes is the time for the signals to propagate through the gates that form the multiplication array. However, an array multiplier requires a large number of gates, and so it is not an economical unit for the development of ICs.

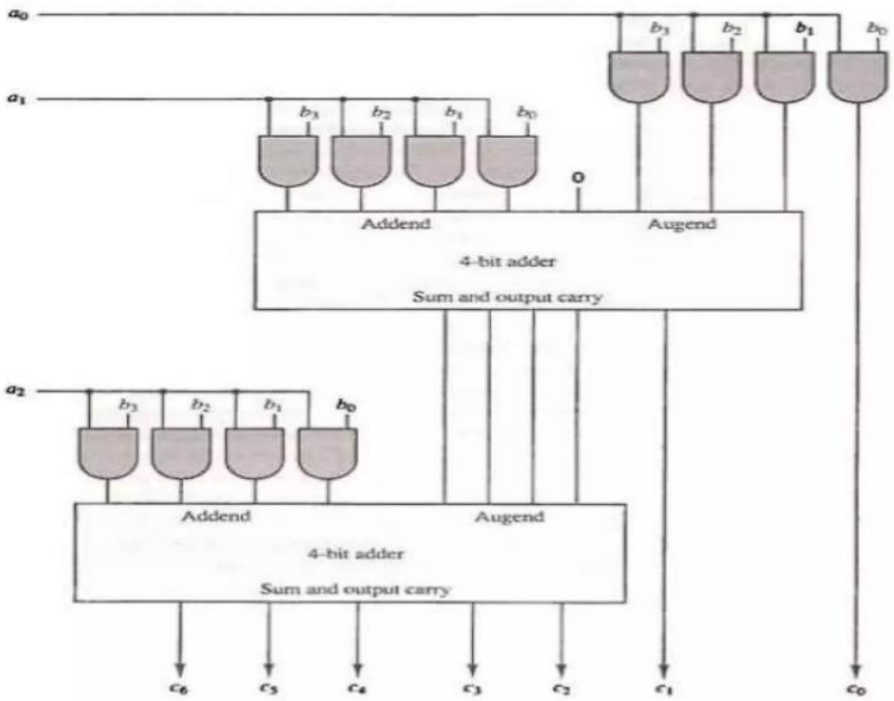
Now we see how an array multiplier is implemented with a combinational circuit. Consider the multiplication of two 2-bit numbers as shown in Fig. The multiplicand bits are b1 and b0, the multiplier bits are a1 and a0, and the product is c3 c2 c1 c0. The first partial product is obtained by multiplying a0 by b1b0. The multiplication of two bits gives a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation and can we implement it with an AND gate. As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a1 by b1b0 and is shifted one position to the left. The two

partial products are added with two half-adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full-adders to produce the sum. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.

2-bit by 2-bit array multiplier.



As a second example, consider a multiplier circuit that multiplies a binary number of four bits with a number of three bits. Let the multiplicand be represented by b3b2b1b0 and the multiplier by a2a1a0. Since k=4 and j=3, we need 12 AND gates and two 4-bit adders to produce a product of seven bits. The logic diagram of the multiplier is shown in Figure.



4-bit by 3-bit array multiplier.

For j Multiplier bits and k multiplicand bit we need j*k AND Gate and Total (j-1)k-bit adders to produced a product of j+k bits.

Division Algorithm:

Division of two fixed-point binary number in signed-magnitude representation is done with paper and pencil by a process of successive Compare, shift and subtract operation.

It is simpler than decimal division because is the quotient digit are either 0 and 1and there is no need to estimate how many times the dividend or partial remainder fits into divisor.

The division process is described in Figure. The divisor B has five bits and the dividend A had ten bits.

Divisor:		
	11010	Quotient = Q
B = 10001	0111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geq B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geq B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q; shift right B
	---010100	Remainder $\geq B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder

The devisor is compared with the five most significant bits of the dividend. Since the 5-bit number is smaller than B, we again repeat the same process. Now the 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. Now we shift the divisor once to the right and subtract it from the dividend. The difference is known as a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. Comparing a partial remainder with the divisor continues the process. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Obviously, the result gives both a quotient and a remainder.

Hardware Implementation for Signed-Magnitude Data:

• Instead of shifting the divisor to the right, the dividend, or partial remainder is shifted to the left.

• Subtraction may be achieved by adding to the 2's complement of B.

• H/w need is identical to h/w required for multiplication.

• Sign are alike, the sign of Quotient is plus.

• Sign are unalike, the sign is minus.

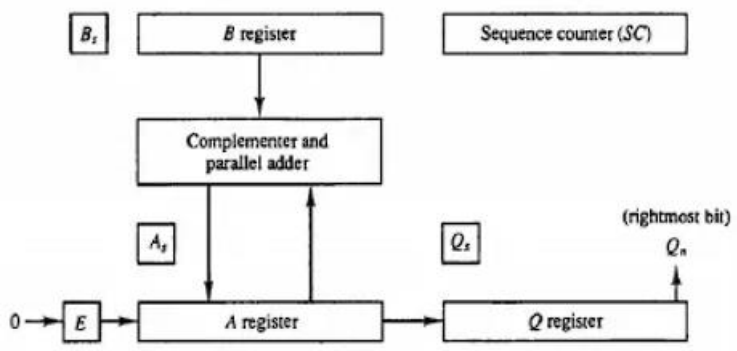
• The sign of remainder is the same as the sign of the dividend.

• **If E=1, it signifies that A>=B**

- A quotient bit 1 is inserted into Q_n
- The partial remainder is shifted to the left to repeat the process
- Divisor is subtracted by adding its 2's complements.

• **If E=0, it signifies that A<B**

- Quotient in Q_n remains a 0
- The value of A is added to restore the partial remainder in A.
- Shift EAQ and add 2's complement



Divisor B = 10001, $\overline{B} + 1$ = 01111

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Figure Example of binary division with digital hardware.

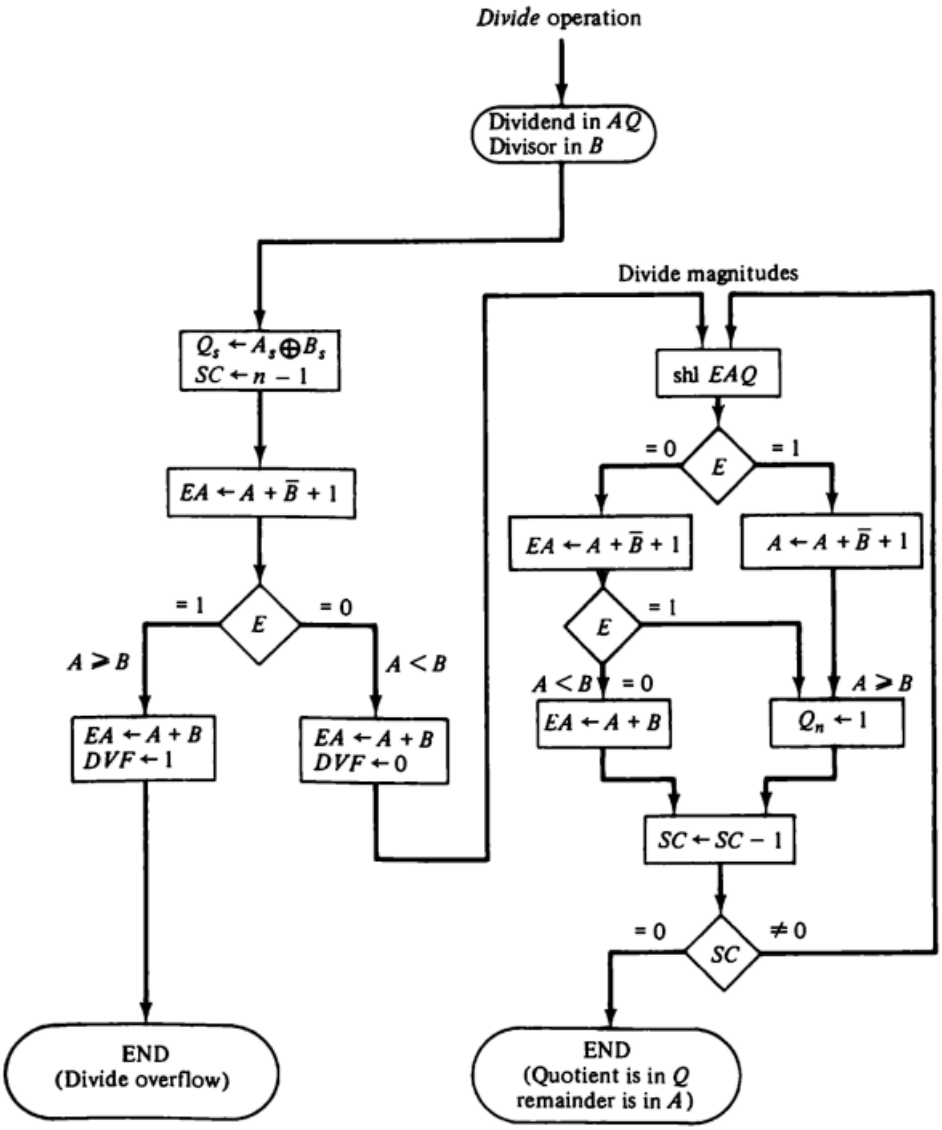
Divide Overflow:

• Dividend is twice bit long then divisor.

• The condition of overflow can be detected when the upper half of the dividend is greater or equal to the divisor.

• Another problem is division by o.

• In case of overflow DVF (divide overflow flag) is used to indicate the condition.



Other division algorithms

Method described above is **restoring method** in which *partial remainder* is restored by adding the divisor to the negative result. Other methods:

Comparison method: A and B are compared prior to subtraction. Then if A>=B, B is subtracted form A. if A<B nothing is done. The partial remainder is then shifted left and numbers are compared again. Comparison inspects end-carry out of the parallel adder before transferring to E.

Nonrestoring method: In contrast to restoring method, when A-B is negative, B is not added to restore A but instead, negative difference is shifted left and then B is added. How is it possible? Let's argue:

- In flowchart for restoring method, when A<B, we restore A by operation A-B+B. Next tine in a loop, this number is shifted left (multiplied by 2) and B subtracted again, which gives: 2(A - B + B) – B = **2A-B.**
- In Nonrestoring method, we leave A-B as it is. Next time around the loop, the number is shifted left and B is added: 2(A-B)+B = **2A-B** (same as above).