Module - 6

Flynn's Classification:

Flynn proposed a classification for the organisation of a computer system by the number of instructions & data items that are manipulated simultaneously.

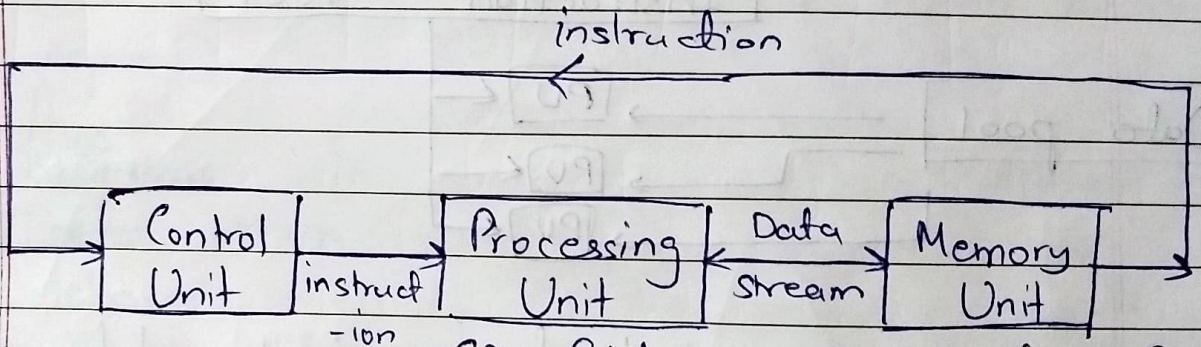
The sequence of instructions read from memory constitutes an instruction stream.

The operations performed on the data in the processor constitute a data stream.

Parallel processing: Many calculations & processes are carried out simultaneously. Large problems can be divided into smaller ones, which can be solved simultaneously.

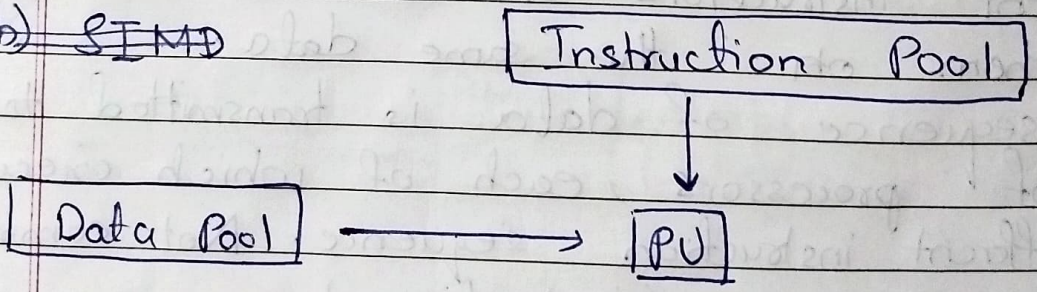
Basically, the classifications are based on the number of the no. of instruction & data streams that can be processed simultaneously:

a) SISD (Single instruction, single data stream):
Uniprocessor machine which is capable of executing a single instruction, operating on a single data stream. Machine instructions are processed in sequential manner. All the instructions & data to be processed are stored in primary memory. Most conventional computers (4th gen) use SISD eg: IBM 701, IBM 1620. No parallelism in SISD.

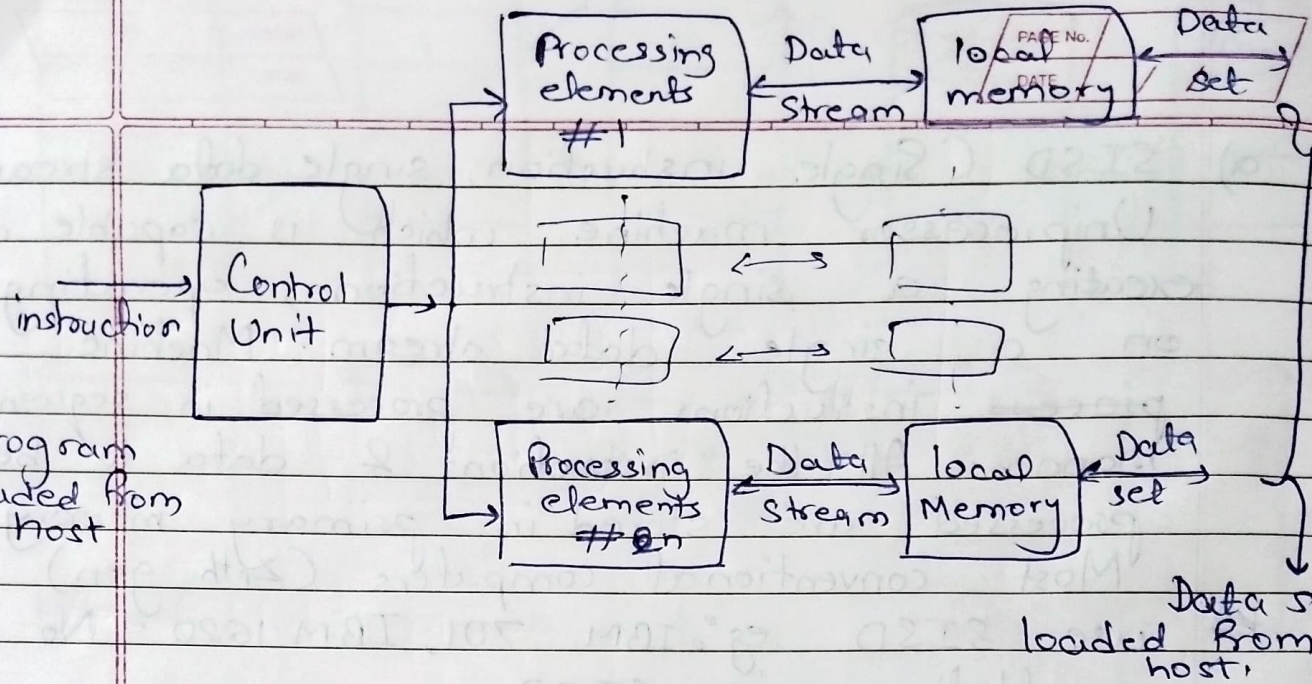


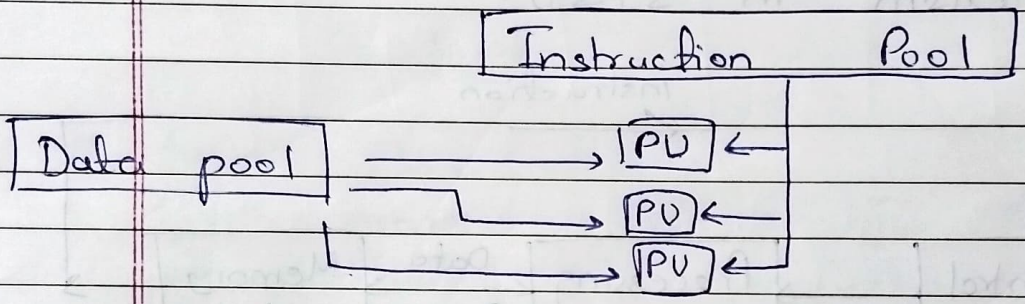
CPU fetches an instruction & its executed using single Data Stream

~~b) SIMD~~

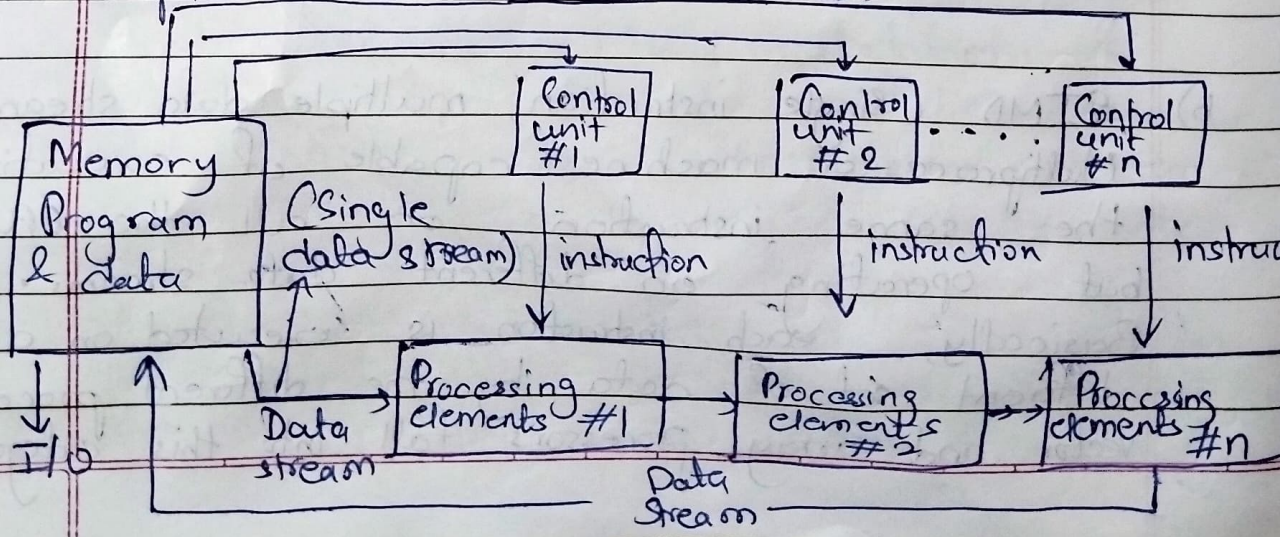


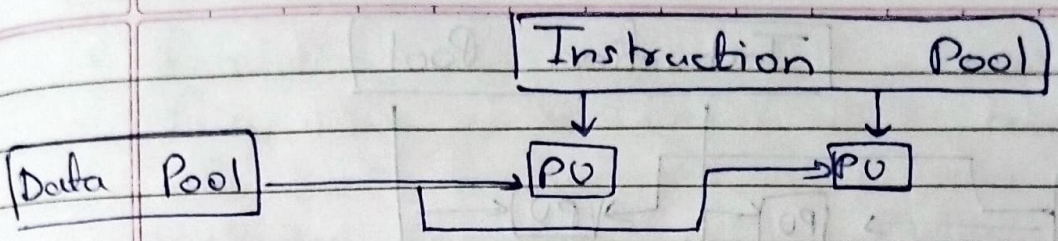
b) SIMD (Single instruction, multiple data stream):
Multiprocessor machine capable of executing the same instruction on all the CPUs but operating on different data streams. Basically, each instruction is executed on a different set of data by the different processors. Vector and array processors fall into this category.





e) MISD (Multiple instruction, single data stream): Multiprocessor machine capable of executing different instructions ~~on different~~ but all of them on the same data.
A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. Not commercially implemented.

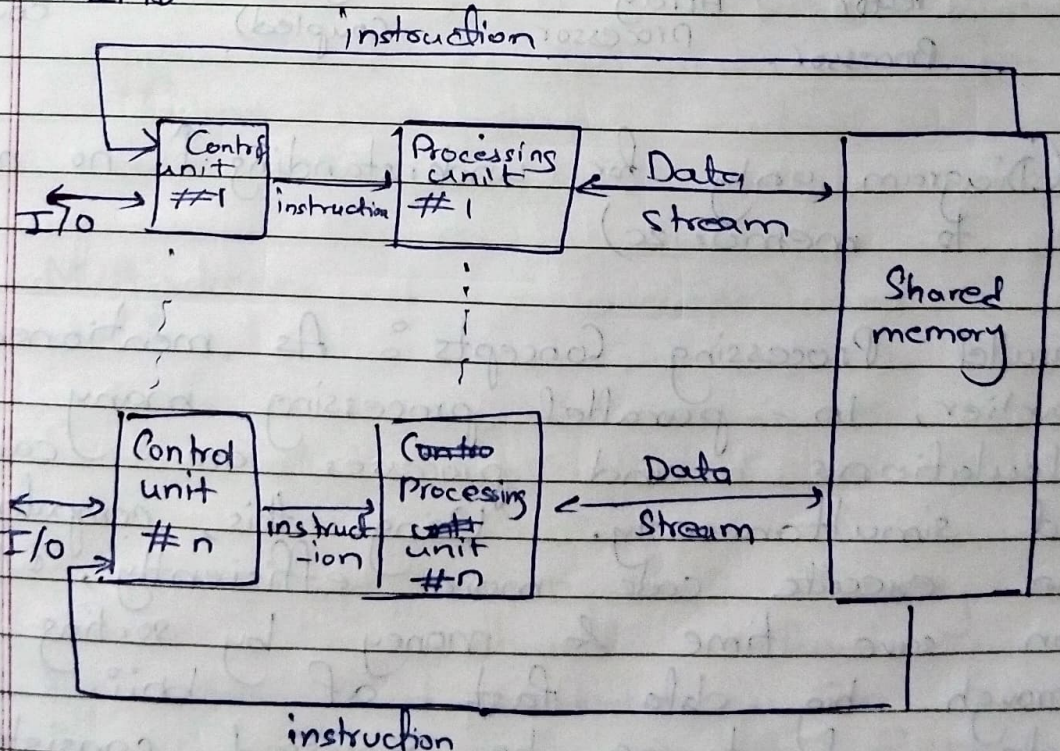


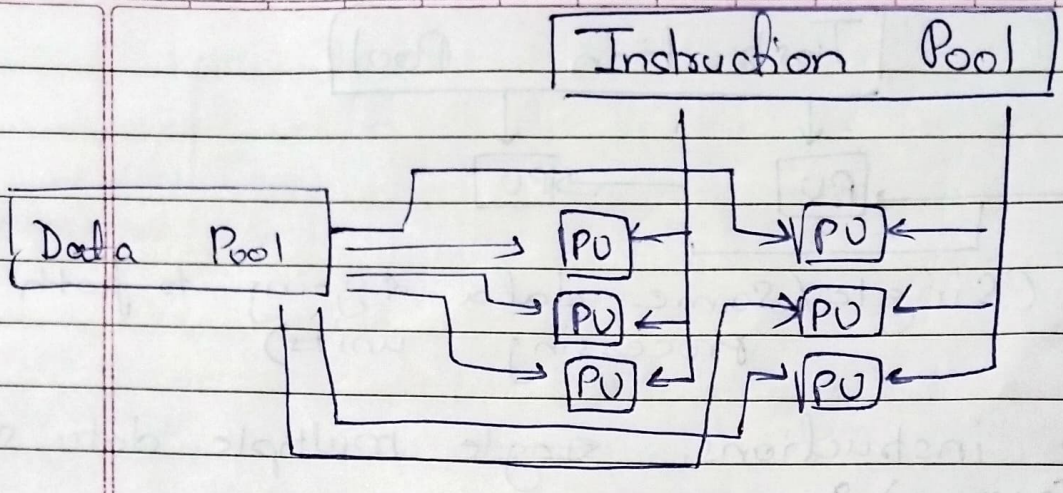


(Single/same data going to both processing units)

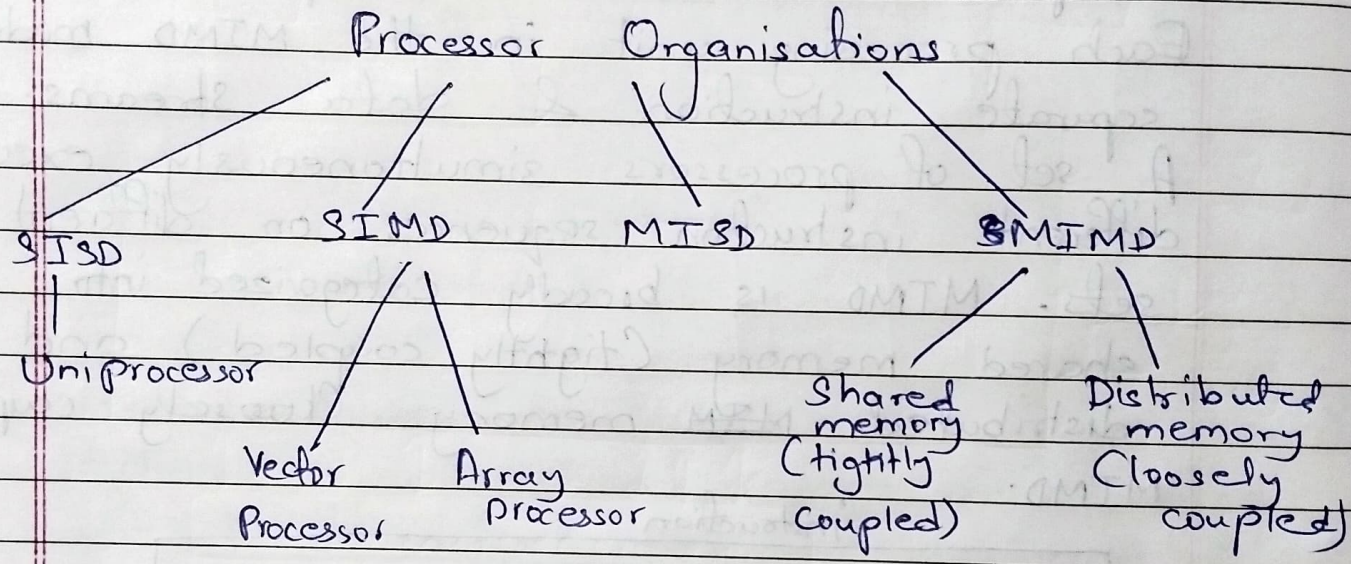
d) Multiple instructions, ~~single~~ multiple data stream (MIMD):

Multiprocessor machine capable of executing multiple instructions on multiple data sets. Each processing unit in the MIMD model has separate instruction & data streams. A set of processors simultaneously execute different instruction sequences on different data sets. MIMD is broadly categorized into shared memory (tightly coupled) and distributed ~~MEM~~ memory (loosely coupled) MIMD.





Parallel processing



(Diagram only for understanding ↑ no need to memorize)

Parallel Processing Concepts : As mentioned earlier, in parallel processing many calculations and processes are carried out simultaneously. Using this, computers can execute code more efficiently, which can save time & money by sorting through big data fast af boii.
Data needs to ~~be~~ be processed consistently to achieve better (fast) data throughput.

* data throughput: the amount of data passing through a process.

PAGE NO.
DATE / /

- Parallelism: techniques to make programs faster by performing several computations at the same time. This requires hardware with multiple processing units. Several ALUs are required to make these several computations possible. Multiple processing means several processors are operating concurrently.

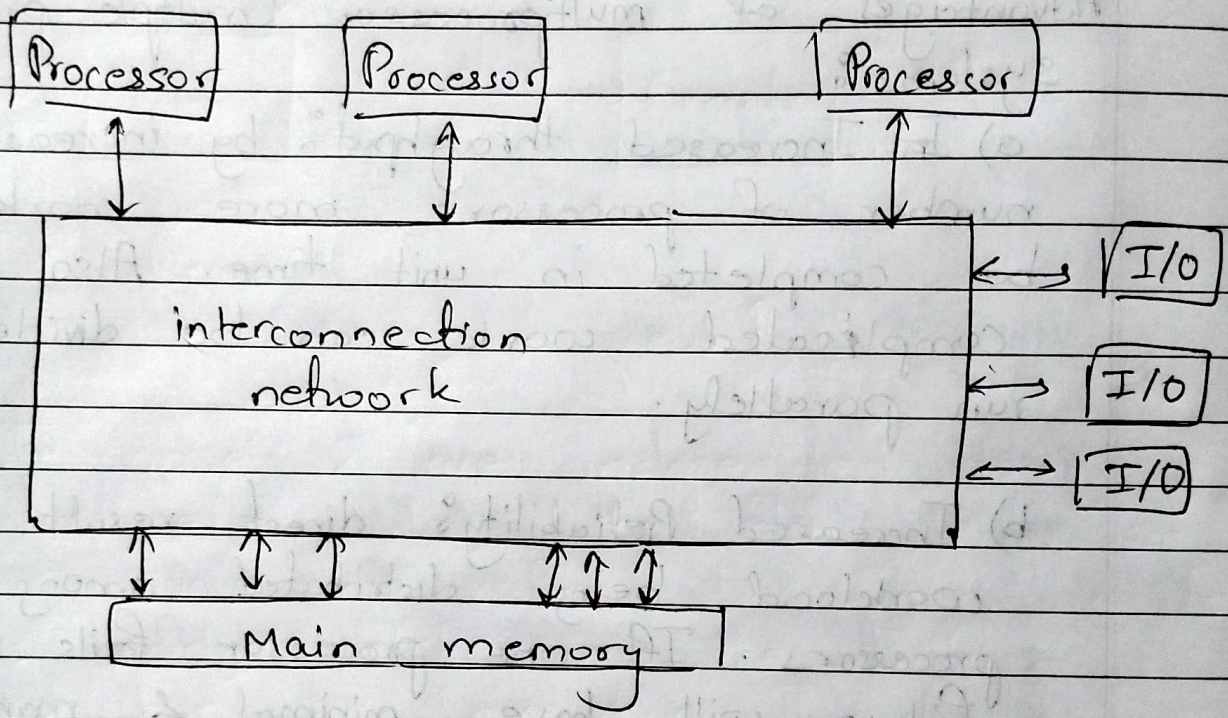
- Advantages of multiprocessor (multiple process) systems:
  a) Increased throughput: by increasing the number of processors, more work can be completed in unit time. Also, a complicated work can be divided & run parallely.

  b) Increased Reliability: direct result of workload being distributed among several processors. If one processor fails then its failure will have minimal & managable effect on the process.

- Multiprocessor systems (MIMD) use 2 approaches:
  a) i. Tightly coupled system (shared memory)
  b) loosely coupled system (distributed memory).

a) Tightly coupled system: Memory & resources are shared between processors. This sharing can speed up the execution of a large program. I/O facilities are interconnected by a bus or other internal connection

scheme, such that memory access time is approximately the same for each processor. All processors can perform the same functions. The system is operated by a single integrated OS. Multiple processors can operate parallely to solve a single problem. This system is also called Symmetric Multiprocessor System (SMP).

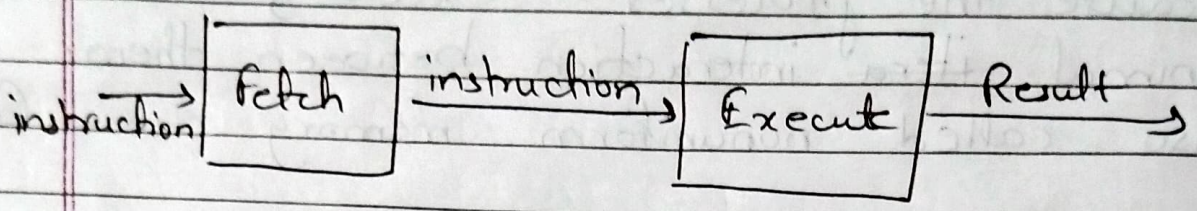


b) loosely coupled system:
Memory is distributed to all system processors, thus each processor has local memory. There is low degree of interaction between tasks. Allows a system to be more adaptable, writing additional code for adding new features is possible, without breaking the existing functionality, using send() & receive() functions to interact among processors. A system can be grown using this, making it scalable. There is a different OS for

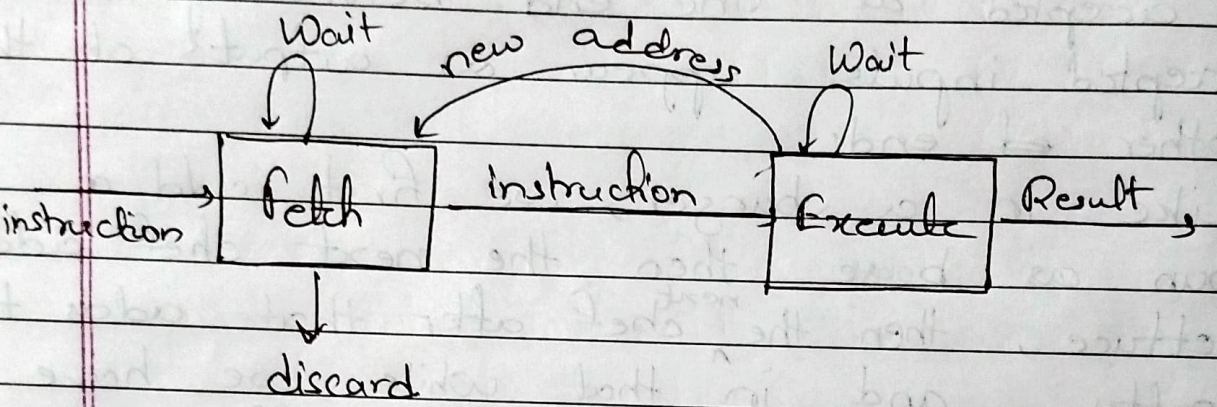
each processor. Memory access is faster because the processes executing have minimal ~~itera~~ interaction between them. Also called nonuniform memory access (NUMA).

Pipelining: In a pipeline, new inputs are accepted at one end before previously accepted inputs appear as outputs at the other ~~ef~~ end.
Like in a burger, we first add a bun as base then the next chef adds lettuce, then the next chef after that adds the patty, and in that while we have laid another bun on a different plate.
Thus, before getting the first output we have taken next input (plate).
Like while making a burger (multiple stages of instruction bun → lettuce → patty → spice → sauce → bun), instruction ~~from~~ for a processor has multiple stages. This means that multiple different stages can be worked on simultaneously like the burger factory.
Pipelining allows next instruction to be fetched ~~before~~ while the processor is performing arithmetic operations. All completed instructions are held in a buffer close to the processor until each instruction is ready to be performed.
The staging of instruction fetching is continuous.

A Simple 2 stage instruction pipeline:



Simplified view ↑



Expanded view ↑

The above 2 are independent stages, as mentioned earlier, the fetch fetches an instruction and buffers it. When Execute is ready, fetch sends the buffered instruction for execution. Thus the process is divided into 2 stages.

To gain further speedup, the pipeline must have more stages:

- Fetch instruction (FI): Reads the next expected instruction into a buffer.
- Decode instruction (DI): Determine the opcode & the operand specifiers.

OPCODE → operation code, specifies the operation to be performed.

operand specifier → on exactly what the operation is to be performed.

- Calculate Operands (CO): Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect or other forms of address calculation.

- Fetch Operands (FO): Fetches each operand from memory. Operands in registers need not be fetched though.

- Execute Instruction (EI): Perform the indicated operation and store the result, if any, in the specified destination operand location.

- Write Operation (WO): Store the result in memory.

With this decomposition, ignoring some factors, the various stages will be of nearly equal duration. This is shown using the following diagram:

PTO

time (units)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
" 2		FI	DI	CO	FO	EI	WO							
" 3			FI	DI	CO	FO	EI	WO						
" 4				FI	DI	CO	FO	EI	WO					
" 5					FI	DI	CO	FO	EI	WO				
" 6						FI	DI	CO	FO	EI	WO			
" 7							FI	DI	CO	FO	EI	WO		
" 8								FI	DI	CO	FO	EI	WO	
" 9									FI	DI	CO	FO	EI	WO

However, several factors limit the performance enhancement. In reality, the six stages are not of equal duration.

A conditional branch can also come in the way, ~~which~~ it can invalidate several instruction fetches.

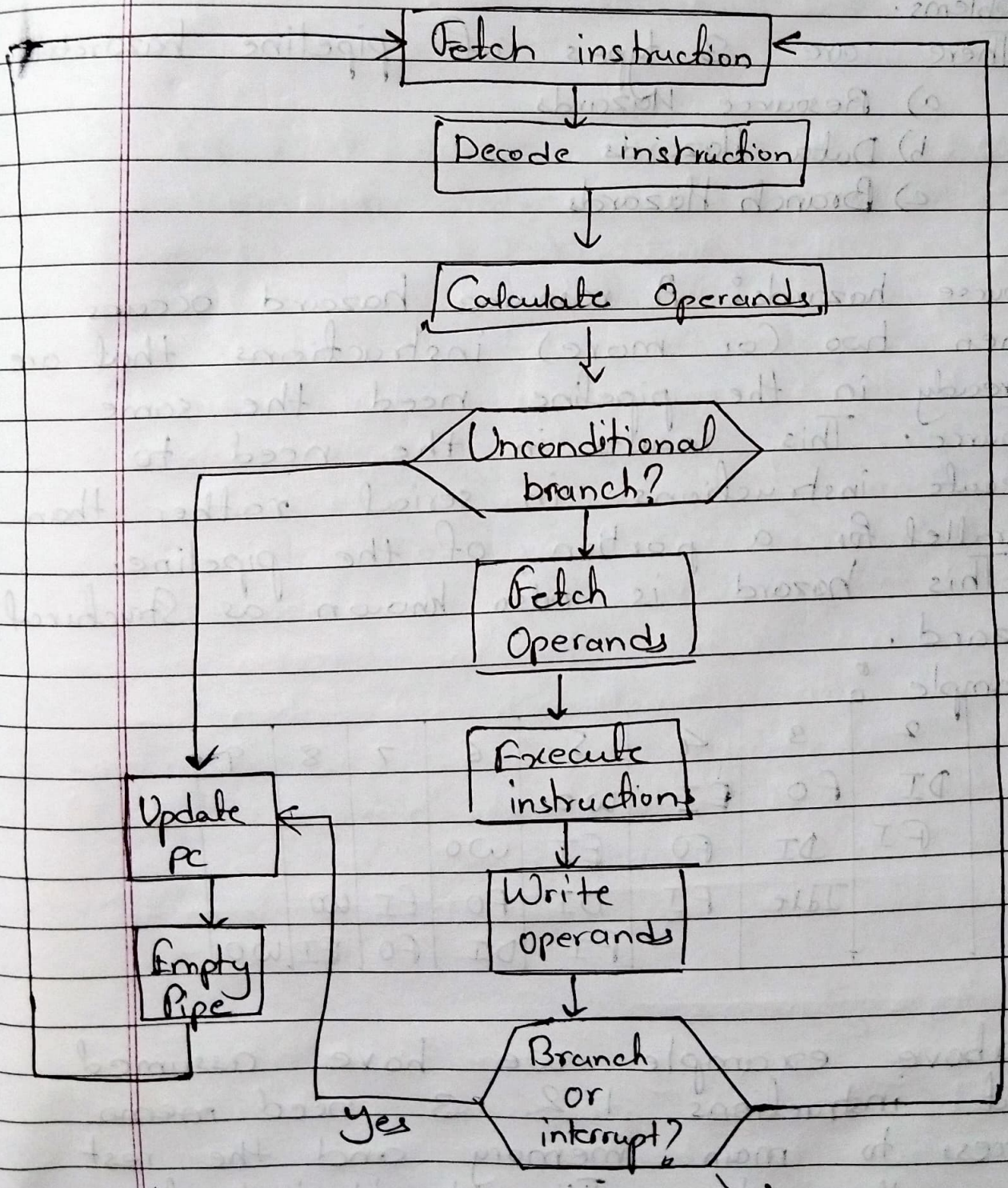
← time → ← Branch penalty →

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
" 2		FI	DI	CO	FO	EI	WO							
" 3			FI	DI	CO	FO	EI	WO						
" 4				FI	DI	CO	EI							
" 5					FI	DI	CO							
" 6						FI	DI							
" 7							FI							
" 15								FI	DI	CO	FO	EI	WO	
" 16									FI	DI	CO	FO	EI	WO

In the above timing diagram, we have assumed that instruction 3 is a conditional branch of instruction 15.

Thus, as soon as instruction 15 is fetched, it hinders the stages of other ~~ins~~

instructions till its complete since the control goes back to instruction 3.
between time units 9 to 12, instructions do not complete, thus making it branch penalty.
The timing diagran expressed as flowchart.



→ ~~necany~~ means any hindrance?
IG

Pipeline Hazards: occurs when a pipline, or some portion of the pipeline must stall because conditions do not permit continued execution. The previous example with instructions 1 & 3 was a pipeline hazard. This can occur due to inter instruction dependencies or job scheduling problems.

There are 3 types of pipeline hazards:

a) Resource Hazards
b) Data Hazards
c) Branch Hazards

a) Resource hazards: A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource. This results in the need to execute instructions in serial rather than parallel for a portion of the pipeline.

This hazard is also known as Structural hazard.

Example:

	1	2	3	4	5	6	7	8	9
I1	FI	DI	FO	EI	WO				
I2		FI	DI	FO	EI	WO			
I3			Idle	FI	DI	FO	EI	WO	
I4					FI	DI	FO	EI	WO

in above example we have assumed that instructions 1 & 3 need memo access to main memory and the rest from other places. Till 1st ~~ist~~ instruction

has not fetched the operands from main memory (FO), 3rd instruction can't be fetched; hence for instruction 3, the third unit of time is idle.

A solution for this is to use separate cache for instruction & data. We can also increase the number of resources.

b) Data Hazards: Occurs when instruction in pipeline depends on the result of the previous instruction that is still in pipeline yet to be completed. There are 3 types of data hazards:

i) Read after write (RAW): An instruction modifies a register or memory location and a succeeding instruction reads the data in that memory or register location. A hazard occurs if the succeeding instruction reads the result before the original instruction has completed modifying it or in other words, before it has completed the write operation.

ii) Write after Read (WAR): An instruction reads a register or memory location and a succeeding instruction writes to the location. A hazard occurs when succeeding instruction writes to the location before original instruction has read it.

iii) Write after Write (WAW): Two instructions both write to the same location. Hazard occurs when succeeding instruction writes before the original ~~one~~ instruction.

The above hazards can only occur in pipelining, if strictly sequentially instructions were ~~exe~~ executed then the output would be correct.

RAW: Add AX, BX
MOV DX, AX

WAR: MOV AX, DX
Add AX, BX

WAW: Add AX, BX
Sub DX, AX

c) ~~Branch hart~~ Branch hazards (or control hazards):
The previously discussed example, where instruction 3 was a conditional branch of instruction 15, caused a hazard, that hazard is called Branch Hazard.
It also occurs when pipeline makes a wrong decision on a branch prediction & thus brings instructions into the pipeline that need to be discarded (like instruction 4, 5, 6, 7).

~~Approaches to deal with branch hazards:~~

- ~~Multiple Streams:~~

Branching basically means: transfer of control (jumps, procedure call/returns, successful branches).

Approches to deal with branch hazards:

- Multiple Streams: Pipelines split. Multiple pipelines / streams are created with each branch so that at least one pipeline will have the correct output.
It's like the ~~thou~~ parallel world / universes. With every possibility our universe splits such that there ~~are~~ ~~mutt~~ is one universe for each possibility.

- Prefetch branch target: When a conditional branch is recognised, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed.
(target here means goal of the branch)
- Loop buffer: A loop buffer contains the n most recently fetched instructions, in sequence. If a branch is to be taken, the hardware first checks whether the branch target is in the buffer, if it is, the next instruction is fetched from the buffer. Loop buffer ~~is~~ ~~be~~ acts ~~as~~ as a guide, basically.

- Branch Prediction: We can try to predict a branch using various techiniques.

- Delayed branch: improving pefromance by ~~of~~ automatically rearranging instructions in a program.

There are 2 designs for pipelines:

- Instruction Pipeline design:
  Uses instruction issuing, the processor is trying to look ahead of current point of execution to locate instructions that can be brought into the pipeline.
  There are 3 types of issuing:
  a) In order
  b) Out - of - order
  c) reorder

- Arithmetic Ppeline design: Arithmetic pipeline divides an arithmetic problem into various sub problems for execution in various pipeline segments. It is used for fixed point operations, floating point operations, integer problems, etc.

Principles of designing pipelined processors:

a) Proper data buffering to avoid congestion and smooth pipelined operations.
(Data buffer is a region of a memory used to temporarily store data while it is being moved from one place to another)

b) Instruction dependence relationship (whatever that means).

c) Logic hazards should be detected and resolved.

d) Avoid collisions and structural hazards (Instruction hazards) by proper sequencing.

e) Reconfiguration of the pipeline should be possible.