④ Real performance - flops for specific operation
⑤ Sustained performance - performance on an application

Page No.
Date

* Sustained Performance < Real < Theoretical Peak

03/01/24

Continued on 09/01

# High Performance Computation (HPC)

Terms

① flops - floating point operation per sec.

→ HPC is teraflops or more ($10^{12}$)
→ petaflops = $10^{15}$ flops. (quadrillion)

② → Mips - Mega instructions per sec.
if there is only one istruction " " = MegaHz.

③ Theoretical Peak Performance - max flops a machine can reach
= CPU speed in GigaHz x No. of CPU cores (process) x CPU inst. per cycle x no. of CPU per node

④ Portability -
can be achieved by containerization
(storing all kind of libraries, modules used for particular process)
→ docker
→ future-proofing

② Modular and scalable
benchmarked and universal so that others can also use.

③ Grid-ready → follows grid-computing for a bigger server, efficient utilization of resources.

# Applications -

① Climate modelling - very big real time data requiring HPC.
② Physics - larger calculations for FT, calculus
③ Financial - stock prediction, currency fluctuations.
④ Banking sector - credits
⑤ Business sector - product prototyping
⑤ Image segmentation (x-ray, MRI)
⑥ Fluid dynamics
⑦ Drug discovery
⑨ gaming

Page No.
Date : / /

# Challenges

(1) cost increases
(2) skilled professional
(3) Integration

In HPC, latency should be low, speed should be high and it should match with the memory.

(4) speed depends on memory. There should be no mismatch.
(5) Power consumption.

# Parallel computation

Divide the instruction in multiple units and each unit is performed by different processors.

→ Motivation for parallelism computation
(2) ↑se no. of transistors in IC for ↑ing the computation power
(3) improvement in storage technology (in disks and memory)

Study kya hota hai #

# Locality of Reference (Technique)

(3) improvement the networking devices and system for data communication. (dependent on internet)

Time to execute a given program: $T = n_c \times t_c$

$T = n_i \times (\text{avg}) \times t_c$

$= n_i \times CPI \times t_c$

$n_c$ = total no of CPU cycles

$t_c$ = time cycle time

$n_i$ = total no of instructions

$n_c/n_i$ = Avg. cycles per instruct

(continue at a JAN)

→ Motivation:

1.) → Parallel computation

2.) ↑se no. of transistors in ICs for ↑ising the computer power

3.) improvement in storage technology (in disks & memory)

eg:- L1 cache, L2 cache, main memory etc.

(#) Locality of Reference (Technique):

(9.) Improving the networking devices & system for data communication (dependent on internet)

5/1/24

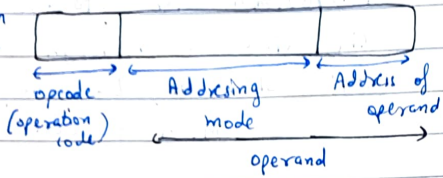
## Classification based on parallel computing

- based on instruction & data streams
- based on structure of computers
- based on how memory is accessed
- based on grain size → instruct" size

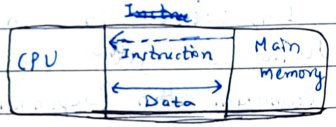
(i) Stream → Sequence or flow Instructions

Grain size → size of instruction of the program.

Instruction cycle




Instruction stream - Instructions come from main memory to CPU.




Data stream - bidirectional

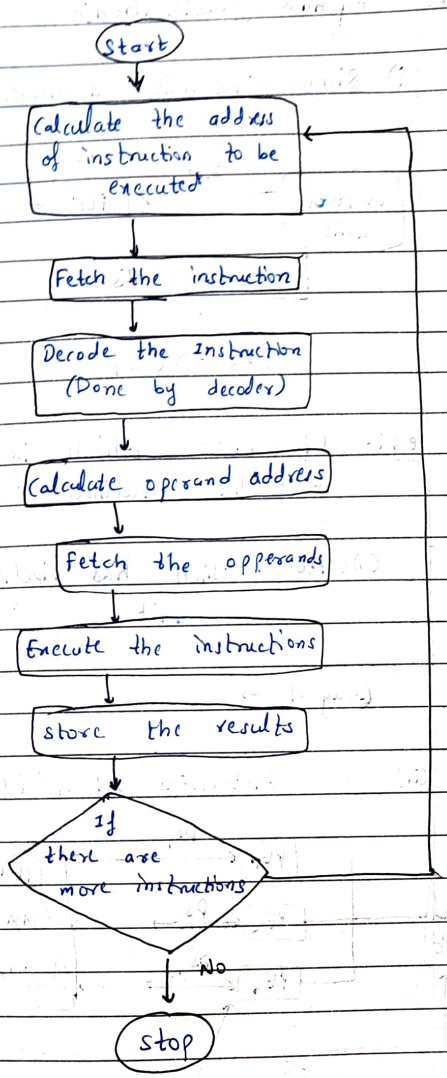
Instruction stream - Unidirectional

## Flynn Classification

whenever architecture uses instr. & data stream.

not ~~necessary~~ necessary that it uses parallel architecture

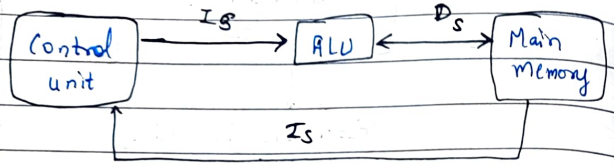
Instruction clock style



Page No.
Date

## # Flynn's classification:-

### i) Single instruction & single data stream (SISD)



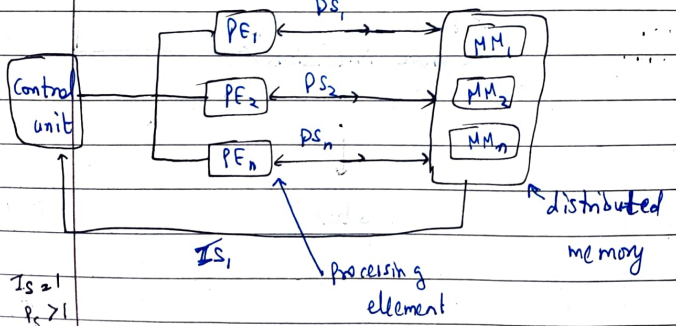

$I_S = D_S = 1$

eg:- CDC 6600 which is unpipelined but has multiple functional units

CDC 7600 → which has pipelined arithmetic unit.

Cray-1 →

### (ii) single Instructions and multiple data streams




Data streams coming from multiple memory locations
One instruction is divided & given to multiple ~~pe~~ processing elements.

Each PE can have its own memory. (eg:- ~~printer~~ printer / fax ~~machine~~ machine can have~~s~~ own memory)

=> Global communication & another main memory communication

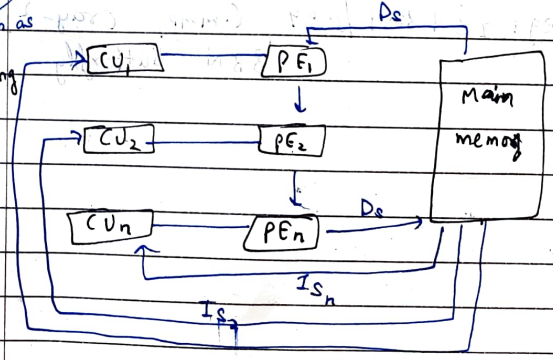
agar comp. ke sath printer / fax machine hai toh unki bhi comm^n overall ho sakti hai

~~jis~~ computer ke ~~kaam~~ kaam kr rahi hain ushi memory.

eg:- BSP, MPP, DAP, CM-1 (Connection machine.)

### (iii) Multiple instructions & single data stream.

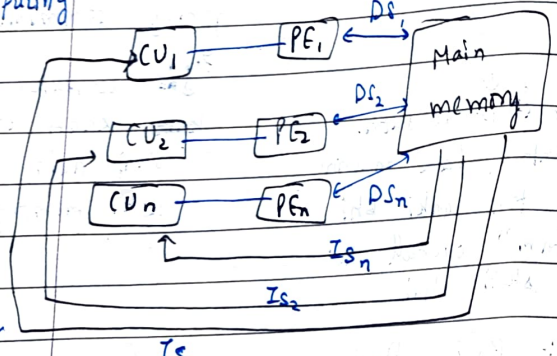
also known as N-version programming



eg:- ~~carne~~ Carne Millon university
derived → C.mmp

not used so frequently { Real time computers must be fault tolerant
But here if fault in PC → This same goes to $PE_2$ → So not fault tolerant

iv) multiple instruct$^n$ & multiple data streams
also known as parallel computing




$Is > 1$
$DS > 1$

This is asynchronous

eg: IBM 3081/3084, C.mmp, Cray-2, BBN-Butterfly

JAN

# To ↑ sc. computer speed / speed of program

① reduce cycle time ($t_c$) by ↑sing clock freq.
② reduce number of instructions
③ reduce CPI → using parallelism

- ① Instruction level - Pipelining {overlapping of Instruct$^n$}
- ② Internal Parallelism
- ③ External Parallelism

# Temporal Parallelism :-

Time-based

→ This methods breaks up job into a set of tasks to be executed overlapped in time

→ This is also called assembly / pipeline / vector processing

Conditions: (in which temporal llism can be applied) :-

→ The jobs to be carried out are identical.

→ A job can be divided into many independent tasks.

09/01/24

Page No.
Date : / /

⑤ Moore's law

No. of transistors on computer chips doubles approximately every 2 years.
This implies CPU speed doubles every 18 months.

Time to execute a given program., $T = n_c \times t_c$

$N_c$ = total no. of CPU cycles or

$t_c$ = cycle time.

$n_i$ = total no. of instructions

$T = n_i \times \left(\frac{n_c}{n_i}\right) \times t_c$

($\frac{n_c}{n_i}$ ← avg cycles per instruction)

$= n_i \times CPI \times t_c$

# To ↑se computer speed / speed of program
① reduce cycle time ($t_c$) by ↑sing clock freq.
② reduce no. of instructions ($n_i$) by using better compilers and efficient algo
③ reduce CPI by using parallelism.
Types of Ilism → ① instruction level Ilism by pipelining - overlapping of instructions.
② internal Ilism
③ external Ilism

—x—

# Temporal parallelism ⇒ This method breaks up a job into a set of tasks to be executed overlapped in time.
(Temporal = time-based)
aka assembly-line processing, pipeline processing & vector processing.

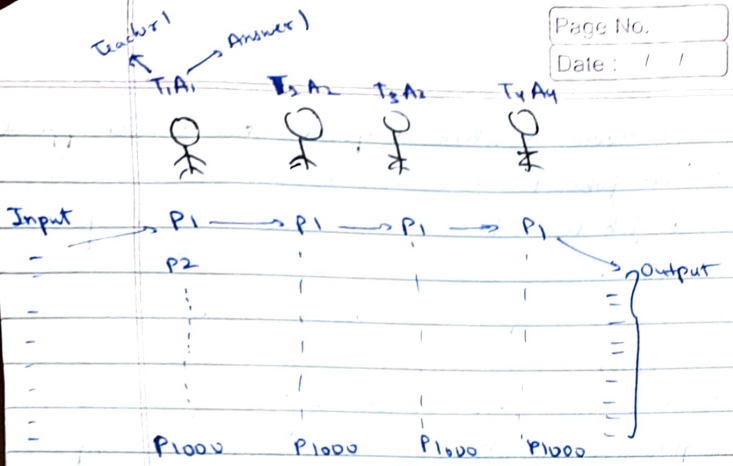
→ Conditions in which temporal Ilism can be applied
① The jobs to be carried out are identical.
② A job can be divided into many independent tasks.
③ Time taken for each task should be approximately same.
④ Time taken to send a job from one processor to the next is negligible compared to the time needed to do a task,
⑤ the no. of tasks are much smaller as compared to the total no. of jobs to be done.

Let n = no. of jobs
p = time to do a job
k = no. of tasks (let each job be divided into k tasks)
t = time to do each task = $\frac{p}{k}$

* Time to complete n jobs with no pipeline processing (no parallelism) = $np$
* Time " " with pipelining = $p + \frac{(n-1)p}{k}$

→ This can be sped up by pipelining in which the time taken will be sped up by → $\frac{np}{p + \frac{(n-1)p}{k}}$ = Ratio of w/o pipelining with pipelining

$= \frac{np}{\left(\frac{k+n-1}{k}\right)p}$




Disadvantages

(1) Synchronization – for this identical time should be taken for each task in the pipeline so that a job can flow smoothly in the pipeline without holder (ek job koi bohot der lekar na baitha ho).

(2) Bubbles in the pipeline – If some tasks are absent in a job, bubbles are formed in the pipeline. other processor will sit idle if Q3, Q4 are not attempted.

(3) Fault tolerance – If a processor crashes in b/w a cycle, synchronization breaks and thus the system doesnot tolerates fault.

(4) Inter-task communication – If too much time is consumed to pass the task b/w the processors in the pipeline. This time should be smaller as compared to the time taken to execute the job.

(5) Scalibility – The no. of processors working in the pipeline cannot be ↑ed after a certain limit. There comes a limit after which no. of processors cant be ↑ed.

10/01/24

speed up by $= \frac{n}{\frac{(k+n-1)}{k}} = \frac{nk}{(k+n-1)}$

$= \frac{k}{1+\frac{k-1}{n}}$

If $n \gg k$ then $\frac{k-1}{n} \simeq 0$ and speed up $= k$ (↳ no. of task in which a job is divided)

Inference ⇒ and each task is given to different processor

speed up is proportional to no. of processors for condition – no. of job >> no. of task

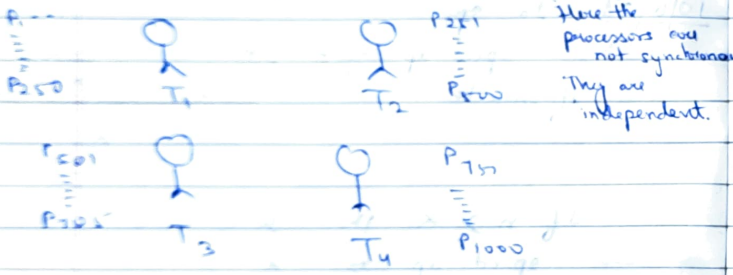
Adv → (i) each processor can be fine-tuned to do one kind of a job.

→ One processor can become expertise for one kind of task.

* If a speed up of a method is directly prop to the no. of processors then the method is said to be scale well
→ temporal Ilem scales well.

---

# Data Ilem with static-assignment
The i/p data is divided into individual sets and processed simultaneously.




Here the processors are not synchronous. They are independent.

Let
n = no. of jobs
p = time to do a job
k = no. of processors to do the job
kxq = kq = time to distribute the job to k processors (teachers)

time for single k processor job distribution

Observe that this time is prop. to no of processors.

∴ time to complete n jobs by single processor = np

∴ " " by k processors = $\frac{np}{k} + kq$

speed up = $\frac{np}{\frac{np}{k} + kq} = \frac{knp}{k^2q + np}$

$= \frac{k}{1 + \frac{k^2q}{np}}$

If $k^2q < np$ then speed up = k.

# efficiency = $\frac{\text{speed-up}}{k}$

If the no. of processors ↑ses, the time to distribute jobs to processors also ↑ses.
Hence, speed up is not directly prop to the no. of processors.

Advantages

(1) No synchronisation required - all teachers can check copies without dependency on other
(2) Dependency due to Bubbles are not present. All processor can perform complete job. If bubble arises then processor can tackle independently.
(3) More fault tolerance. Agar koi break kele toh dusre effect ni honge
(4) No inter task communication delay
↳ does not exist here

Page No.
Date : / /

Difference between basic

→ Temporal – has processor ek particular task hi karr sakta hai pipeline.

→ Data – The jobs are divided and given to different processor which can do complete job independently

Disadvantages

① Static assignment – The assignment of the jobs to each processor is pre-decided. Thus, if a processor is slow then the completion time of the total job will be slowed down. If there are more bubbles, that single processor will take more time.

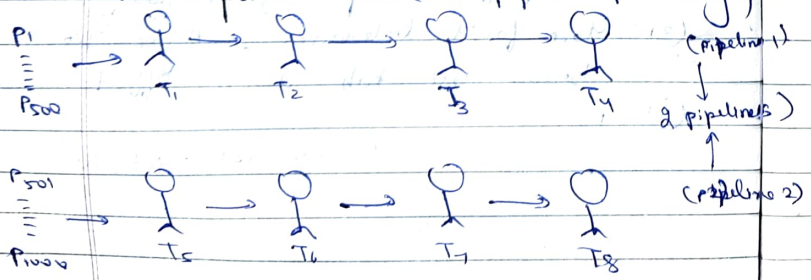
② We must be able to divide the set of jobs into subsets of mutually independent jobs, each subset should take the same time to get completed ⇒ evenly distributed

③ Each processor must be capable of executing all the jobs.

④ The time taken to divide the set of jobs into equal subsets of jobs should be small

aka

# Mixed parallelism (Hel pipeline processing)




→ This method is effective only if the no. of jobs given to each pipeline is much larger than the stages/segment of the pipeline ($T_1, T_2, T_3, T_4$)

→ Both pipelines process simultaneously

→ This method almost halves the time taken by a single pipeline.

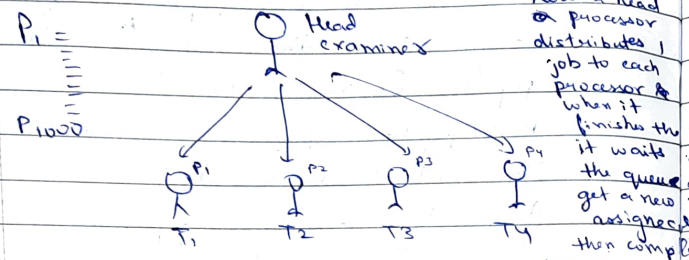
Disad → both disadv of temporal & data

Examples of computers performing this Hein → Cray, NEC-SX

This method is very efficient for numerical computing in which a no. of long vectors and large matrices are used as data and could be processed simultaneously.

Page No.
Date : / /

# Data Ilsim with dynamic assignment




Here a head processor distributes 1 job to each processor & when it finishes the job it waits in the queue & to get a new job assigned then completes it

This process continues until all the jobs are completed.

Advantage

1. Balancing of the work assigned to each processor ~~directly~~ dynamically as the processor progresses.
2. A processor that finishes the job quickly gets another job immediately and does not need to sit idle.
3. Time to execute a job may widely vary without creating a bottleneck.
4. This method is not affected by bubbles.

Disadvantages

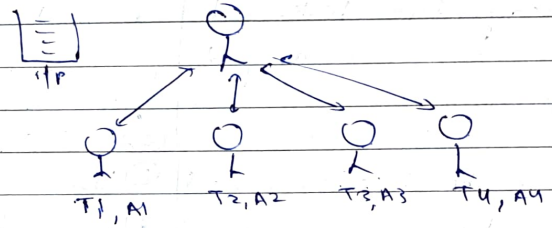
1. If many processors complete their job simultaneously, they need to wait in the queue as only one will be attended at a time.
2. The main processor can become a bottleneck
3. The main processor itself is idle b/w handing out jobs.
4. It is difficult to increase the no. of processors as it will increase the prob of many processors completing their jobs simultaneously thereby leading to long queue of processors.

# Data Ilsim Quasi Dynamic Scheduling

Giving each teacher/processor unequal sets of answersheets to correct / jobs

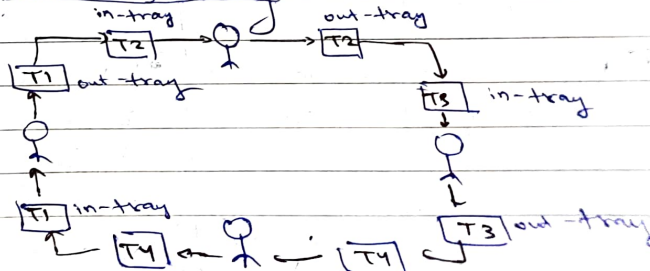
# Other Ilsims

1. Specialist : data Ilsim




har processor do one task of one job.

2. Coarse Grained specialist Temporal Parallel Processing -



→ Instead of spending time to wait for other processors to complete their tasks the processors takes the task from their in-tray, executes it and then place it in their out-tray.

→ Initially, if there is no task in its in-tray it will wait for it to be filled.

# Temporal Vs Data Ilism

| Temporal | Data. |
|---|---|
| ① Job is divided into set of independent tasks and these tasks are assigned for processing | ① Complete job is assigned to the processor. |
| ② Tasks should contain equal time. Pipeline stages should thus be synchronized. | ② Jobs may take different times. No need to synchronize the jobs. |
| ③ Bubbles in jobs lead to idleing of processors | ③ Bubbles dont cause idleing of processors. |
| ④ Processors are specialized to do specific tasks efficiently | ④ Processors should be general purpose and may not do all tasks efficiently |
| ⑤ Not tolerant to processor faults | ⑤ Tolerates processor faults. |
| ⑥ Task assignment is static (pre-decided). | ⑥ Task assignment can be static, dynamic & quasi dynamic. |
| ⑦ efficient with fine-grained tasks (more specialized tasks) | ⑦ efficient with coarse grained task and quasi dynamic scheduling (less specialized task) |

—— x ——

16/01/24 **Pipelining**

Pipeline is useful when same processing is applied over multiple units.

→ It is a technique to decompose a sequential process into sub-operations

Task = one operation performed at all segments / stages.

example – $A_i \times B_i + C_i$ , $i = 1, 2, \ldots 5$

$R_1 \leftarrow A_i$ $R_2 \leftarrow B_i$

$R_3 \leftarrow R_1 \times R_2$

$R_4 \leftarrow C_i$

$R_5 \leftarrow R_3 + R_4$

| Clock Pulse | Segment 1 | | Segment 2 | | Segment 3 |
|---|---|---|---|---|---|
| | R1 | R2 | R3 | R4 | R5 |
| 1 | A1 | B1 | | | |
| 2 | A2 | B2 | A1*B1 | C1 | A1*B1+C1 |
| 3 | A3 (simultaneously fetch as well as execute bcz it is pipelining) | B3 | A2*B2 | C2 | (One inst executed in 3 clock cycles) → A2*B2+C2 |
| 4 | A4 | B4 | A3*B3 | C3 | A3*B3+C3 |
| 5 | A5 | B5 | A4*B4 | C4 | A4*B4+C4 |
| 6 | – | – | A5*B5 | C5 | A5*B5+C5 |

Total time with pipelining = 7 (3 for 1st, 1 for rest all)

without pipelining = 15

Time required to perform 1st task = $k \times t_p$ (k → no. of segments, $t_p$ → time for one segment)

Time " " remaining (n−1) task = $(n-1) \times t_p$

Total time = $(k+n-1)\,t_p$

# Consider a non-pipelining system that takes $t_n$ time to perform a task (often sequential)

⇒ for n tasks = $n \times t_n$

★ speed up = $\frac{n \times t_n}{(k+n-1)\,t_p} = \left(\frac{\text{non-pipelining}}{\text{pipelining}}\right)$

Case 1 – If $n >>> k$, speedup = $\frac{t_n}{t_p}$ (max speedup) = ideal

Case 2 – When time req to perform one task is same in pipeline and without pipeline :-

⇒ speed up = $\frac{n}{k+n-1}$

(can't be greater than k)

## 5-segment instruction Pipeline

Steps

1. Instruction fetch (IF)
2. Decode & Address calculation (DA)
3. Operand Fetch (OF)
4. Execution (Ex)
5. WB (Write-Back Result)

| Clock Pulse → / Inst ↓ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| I1 | IF | DA | OF | Ex | WB | | | |
| I2 | | IF | DA | OF | Ex | WB | | |
| I3 | | | IF | DA | OF | Ex | WB | |
| I4 | | | | IF | DA | OF | Ex | WB |
| I5 | | | | | IF | – | – | |
| I6 | | | | | | | | |
| I7 | | | | | | | | |

Assume this is branched instruction

Stall cycles (wasted time)

• This I5 will not be fetched but discarded until I4 (condition) is executed, bcz it is a branched statement. Eh baar I5 fetch nahi discard hogaya

This I7 will be fetched on 8th cycle after Ex of I4

# Branch cycle also results in Tee in time

| | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|
| I7 | IF | DA | OF | Ex | WB | |
| I8 | | IF | DA | OF | Ex | WB |

Page No.
Date : / /

Total instructions executed = 6 (I1 - I4, I7, I8)

⇒ Condition is checked in the execution stage of the branch instruction and the instructions which are fetched after the branched inst will get discarded. (these are the stall cycles which are discarded and wasted time)

Here, k = 5
n = 6
Acc to formula = k+n-1
= 10
But in actual = 13

→ These extra cycles taken by pipeline because of any hazard or reason are stall cycles.

∴ Total cycles = k+n-1+stall cycles

→ If branch condition is evaluated in ith segment (here it was in 4th) then the no. of stall cycles are i-1.

Page No.
Date : / /

delay in inst due to any reason

## Pipeline Hazards

Pipeline hazards are often seen as mile-steps in the optimization of processes within the pipelining concept.

→ Hazard can occur during stages ① fetch ② decode or ③ execution

17/01/24

① Hazard are usually related to issues with the memory system such as latency. (conflict b/w 2 instructions)

Decoding ② An instruction might need to wait for completion of another instruction which currently occupies the decoder.

Execution ③ If there is a conflict in accessing a functional unit (like register, ALU)

# Types of pipeline hazards

① Structural Hazard / Resource conflict

These occur when the same hardware resource is desired by multiple instructions simultaneously.

MUL takes more time for exe than ADD

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| MUL | IF | DA | OF | EX | EX | WB | |
| ADD | | IF | DA | OF | EX | EX | WB |

EX becomes stall cycle bcz ALU is busy with MUL and is executed later

Page No.
Date : / /

Solutions to this – (incase one is dependent on other instr.)

① incur stall cycles. (increase)
② ↑se no. of resources. (not much feasible)

② Data Hazard / Data dependency

When the execution of one inst. is dependent on completion of the other.

These are categorized into 3 types

a) RAW (Read After Write)
- aka True dependency
- occurs when an inst. depends on result of prev. inst

eg- i: $R_1 \leftarrow R_2 + R_3$
j: $R_5 \leftarrow R_1 + R_4$

→ j reads a source before it is written by i.

# here you also have to check kitne time baad dependency aa rahi hai else no Hazards occur. eg- I1 and I5 are dependent. So I5 take I1 bhi complete ho jaega, no stalls would occur

b) WAR (Write after Read)
- aka anti-dependency & false dependency
- when an inst. depends on the reading of a value before that value is overwritten by its previous instruction

(pehle kuch value thi $R_1$ me but is updated later. But you read prev val not updated one)

Page No.
Date : / /

eg i: $R_1 \leftarrow R_2 + R_3$
j: $R_2 \leftarrow R_4 + R_7$

→ If j first write in $R_2$ and i reads later.

c) WAW (Write after Write)
- aka write-dependency & false dependency
- when a value is written by the instruction befor the prev inst. writes that value.

eg: i $R_1 \leftarrow R_2 + R_3$
j: $R_1 \leftarrow R_4 + R_7$

→ If j writes in source before i writes. (duno ~~kaise~~ apne time pe fetch hui. lekin j got exe before i and then i got exe, so this is hazard bcz current value should be j but here it is i).

# Register Renaming
* We can write the result in another register in case of false dependency. Value list or reg me store karli

Solutions to this –

① Compiler or language translator will generate a program where data dependency is not present (aka Delayed load) – Software s/n.
This delay is produced by writing other independent or no ops instructions b/w i and j.

eg i: $R_1 \leftarrow R_2 + R_3$
j: $R_4 \leftarrow R_1 + R_5$

IF DA OF EX WA

All segments have diffe seprate H/ws

(2) H/w sol$^n$ - ~~after detecting data dependency~~.

When a H/w can detect and solve the data dependency.

(i) Hardware Interlock

when an inst is executed, all segments have seprate hardware, when values is stored in one hardware that particular H/w is locked until the whole inst is executed. when it completes, then it allows other inst. to update further values.

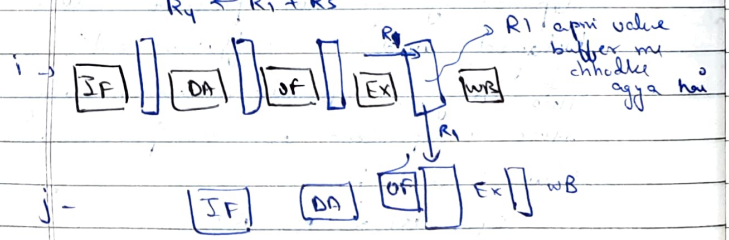
→ All prev H/ws get locked until WB is done.


(ii) Operand Forwarding / Bypassing

All segments have own buffer to store prev. value results in it.

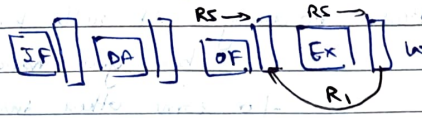
eg $R_1 \leftarrow R_2 + R_3$

$R_4 \leftarrow R_1 + R_5$




→ $R_1$ is executed and stored in Ex buffer And then becomes stored in OF buffer (dependent - $R_1$)

$R_2$ - not dependent




This does not make us wait till WB and removes time delay and stall cycles.

⇒ No stalls for ALU to ALU dependency. But for ALU to Mem or Memory to ALU stalls are required.

Condition Done inst ALU me exe honi chahiye

(3) Control Hazard (Branch Difficulty)

When the CPU decodes an inst as branch inst then CPU has to wait till branch outcome is available to fetch new instruction.

Solutions

s/w soln (i) Delayed Branch

Jab tak branch inst exe, add nops and independent inst b/w

Page No.
Date : / /

② Hdw soln – Branch Prediction.
It predicts whether the current branch inst will jump or not.
If Pred = correct – no prob
Pred = wrong – it rolls back bcz you have gone to some other inst and then comes back.

— x —

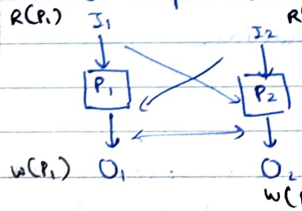
14/01/24 Bernsteen's Conditions

Bernsteen gave a set of conditions on the basis of which we check if 2 processes can execute ||ely or not

Here, $P_i$ → software entity
Input → $I_i$ (set of all i/p variables for a process)
↳ aka Readset / Domain of $P_i$

Output → $O_i$ (set of all o/p variables)
aka Write Set / Range

→ Consider 2 processes which are interdependent




For 2 processes to execute ||ely they should be independent of...

$I_1$ = Read of $P_1$ = $R(P_1)$
$O_2$ = Write of $P_2$ = $W(P_2)$

Page No.
Date : / /

Conditions (given by Bernsteens)

① $I_1 \cap O_2 = \phi$ (anti-independent)
i/p of 1st process and o/p of 2nd process should have nothing common

② $I_2 \cap O_1 = \phi$ (flow-independent)
Read($P_2$) should be independent of $W(P_1)$

③ $O_1 \cap O_2 = \phi$ (output independent)

# In general, $P_1, P_2 \dots P_k$ can execute in ||el if Bernstein's conditions are satisfied on a pair wise basis.

⇒ $P_1 \,||\, P_2 \,||\, P_3 \dots ||\, P_k$ [iff $P_i || P_j$ for all $i \neq j$]

Note → If $I_1 \cap I_2 \neq \phi$ → it does not prevent ||elism or $= \phi$ ($I_1, I_2$ see bank wali pdf)

Properties of ||elism relation

① Commutative ($P_i || P_j \iff P_j || P_i$)

② Non Transitive (if $P_i || P_j$ & $P_i || P_k$, does not imply $P_i || P_k$)

③ Associative $[(P_i || P_j) || P_k = P_i || (P_j || P_k)]$

P.T.O

eg- $P_1 \Rightarrow C = D \times E$

$P_2 \Rightarrow M = G + C$

$P_3 \Rightarrow A = B + C$

$P_4 \Rightarrow C = L + M$

$P_5 \Rightarrow F = G + E$ , Check Hazard

| Stmts | $I_i$ | $O_i$ |
|---|---|---|
| $P_1$ | $\{D, E\}$ | $\{C\}$ |
| $P_2$ | $\{G, C\}$ | $\{M\}$ |
| $P_3$ | $\{B, C\}$ | $\{A\}$ |
| $P_4$ | $\{L, M\}$ | $\{C\}$ |
| $P_5$ | $\{G, E\}$ | $\{F\}$ |

You have to check Hazard for all pairs $\Rightarrow {}^5C_2 = 10$ pairs

a) for $P_1$ & $P_2$ : ① $I_1 \cap O_2 = \phi$

$\{D, E\} \cap \{M\} = \phi$ ✓

② $I_2 \cap O_1 = \phi$

$\{G, C\} \cap \{C\} \neq \phi$ ✗

$P_1 \not\parallel P_2$

b) for $P_1$ & $P_3$ ⊙ : $P_1 \not\parallel P_3$

c) $P_1 \not\parallel P_4$

d) $P_1 \parallel P_5$

e) $P_2 \parallel P_3$

f) $P_2 \not\parallel P_4$

g) $P_2 \parallel P_5$

h) $P_3 \not\parallel P_4$

i) $P_3 \parallel P_5$

j) $P_4 \parallel P_5$

—×—

As studied earlier, there can be delays in pipeline

## Types of Pipeline Delay

① Uniform Delay Pipeline

All stages / segments will complete their operations by taking the same time.

cycle Time ($T_p$) = Stage Delay.

If there are buffers b/w the stages, then

$T_p$ = Stage Delay + Buffer Delay

② Non-uniform Delay Pipeline

All stages will complete their operations by taking different times.

$T_p$ = max (Stage Delay)

eg- In 4-segment pipeline, delays are 1, 2, 3, 4 $\Rightarrow T_p = \max(1, 2, 3, 4)$

$= 4$ ns

If Buffers are there $\Rightarrow$ Max (Stage Delay + Buffer Delay)

eg- Assume 4-segment pipeline having stage delay as 2ns, 8ns, 3ns, 10ns. Here, we have to determine the time taken to execute 100 tasks in the above pipeline.

Here, $T_p = 10$ ns.

Page No.
Date :

Execution time = $(k+n-1)T_p$ = no. of cycles × $T_p$
= $(4+100-1) \times 10$
= 1030 ns

(4 = k, 100 = n, no. of cycles without stall for stall # done before)

# Formula for Performance of pipeline with stalls

(1)

speedup = S = $\frac{\text{Avg exe time non-pipeline}}{\text{Avg exe time pipeline}}$

$= \frac{(CPI \times \text{Cycle Time})_{\text{non pipeline}}}{(CPI \times \text{Cycle Time})_{\text{pipeline}}}$

Note ⇒ In the pipelined processor, ideal CPI = 1 but because of the stalls CPI becomes greater than 1.

Then speed up becomes –

$S = \frac{(CPI \times \text{cycle time})_{\text{non-pipline}}}{(1 + \text{no. of stalls per inst}^{n}) \times (\text{Cycle Time})_{\text{pipeline}}}$

$(CPI)_{\text{pipeline}}$

(2) If $(\text{Cycle Time})_{\text{pipeline}} = (\text{Cycle Time})_{\text{non-pipeline}}$ then

speedup = $\frac{(CPI)_{\text{non-pipeline}}}{1 + \text{no. of stalls per instruction}}$

wrong branch inst add stalls

PRAM Model

(Parallel Random Access Machine)

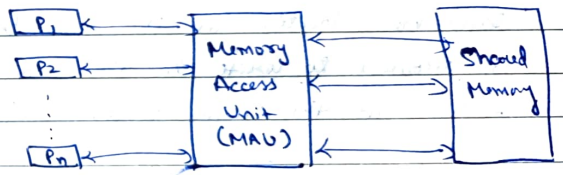
It is an abstract machine for designing an algorithm applicable to parallel computers.

A PRAM Model contains –

① a set of similar type of processors.

② all processors share a common memory unit.
processes can communicate among themselves through shared mem only

③ A Memory Access Unit (MAU) connects the processors with the single shared memory




PRAM Architecture

'h' no. of processors can independent operations on 'n' no. of data in a particular unit of time.

Problem ⇒ This may result in simultaneous access of same memory location by different processors.

Page No.
Date : / /

# To solve this, some constraints are there:

① Exclusive Read Exclusive Write (EREW)
No 2 processors are allowed to read from or write to the same memory location simultaneously.

② Exclusive Read Concurrent Write (ERCW)
No 2 processors are allowed to read from the same memory location at the same time but are allowed to write the same memory location simultaneously

③ Concurrent Read Exclusive Write (CREW)
All processors are allowed to read from the same mem location at the same time but not allowed to write to the same mem location at the same time.

④ Concurrent Read Concurrent Write (CRCW)
All processors are allowed to read from and write to the same mem location at the same time.

# Methods to implement PRAM Model

① Shared Memory Model
② Message Passing Model
③ Data Parallel Model