

Basics About Computer / clock cycle :

L So basically each operation happens in a fixed no of clock cycles and lesser the time to execute each cycle = 1/(3ghz) for a 3ghz processor means more is the speed of execution of instruction , am i right ?

Show drafts ▾ 🔍

◆ Your understanding is mostly correct, but there are some key points to keep in mind about processor clock speed and instruction execution:

Clock cycles and instructions:

- You're right that most individual instructions require a fixed number of clock cycles to complete. This number depends on the complexity of the instruction and the specific processor architecture.
- However, some processors can execute multiple instructions simultaneously through techniques like **pipelining** and **out-of-order execution**. These techniques can effectively overlap the execution of different instructions, reducing the overall time to complete a task.

Execution time and clock speed:

Execution time and clock speed:

- You're correct that the time to execute a single cycle in a 3 GHz processor is $1/(3 \text{ GHz}) = 0.33 \text{ nanoseconds}$. However, as mentioned earlier, most instructions take multiple cycles to complete.
- Additionally, the clock speed only reflects the internal clock of the processor and doesn't account for other factors that can affect performance, such as:
 - **Memory access times:** Accessing data from memory (RAM) takes longer than internal processor operations. Slower memory can bottleneck performance even with a fast processor.
 - **Cache size and efficiency:** Processors have caches that store frequently used data close at hand. Larger and more efficient caches can significantly reduce memory access times, improving performance.
 - **Software optimization:** The way software is written can significantly impact performance. Code that is optimized for the specific processor architecture will run faster than code that is not.

Overall:

Overall:

- Clock speed is an important factor in processor performance, but it's only one piece of the puzzle.
- The number of clock cycles per instruction, memory access times, cache efficiency, and software optimization all play crucial roles in determining the overall execution speed of a task.

I hope this clarifies the nuances of clock speed and its impact on instruction execution.

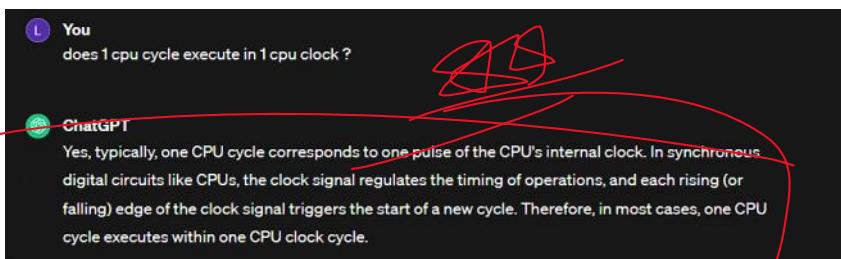
Yes, you're absolutely right! A 3 GHz processor does indeed mean it can execute **3 billion clock cycles in 1 second**. This is the core idea behind clock speed - it measures the raw processing power of a CPU by counting the number of cycles it can complete per second.

WHAT DOES DUAL CORE / QUAD CORE / MULTI CORE MEAN IN GENERAL :

Dual Core \rightarrow 2 ALU \rightarrow 1 W	Quad Core \rightarrow 4 ALU \rightarrow 1 W
---	---

1 2 3 4 5 6 7 8

Ye jo upar banate hai pipelining ke diagrams me these are basically CPU CYCLES !!



1 clock me : 1 cpu cycle chalta ha

You're absolutely right! Cycles per instruction (CPI) is exactly the same as clocks per instruction. Both terms refer to the average number of clock cycles a processor requires to execute a single instruction. Essentially, they are just different ways of saying the same thing:

- **Clocks per instruction:** Emphasizes the unit of measurement (clock cycles) used to count the execution time.
- **Cycles per instruction:** Emphasizes the specific execution unit (cycles) involved in processing the instruction.

Both terms are commonly used, and you can choose whichever one seems more intuitive for the context.

I hope this clarifies the equivalence between these two terms!

$$T = n_c \times t_c \Rightarrow n_i \times \frac{n_c}{n_i} \times t_c \Rightarrow n_i \times CPI \times t_c$$

$n_c \rightarrow$ no of CPU cycles
 $t_c \rightarrow$ time of the cycle

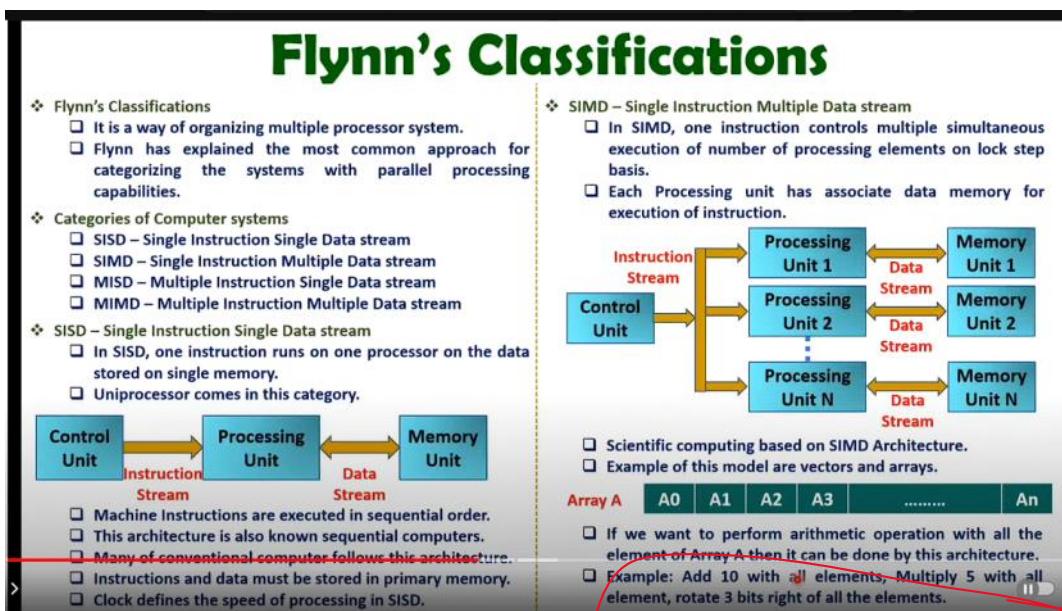
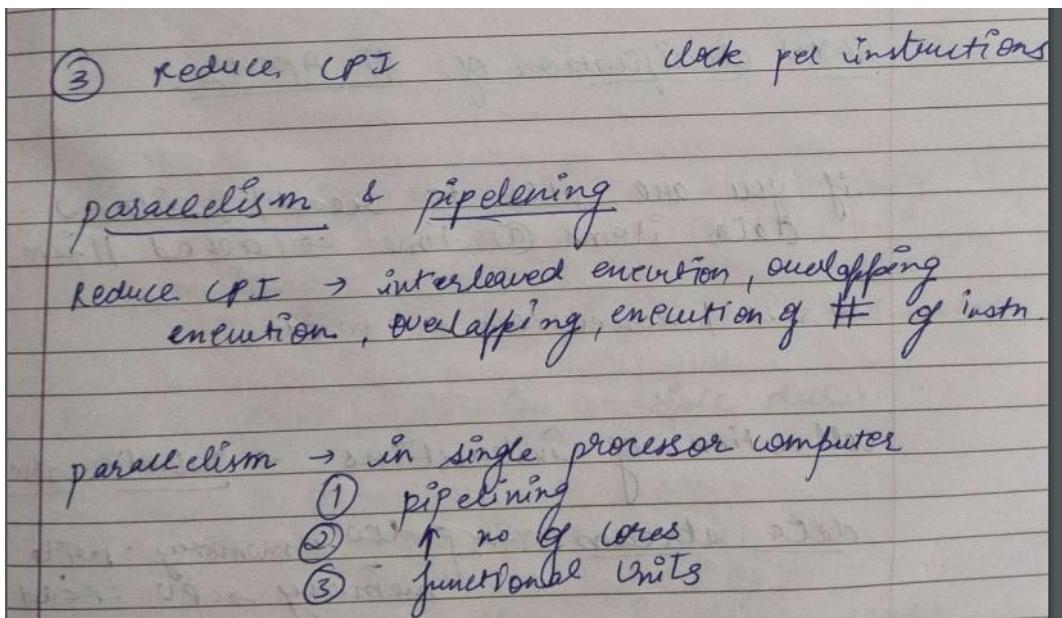
Date _____

$n_i \rightarrow$ no of instructions
 $CPI \rightarrow$ no of clocks required for 1 instr

IMP concept / Basic Concept :

oye sun agar maine kisi processor ki Clock Speed badha di that means ki 1 cpu cycle ko execute karne ka samay kam ho jayega that is : pipelining me jo upar cols banate the : unme har ek individual col ko execute karne ka time kam ho jayega that

is why : performance speed badh jayeagi

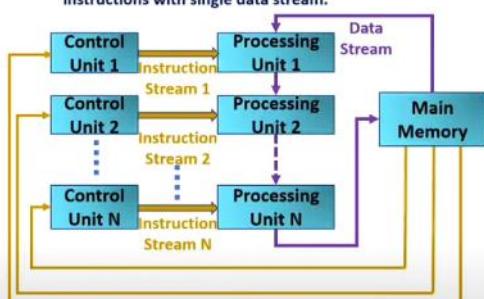


THIS WORK WILL NOT BE DONE IN O(1) USING SIMD JOKI SISD ME : O(N) ME HOTA !!

1 PROCESSING UNIT IS ESSENTIALLY 1 ALU !! AND USKE SAATH WALE REGISTERS and all !! { 1 core = 1 processing unit }

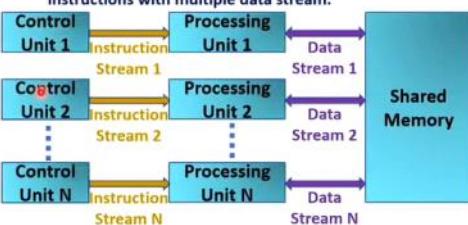
Flynn's Classifications

- MISD – Multiple Instruction Single Data stream
 - In MISD, Here multiple processor executes different instructions with single data stream.



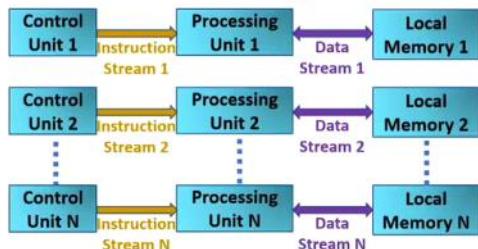
- This architecture is not implemented commercially.
- Example : Lets have same data Δ to be used in multiple instructions.
 - $A = \sin \Delta$
 - $B = \cos \Delta$
 - $C = \tan \Delta$

- MIMD – Multiple Instruction Multiple Data stream
 - In MIMD, Here multiple processor executes different instructions with multiple data stream.



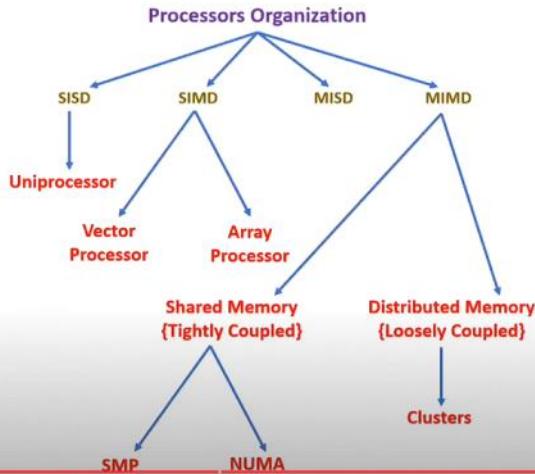
- Shared Memory in MIMD system is Symmetric multiprocessor system (SMP).
- Different processor takes instructions and data from common shared memory.
- With the use of buses, multiple processor executes program along with shared memory.

Flynn's Classifications



- Local Memory with MIMD is Non Uniform Memory Access NUMA organization.
- NUMA is costlier compared to SMP.
- Collection of Uniprocessor or SMP can be used to form cluster.
- Symmetric Multiprocessor SMP characteristics
 - There are two or more similar or comparable processor.
 - Memory Access time for all the processor will be same.
 - All the processors shares IO devices through same channel.
 - All the processors can perform same function. (That's why it is called symmetric multiprocessor system)

Computer Classifications by Flynn >



WHAT IS SISD BOTTLE NECK ?

A SISD bottleneck occurs when the performance of a computing system is limited by the sequential execution of instructions and the inability to exploit parallelism within the workload. Since SISD architectures do not allow for concurrent execution of instructions or processing of multiple data items simultaneously, they may struggle to efficiently handle tasks that can benefit from parallel processing.

In modern computing, where tasks often involve large volumes of data and complex computations, the limitations of SISD architectures can lead to performance bottlenecks, especially in applications that require high levels of parallelism. Tasks such as data processing, scientific simulations, and multimedia processing can benefit significantly from parallel execution, and SISD architectures may not be able to fully exploit the available computational resources.

Overlapped / Interleaved Execution of Instructions to achieve approximately / nearly 1 cycle / instruction on an avg is called Pipelining !!

→ 1 instruction can be divided into 4 phases of stages

- i) instruction fetch → memory to register
- ii) Read/decode → memory → register.
- iii) execution → ALU
- iv) write back → reg → memory

measure the performance of pipelines -

① speedup

② efficiency / utilization

→ no of stages in an instruction.

depth of pipeline → no of ops that can be performed in 1 cycle.

$$CPI = \frac{n + \text{depth} - 1}{n} \Rightarrow \frac{\text{total no of CPU cycles}}{\text{total no of instructions.}}$$

efficiency → $\frac{\text{no of boxes utilised}}{\text{Total no of pipelines.}}$

speed up → $\frac{\text{Time to execute without pipelining}}{\text{Time with pipelining}}$

	1	2	3	4	5	6	7	8
IF	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇	I ₈
ID	Y	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇
EX	Y	Y	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆
WB	Y	V	X	I ₁	I ₂	I ₃	I ₄	I ₅

CPI = 1
cycle time → T → pipe width time → 7.

$$\text{EFFICIENCY} = (8 * 4 - 6) / (8 * 4)$$

→ in general.

max efficiency → speed up.

no of stages / depth.

efficiency < 1

speedup < m.

$$\text{speedup} = \frac{D}{CPI} \rightarrow \text{depth}.$$

$T_{op} \rightarrow \text{Time} \Rightarrow 1 \times \text{depth} + (n-1) \times 1$
without pipeline.

T_{wp} Time with pipeline $\Rightarrow n \times m$.
 $m \rightarrow \text{depth}$, $n \rightarrow \text{no of instructions}$

$$\text{more speedup} \rightarrow \frac{T_{wp}}{T_{op}} \Rightarrow \frac{n \times m}{m + n - 1}$$

$$\rightarrow T_{op} = nk t_c$$

$$T_p = (n+k-1)t_c$$

$$\rightarrow \text{Speed up } S = \frac{T_{wp}}{T_p} = \frac{nk t_c}{(n+k-1)t_c} = \left\lceil \frac{nk}{(n+k-1)} \right\rceil$$

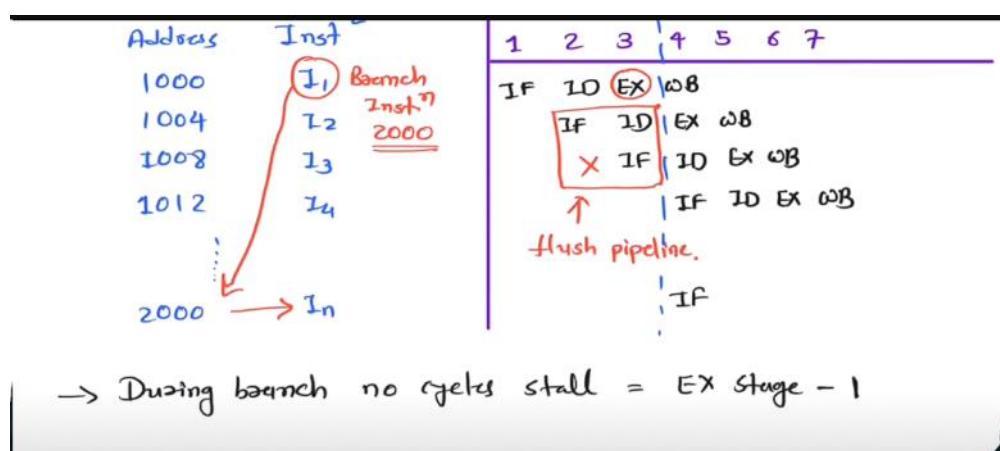
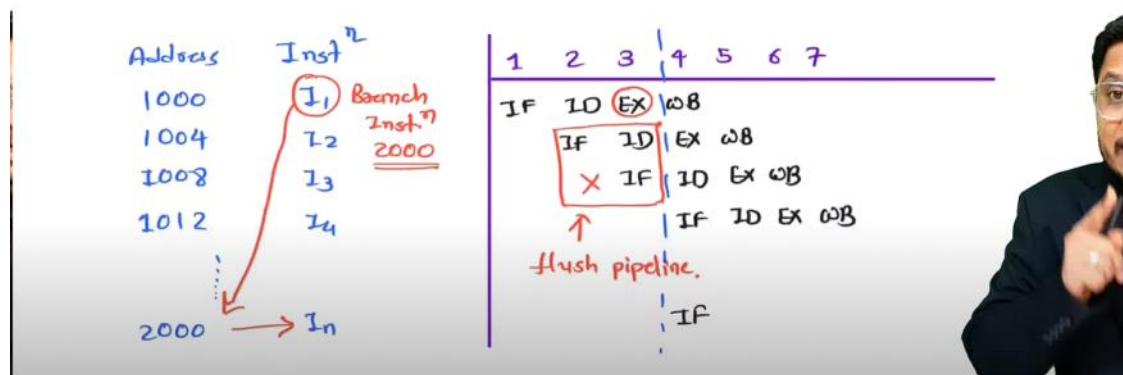
$$\rightarrow \text{Throughput} = \frac{n}{(n+k-1)t_c} = \left(\frac{1}{t_c} \right) \leftarrow \text{ideal}$$

THROUGHPUT = NO OF INSTRUCTIONS EXECUTED PER SEC !!

Do it only when no of instructions are not given { assume them to be infinite } and thereby get the throughput !!

CONTROL HAZARD :

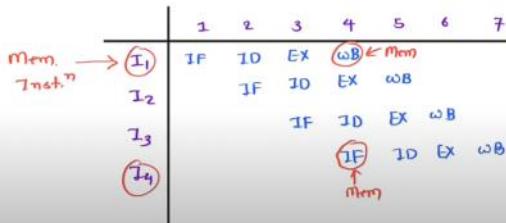
→ Let's have four stage pipeline : IF, ID, EX, WB



→ During branch no cycles stall = EX stage - 1

→ Let's take four stage pipeline :

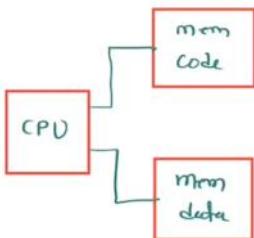
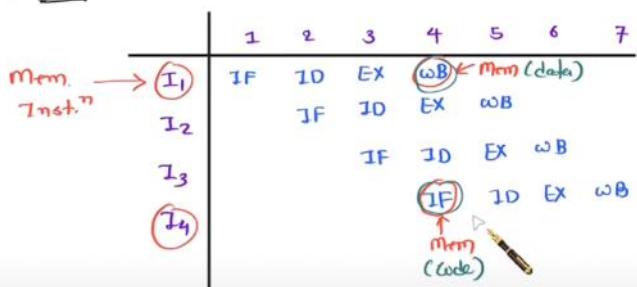
1. Instruction Fetch (IF)
2. Instruction decode + Operand decode (ID)
3. Execute Instruction (EX)
4. Writeback (WB).



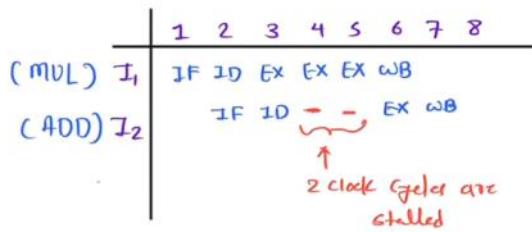
3:59 / 8:27 • 1 Example of Structural Hazards in Pipelining >

Structural Hazards in Pipelining

→ Solⁿ - Harvard Architecture.



Another Eg Of Structural Hazard :



Eg of Stalls in PipeLine Due to Time Differnce in execution of Tasks :

Q Consider 5 instruction with following clock requirement?

	F	D	E	WB
I ₁	1	2	1	1
I ₂	1	2	2	1
I ₃	2	1	3	2
I ₄	1	3	2	1
I ₅	1	2	1	2

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
I ₁	F	D	D	E	WB													
I ₂	F	X	D	D	E	E	WB											
I ₃	F	F	X	D	X	E	E	E	WB	WB								
I ₄	X	F	X	D	D	D	X	E	E	E	WB							
I ₅			F	X	X	X	D	D	X	E	WB	WB						

- We understand that different stages in a pipeline may have different delays, it also depends on the type of instruction that how much time a particular stage will take for a specific instruction.
- If we increase the time of a clock to the time taken by the slowest stage of pipeline, then each instruction takes one clock then with pipe-line processing long run, we achieve
 - CPI(Clock Per Instruction) = 1

Q Consider the following processors (ns stands for nanoseconds). Assume that the pipeline registers have zero latency. (Gate-2014) (2 Marks)

P₁: Four-stage pipeline with stage latencies 1 ns, 2 ns, 2 ns, 1 ns. $\rightarrow 2 \text{ ns} = \frac{1}{2 \text{ ns}} = 0.5 \text{ GHz}$

P₂: Four-stage pipeline with stage latencies 1 ns, 1.5 ns, 1.5 ns, 1.5 ns. $= 1.5 \text{ ns} = 0.667 \text{ GHz}$

P₃: Five-stage pipeline with stage latencies 0.5 ns, 1 ns, 1 ns, 0.6 ns, 1 ns. $\rightarrow 1 \text{ ns} = 1 \text{ GHz}$

P₄: Five-stage pipeline with stage latencies 0.5 ns, 0.5 ns, 1 ns, 1 ns, 1.1 ns. $\rightarrow 1 \text{ ns} = 1 \text{ GHz}$

Which processor has the highest peak clock frequency?

- (A) P₁
- (B) P₂
- (C) P₃
- (D) P₄

STALLS IN PIPELINING MEANING :

L What are stalls in pipelining ?

Show drafts ▾ 🔍

◆ In the world of pipelined processors, stalls refer to temporary pauses in the execution of instructions. These pauses occur to resolve hazards, which are potential conflicts or dependencies that could lead to incorrect results if instructions are executed out of order.

Data Hazards in Pipelining

I₁: ADD R₁, R₂, R₃ +

I₂: ADD R₁, R₁, R₅ +

	1	2	3	4	5	6	7	8	9
I ₁	IF	2D	EX	m	wB	Result of R ₁			
I ₂	IF	2D	EX	m	wB	↑ False result in R ₄			



→ Operand Forwarding is the soln.

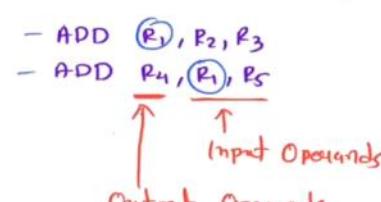
	1	2	3	4	5	6	7	8	9
I ₁	IF	2D	EX	m	wB	Result of R ₁			
I ₂	IF	2D	EX	m	wB	↑ Correct result.			

SO BASICALLY IN BTW EVERY STAGE THERE WILL BE A BUFFER REGISTER STORING THE VALUES OF

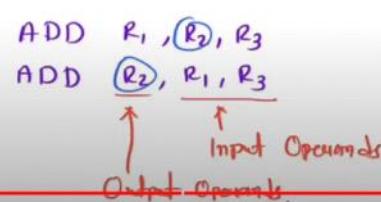
THE RESULTS THAT ARE NOT YET STORED INTO MEMORY BUT MIGHT BE NEEDED BY THE NEXT / UPCOMING INSTRUCTIONS AS INPUT AND TO ENSURE THAT THEY DON'T TAKE THE STALE VALUE THIS CONCEPT CAN BE USED !!

TYPES OF DATA DEPENDANCY / DATA HAZARDS :

→ RAW - Read After write (Actual / True dependency)

- ADD $\underline{R_1, R_2, R_3}$ $IO\{N_I\} \cap OO\{I^{\text{st}}_I\} \neq \phi$
- ADD $\underline{R_4, R_1, R_5}$

Input Operands
Output Operands.

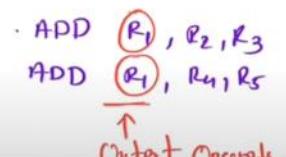
→ WAR - Write After Read (Anti dependency).

ADD $R_1, \underline{R_2, R_3}$ $IO\{I^{\text{st}}_I\} \cap OO\{N_I\} \neq \phi$
ADD $\underline{R_2, R_1, R_3}$

Input Operands
Output Operands.

11:59 / 14:09 • Write After Write Data Hazard >

Output Operands.

→ WAW - write After write (Output dependency)

- ADD $\underline{R_1, R_2, R_3}$ $OO\{I^{\text{st}}_I\} \cap OO\{N_T\} \neq \phi$
- ADD $\underline{R_1, R_4, R_5}$

Output Operands.

11:57 / 14:09 • Write After Write Data Hazard >

2 INSTRUCTIONS READING THE SAME VALUE WO PROBLEM OFC NAHI KAREGA UNLESS AND UNTILL SAME CLOCK CYCLE ME MEMORY ACCESS NA HO { STRUCTURAL HAZARD WALI BAATE }

$$\text{EFFICIENCY} = \text{SPEEDUP} / D = N / (N+D-1)$$

Examples on Pipelining in COA

- GATE CS – Consider a 5 stage pipeline with cycle time of 5ns. Calculate the execution time of 100 instructions and Speed up due to pipeline. Also find the utilization.

$$\rightarrow K = 5$$

$$t_c = 5 \text{ ns}$$

$$n = 100$$

$$\rightarrow T_p = (n+K-1)t_c = (100+5-1)5 = 520 \text{ nsec}$$

$$\rightarrow S = \frac{nK}{n+K-1} = \frac{100 \times 5}{100+5-1} = 4.8077$$

$$\rightarrow U = \frac{n}{n+K-1} = \frac{100}{100+5-1} = 0.9415$$



2/51 / 12/06 - 1 Examples on Pipelining

- GATE CS – Consider a 5 stage pipeline. Delay of each stage is given as per 10ns, 16ns, 12ns, 11ns & 14ns, respectively. Calculate the execution time of 100 instructions and Speed up due to pipeline.

$$\rightarrow K = 5$$

$$t_c = 16 \text{ nsec}$$

$$n = 100$$

$$\rightarrow T_p = (n+K-1)t_c = (100+5-1)16 = 1664 \text{ nsec} = 1.664 \text{ usec}$$

$$\rightarrow S = \frac{T_{op}}{T_p} = \frac{100(10+16+12+11+14)}{1664} = 3.786$$



Vvgood Question !!

Here it is

time to

execute

all stages without pipeline

- We understand that different stages in a pipeline may have different delays, it also depends on the type of instruction that how much time a particular stage will take for a specific instruction.
- If we increase the time of a clock to the time taken by the slowest stage of pipeline, then each instruction takes one clock then with pipe-line processing long run, we achieve
 - CPI(Clock Per Instruction) = 1



THIS IS THE CONCEPT USED IN ABOVE QUESTION !!

- GATE 2011 CS – Consider an instruction pipeline with four stages (S1, S2, S3 & S4) each with combinational circuit only. The pipeline registers are required between each stage and at the end of the last stage. Delay for the stages and for the pipeline registers are as given in the figure.



- What is the approximate speed up of the pipeline in steady state under ideal conditions when compared to the corresponding non pipeline implementation?

$$S = \frac{T_{wp}}{T_p} = \left\lceil \frac{5 + 6 + 11 + 8}{11 + 1} \right\rceil = 2.5$$



Super IMP CONCEPT :

TO REMOVE THE DATA HAZARD WALI PROBLEM WE NEED TO INSTALL THESE BUFFER / DELAY REGISTERS { TO DO THE OPERATOR FORWARDING !! }

ALSO : IN PIPELINE WITH VARIABLE STAGE TIME : TIME OF SLOWEST ONE IS TAKEN AS THE CLOCK CYCLE AND THEREFORE : TC BECOMES = (11+1) NS { VVIMP !! }

NOW : TP = TC*(N+D-1) AND TWP { WITHOUT PIPELINE } = TOTAL TIME TO EXECUTE ALL THE INSTRUCTIONS = (SUM OF DELAYS OF ALL STAGES) * (N)

HERE : SUM OF DELAYS OF ALL STAGES IS ONLY TAKEN AND NOT THE DELAY REGISTERS FOR NO PIPELINE CASE BCZ : APKA : DELAY REGISTERS STRUCTURAL HAZARDS KO HATANE KE LIYE HOTE HAI JOKI SIRF : PIPELINING ME HONGE !!

HERE : N IS NOT GIVEN : SO IN ALL THOSE CASES : N->INFINITY AND BY LIMITS : SPEEDUP = TC/ (SUM OF DEALYS OF ALL STAGES)

Examples on Pipelining in COA

- GATE CS – Consider a pipeline having 5 stages with duration 10ns, 30ns, 45ns, 80ns and 35ns. If the buffer delay is 20ns, calculate the speed up of pipelined processor.

$$S = \frac{T_{wp}}{T_p} = \left\lceil \frac{10 + 30 + 45 + 80 + 35}{80 + 20} \right\rceil = 2$$



- GATE CS – Assume we have two pipelines P1 and P2, respectively. P1 has 6 stages, having execution time of 12ns, 14ns, 19ns, 20ns, 22ns and 25ns. P2 has 4 stages each having execution time of 10ns. Calculate the time that can be saved while using P2 pipeline over P1 pipeline, If 2000 instructions are executed.

$$\rightarrow P_1, K_1 = 6, t_{c_1} = 25 \text{ nsec}$$

$$\rightarrow P_2, K_2 = 4, t_{c_2} = 10 \text{ nsec}$$

$$\rightarrow n = 2000$$

$$\rightarrow T_{P_1} = (n + K_1 - 1) t_{c_1} = (2000 + 6 - 1) 25 = 50125 \text{ nsec}$$

$$\rightarrow T_{P_2} = (n + K_2 - 1) t_{c_2} = (2000 + 4 - 1) 10 = 20030 \text{ nsec}$$

$$\rightarrow \Delta T = T_{P_1} - T_{P_2} = 50125 - 20030 = 30095 \text{ nsec}$$



- **GATE CS** – Assume a pipeline P which operates at 3GHz clock rate. It has a speed up factor of 10 and efficiency of 40%. Calculate the number of stages in the above pipeline.

$$\rightarrow f_{clock} = 3 \text{ GHz}$$

$$S = 10$$

$$\eta = 40\% = 0.4$$

$$\rightarrow S = K\eta \Rightarrow K = \frac{10}{0.4} = 25$$



Examples on Pipelining in COA

- **GATE 2015 CS** – Consider a non pipelined processor with a clock rate of 2.5GHz and average cycles per instruction of four. The same processor is upgraded to a pipelined processor with five stages; but due to internal pipeline delay, the clock speed is reduced to 2GHz. Assume that there are no stalls in the pipeline. The speed up achieved in this pipelined processor is

$$\rightarrow f_{cl} = 2.5 \text{ GHz}, CPI = 4$$

$$\rightarrow K = 5, f_{cl} = 2 \text{ GHz}$$

$$\rightarrow S = \frac{T_{avg}}{T_p} = \frac{4/2.5 \times 10^9}{1/2 \times 10^9} = \frac{8}{2.5} = 3.2$$



AGAIN : bcz : n is not given : n and n+k-1 wali terms gayab hogayi for n->infinity !!

mode pipeline in computer architecture

□ **GATE CS** – The stage delays in a 4 stage pipeline are 5ns, 6ns, 4ns and 5ns. The second stage is replaced with a functionally equivalent design involving two stages with respective delays 4ns and 5ns. The throughput increase of the pipeline is %

$\rightarrow P-1 \{ 5\text{ns}, 6\text{ns}, 4\text{ns}, 5\text{ns} \} \quad \rightarrow P-2 \{ 5\text{ns}, 4\text{ns}, 5\text{ns}, 4\text{ns}, 5\text{ns} \}$

$\rightarrow \text{Throughput} = \frac{n}{(n+k-1)t_c} = \frac{1}{t_c}$ t actually

$\rightarrow t_{c1} = 6\text{ns}, t_{c2} = 5\text{ns}$

$\rightarrow TP_1 = \frac{1}{t_{c1}} = \frac{1}{6\text{ns}}$

$\rightarrow TP_2 = \frac{1}{t_{c2}} = \frac{1}{5\text{ns}}$

$\rightarrow \% \uparrow_a = \frac{TP_2 - TP_1}{TP_1} \times 100$

$= \frac{1/5 - 1/6}{1/6} \times 100$

$= \frac{6}{5} - 1 = 0.2 \times 100 = 20\%$

10:13 / 15/23 • 2 Example on Pipelining >

NUMERICALS ON BRANCH HAZARDS :

NO OF STALLS DURING BRANCH HAZARDS = (STAGE NO OF EXECUTION) -1

WHY ? BCZ : EXECUTION STAGE PE PATA CHALEGA KI AB TO FLUSH KARNA PADEGA PICHLE WALE STAGES ME PADE INSTRUCTIONS KO !! THAT IS UTNI STAGES ME PADE INSTRUCTIONS KO WAPIS SE RE EXECUTE KARNA PADEGA BAAD ME !!

Examples on Pipelining in COA

5 Examples on Pipelining

- GATE 2009 CS – Consider an instruction pipeline with five stages without any branch prediction: Fetch Instruction (FI), Decode Instruction (DI), Fetch Operand (FO), Execute Instruction (EI) and Write Operand (WO). The stage delay for FI, DI, FO, EI and WO are 5ns, 7ns, 10ns, 8ns and 6ns, respectively. There are intermediate storage buffer after each stage and the delay of each buffer is 1ns. A program consisting of 12 instructions I₁, I₂, I₃, ..., I₁₂ is executed in this pipeline processor. Instruction I₄ is only the branch instruction and its branch target is I₉. If the branch taken during the execution of this program, the time needed to complete the program is?

- $\frac{I_1, I_2, I_3, I_4, I_5, I_6, I_7}{n = 8}$
- $F1, P1, F0, E1, W0$
 \uparrow
 4th Stage.
- No of stalled cycles = $4 - 1 = 3$
- $T_p = t_p + t_s$
 $= (n+k-1)t_c + xt_c$
 $= (8+5-1)11 + 3 \times 11$
 $= 15 \times 11$
 $= 165 \text{ nsec}$



PROBLEM BASED ON STRUCTURAL HAZARD :

- GATE CS** – Consider the pipelined processor with the following four stages:

IF: Instruction Fetch

ID: Instruction Decode and Operand Fetch

EX: Execute

WB: Write Back

- The IF, ID and WB stages take one clock cycle each to complete the operation. The number of clock cycles for the EX stage depends on the instruction. The number ADD and SUB instructions need 1 clock cycle and MUL instruction needs 3 clock cycle in the EX stage. **Operand forwarding** is used in the pipelined processor. What is the number of clock cycles taken to complete the following sequence of instructions? = 8

~~ADD R2, R1, R0 ;R2 \leftarrow R1 + R0
MUL R4, R3, R2 ;R4 \leftarrow R3 x R2
SUB R6, R5, R4 ;R6 \leftarrow R5 + R4~~

J₁- IF ID EX WB
J₂- IF ID EX EX EX WB
J₃- IF ID - - EX WB


2 cycles stalled.

9:52 / 12:26 • 2 Example of Pipelining >

Structural Hazard



THAT IS WHY DATA HAZARD NOT TAKEN INTO PICTURE !!

Digitized by srujanika@gmail.com

J ₁	(3)	MUL R2, R0, R1	;R2 \leftarrow R0 x R1
J ₂	(6)	DIV R5, R3, R4	;R5 \leftarrow R3 / R4
J ₃	(1)	ADD R2, R5, R2	;R2 \leftarrow R5 + R2
J ₄	(1)	SUB R5, R2, R6	;R5 \leftarrow R2 - R6

~~I₁: IF ID OF PO PO PO WO~~
~~I₂: IF ID OF -- PO PO PO PO PO PD WO WO~~
~~I₃: IF ID OF - - - - - PO WO~~
~~I₄: IF ID OF - - - - - - - - - PO WO WO~~

$$\begin{aligned}\rightarrow \text{Cycles} &= \text{Normal Cycles} + \text{Hazardous Cycles} \\ &= (n+k-1) + (5+2) \\ &= (4+5-1) + 5+\end{aligned}$$

Short cut for such ques !! { not recommended for paper }

□ What is the number of cycles needed to execute the following sequence of instruction without data forwarding in above question?

1. $MUL R2, R0, R5$ (3) ; $R2 \leftarrow R0 \times R1$
 2. $DIV R5, R3, R4$ (6) ; $R5 \leftarrow R3 / R4$
 3. $ADD R2, R5, R2$ (1) ; $R2 \leftarrow R5 + R2$
 4. $SUB R5, R2, R6$ (1) ; $R5 \leftarrow R2 - R6$

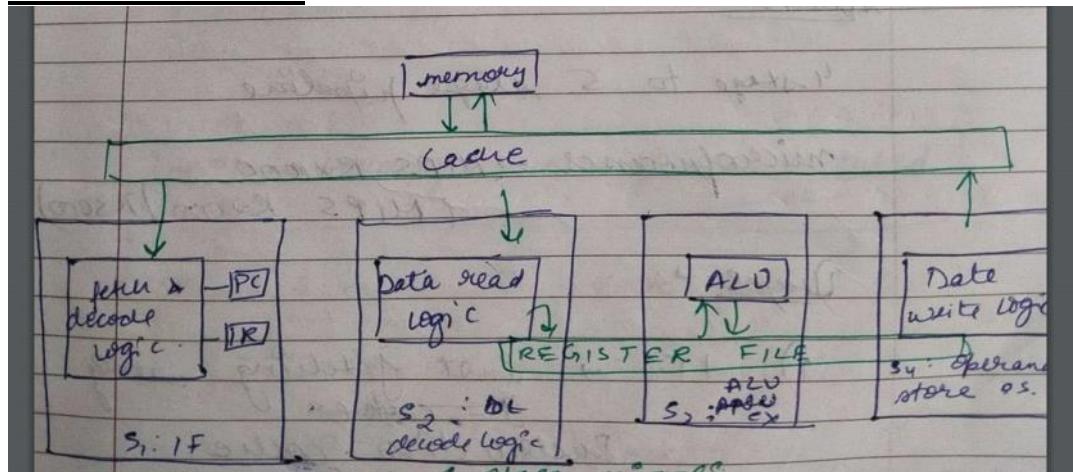
$\begin{matrix} \text{Output} & \text{Input} \\ \text{Operands} & \text{Operands} \end{matrix}$

I₁: IF ID OF PO (PO) (PO) WB
 I₂: IF ID OF -- -- PO PO PO PO PO (WB) WB
 I₃: IF ID -- -- -- -- -- -- -- -- -- -- -- -- -- OF PO WB
 I₄: IF ID -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- OF PO WB

IF THERE IS A DATA DEPENDANCY WE CAN TAKE THE VALUE ONLY AFTER WRITE BACK HAS HAPPENED !!! UNLIKE CONTROL DEPENDANCY JAHAN EX PE HI PATA CHAL JATA THA KI KINKO FLUSH KARNA HAI !!

NOTE :

IN 4 STAGE PIPELINE :



**ANOTHER QUES BASED ON STRUCTURAL HAZARD :
HERE THAT SHORTCUT IS USED !!**

BASICALLY : EXTRA CYCLES DUE TO HAZARD = (EXTRA CLOCK CYCLES { I.E CYCLES TAKEN-1})*(NO OF INSTRUCTIONS JINKE SAATH YE HAI !! }

Examples on Pipelining Hazards in COA

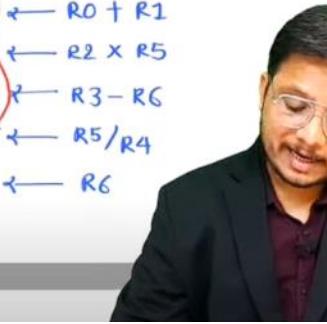
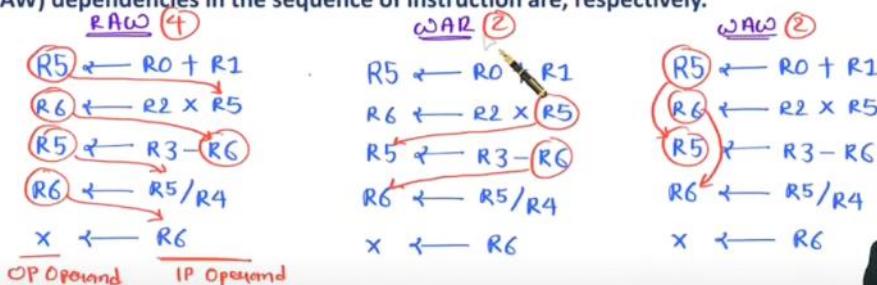
GATE 2018 CS – The Instruction pipeline of a RISC processor has the following stages: Instruction (IF), Instruction Decode (ID), Operand Fetch (OF), Perform Operation (PO) and Writeback (WB). The IF, ID, OF and WB stages take 1 clock cycle each for every instruction. Consider a sequence of 100 instructions. In the PO stage, 40 instructions take 3 clock cycles each, 35 instructions take 2 clock cycles each and the remaining 25 instructions take 1 clock cycle each. Assume that there are no data hazards and no control hazards. The number of clock cycles required for completion of execution of the sequence of instruction is?!

$$\begin{aligned} \rightarrow \text{No of cycles} &= \text{Normal pipeline cycles} + \text{cycles due to structural hazard} \\ &= (n+k-1) + 35 \times 1 + 40 \times 2 \\ &= (100+5-1) + 35 + 80 \\ &= 219 \end{aligned}$$



- The number of Read After Write (RAW) dependencies, Write After Read (WAR) dependencies and Write After Write (WAW) dependencies in the sequence of instruction are, respectively.

- A. 2, 2, 4
B. 3, 2, 3
C. 4, 2, 2
D. 3, 3, 2



NOTE :

APKA : RAW ME : APKA : 1ST WALE OUTPUT KO AAP DEKHOGE INPUT TO NAHI : FOR ALL THE : INSTRUCTIONS AFTER IT BUT BUT BUT : INTERESTING CATCH IS THAT : AGAR : WAHI SAME OUTPUT AA GAYA KISI INSTRUCTION ME : TO USKE BAAD SE : NAHI DEKHOGE AAP !! WHY ? BCZ : USKE CASE ME : AAGE ANE WALA OUTPUT JO HOGA : WO CREATE KAREGA RAW WALI DEPENDANCY !!!

CPI & Speed Up in Pipelining with Hazard

→ Without Hazard.

$$CPI = \frac{\text{Total Cycles}}{\text{Total Inst.}^n} = \frac{(n+k-1)}{n} = (1) \leftarrow \text{ideal.}$$



→ With Hazard { ∞ cycles due to hazard }.

$$CPI = \frac{(n+k-1) + x}{n} = \left[1 + \frac{x}{n} \right] \leftarrow \text{ideal.}$$

→ without Hazard.

$$S = \frac{T_{op}}{T_p} = \frac{T_{op}}{CPI T_p}$$

With hazard

IMP NUMERICAL :

GATE CS – Consider a 5 stage pipeline which is executing a program of 100 instructions. Among all instructions 10 instructions cause 3 stall cycles each.

- A. Calculate CPI of Pipeline?
- B. If Pipeline cycle time is 3ns then what is average instruction execution time?
- C. Calculate CPI of pipeline in ideal conditions with hazards?
- D. If pipeline cycle time is 3ns then what is average instruction execution time in ideal conditions?

$$\rightarrow K = 5$$

$$n = 100$$

$$x = 10 \times 3 = 30$$

$$1) CPI = \frac{(n+k-1)+x}{n} = \frac{100+5-1+30}{100} = 1.34$$

$$2) t_{inst} = CPI \times t_c = 1.34 \times 3 = 4.02 \text{ nsec}$$

$$3) CPI_{WAL} = 1 + \frac{x}{n} = 1 + \frac{30}{100} = 1.3$$

$$4) t_{inst_WAL} = CPI_{WAL} \cdot t_c = 1.3 \times 3 = 3.7 \text{ nsec.}$$



$= [\text{Execution Stages Numbers} - 1]$
per Branch Inst.

IN CASE OF CONTROL HAZARD !! { FOR STRUCTURAL HAZARD : APKO SUMMISION OF (EXTRA CYCLES) WALA FORMULA PEHLE HI BATA DIYA THA }

Examples on Pipelining Hazards in COA

GATE 2014 CS – Consider a 6 stage instruction pipeline, where all stages are perfectly balanced. Assume that there is no cycle time overhead of pipelining. When an application is executing on this 6 stage pipeline, the speedup achieved with respect to non pipelined execution if 25% of the instructions incur 2 pipeline stall cycles is?

$$\rightarrow K = 6$$

$$\rightarrow CPI = 1 + \frac{x}{n}$$

$$= 1 + 0.25 \times 2$$

$$= 1.5$$

$$\rightarrow S = \frac{T_{nop}}{CPI \cdot T_p} = \frac{6t_c}{1.5 \cdot t_c} = 4$$

A WHOLE LOT OF CONCEPTS APPLIED HERE !!

1. **CPI FOR HAZARD SITUATIONS { BCZ N NOT GIVEN TAKEN IDEAL CASE } X=25% OF N * 2**

2. **Speedup = TwithoutPipeline / (TwithPipeline * cpi)**

Isme n not given so : n / n-1+k wali term 1 ban jaati hai !!

```

types of pipeline:
1. uniform delay pipeline: every stage (instruction fetch, decode, execute, write back) will take equal time to complete one operation
cycle time (tp) = stage delay

cycle time (tp) = stage delay + buffer delay (when stalls are added)

2. non uniform delay pipeline
difference stages will take different units of time

cycle time (tp) = maximum stage delay

with buffers (they may be added for other reasons like for removing hazards):
cycle time (tp) = max( stage delay + buffer delay)

```

SUPER IMP CONCEPT { ALREADY USED IN ALL THE : PREVIOUS NUMERICALS BUT FORMALLY DOING IT NOW }

1. MIMD (Multiple Instruction, Multiple Data):
 - Example: Cluster of networked computers working together on a computational task, where each computer operates independently and can execute different instructions on different sets of data concurrently. This architecture is commonly seen in distributed computing environments or in parallel computing clusters.
2. SIMD (Single Instruction, Multiple Data):
 - Example: Graphics Processing Units (GPUs) where a single instruction is applied to multiple data points simultaneously. For instance, when rendering an image, a single instruction could be used to apply a specific transformation to every pixel in parallel.
3. SISD (Single Instruction, Single Data):
 - Example: Traditional von Neumann architecture CPU, where a single instruction operates on a single piece of data at a time. This is the classical model of computation where instructions are executed sequentially.
4. MISD (Multiple Instruction, Single Data):
 - Example: Fault-tolerant systems or systems with redundancy where multiple instructions are applied to the same data stream to detect errors or perform concurrent checks. This architecture is less common in practical systems but can be found in specialized applications such as certain types of signal processing or fault-tolerant computing systems.

PARALLELISM :

CORE CONCEPT :

parallelism in a single processor:

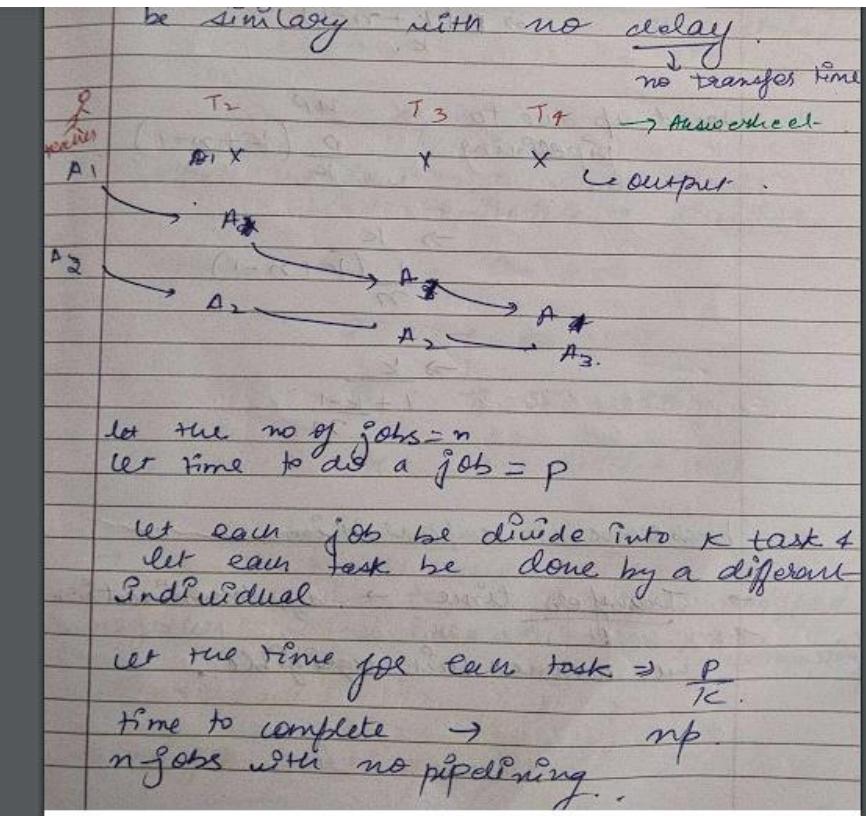
- pipelining**
- increase # cores**
- increase functional units**
- overlap opration of different units of computer**
- increase speed of ALU by exploiting data/temporal ||ism**

How to make computers fast

#cpus > 1 for true ||ism

simultaneous exec --> ||ism

interleaved exec --> only 1 cpu, cpu don't wait for anything --> pipelining



with a pipeline organisation of k individuals
 $\Rightarrow p + (n-1)p$

DERIVATION OF THIS FORMULA { IT IS THE TIME TO COMPLETE N JOBS WITH PIPELINING } = N * TC * CPI
CPI = N+K-1 / N , TC=TIME OF THE SLOWEST STAGE = P/K
AND THEREFORE ANS =((N+K-1)*P)/K !!

WHAT IS TEMPORAL PARALLELISM :

Temporal parallelism, also known as pipelining, refers to the concept of **overlapping** the execution of **different parts of a single task** across multiple processing units. It's like an assembly line, where each unit performs a specific sub-task on different pieces of data simultaneously.

Here's how it works:

1. **Task breakdown:** A complex task is divided into smaller, independent sub-tasks.
2. **Dedicated units:** Each sub-task has its own dedicated processing unit.
3. **Overlap the flow:** While one unit processes data, the next unit in the pipeline starts working on the next piece of data, and so on.

This approach enables faster completion of the overall task compared to doing it sequentially because:

- **No idle time:** Processing units are constantly utilized, leading to improved efficiency.
- **Faster throughput:** Multiple pieces of data are processed simultaneously, increasing the overall output.

REMEMBER : SPEEDUP = D/CPI !!

Problems in Temporal Parallelism :

→ Transfer time → synchronisation.
we assumed this negligible.
④ fault tolerance (all processors are equal) one does fault can't be resolved/fixed.
⑤ scalability not possible
⑥ inter task comp communication
⑦ bubbles in pipeline (one stops all other stops)

Data Parallelism

- Let the time to distribute the jobs to k individuals be kq . Observe that this time is proportional to the number of individuals.
- The time to complete n jobs by a single individual = np
- The time to complete n jobs by k individuals = $kq + np/k$
- Speedup due to parallel processing =

$$\frac{np}{kq + \frac{np}{k}} = \frac{knp}{k^2q + np} = \frac{k}{1 + (k^2q/np)}$$

ChatGPT

Data parallelism is a parallel computing paradigm where the same operation is applied simultaneously to multiple pieces of data. In other words, the computation is performed concurrently on different subsets or partitions of a larger dataset. This approach contrasts with task parallelism, where different tasks or operations are executed concurrently.



- Data parallelism is a form of **parallelization across multiple processors** in parallel computing environments. It focuses on **distributing the data itself across different nodes**, which then operate on that data **concurrently**. This essentially means performing the **same operation on different pieces of data at the same time**.

SIMD (Single Instruction, Multiple Data) computers are **inherently designed for data parallelism**. They achieve parallelism by executing the **same instruction on multiple data elements simultaneously**. This aligns perfectly with the core concept of data parallelism, where different processors work on **independent pieces of data** concurrently.

Data Parallelism

- The main **advantages** of this method are:
 - There is **no synchronization** required between teachers. Each teacher can correct papers independently at his own pace.
 - The problem of **bubbles is absent**. If a question is unanswered in a paper it only reduces the time to correct that paper.
 - This method is **more fault tolerant**. One of the teachers can take a coffee break without affecting the work of other teachers.
 - There is **no communication required** between teachers as each teacher works independently. Thus, there is no inter-task communication delay.

Data Parallelism

- The main **disadvantages** of this method are:
 - The assignment of jobs to each teacher is pre-decided. This is called a static assignment. Thus, if an individual is slow then the completion time of the total job will be slowed down.
 - We must be able to divide the set of jobs into subsets of mutually independent jobs. Each subset should take the same time to complete.
 - Each individual must be capable of correcting answers to all questions. This is to be contrasted with pipelining in which each individual specialized in correcting the answer to only one question.
 - The time taken to divide a set of jobs into equal subsets of jobs should be small. Further, the number of subsets should be small as compared to the number of jobs.

Other Parallelisms

• Data Parallelism with Dynamic Assignment

- Here a head examiner gives one answer paper to each teacher and keeps the rest with him. All teachers simultaneously correct the paper given to them. A teacher who completes correction goes to the head examiner for another paper which is given to him for correction. If a second teacher completes correction at the same time, then he queues up in front of the head examiner and waits for his turn to get an answer paper. The procedure is repeated till all the answer papers are corrected.

• Data Parallelism with Quasi-dynamic Scheduling

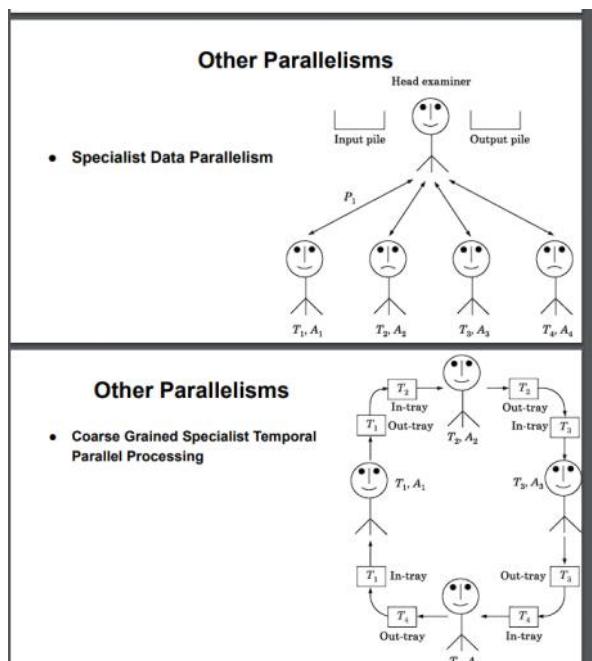
- giving each teacher unequal sets of answer papers to correct.

Dynamic Assignment in Data Parallelism:

- The assignment of tasks or operations to processors happens dynamically during runtime, often based on factors like:
 - **Workload:** Processors can receive tasks based on their current load or availability.
 - **Data characteristics:** Tasks can be assigned based on specific data features within a chunk.
 - **Algorithm requirements:** Different algorithms might dictate dynamic task assignment strategies.
- This allows for better workload balancing and potentially improves efficiency on tasks with uneven data distribution or workload changes.

quasi dynamic ->same as data ||ism with dynamic assignment but assign >1 task at a time

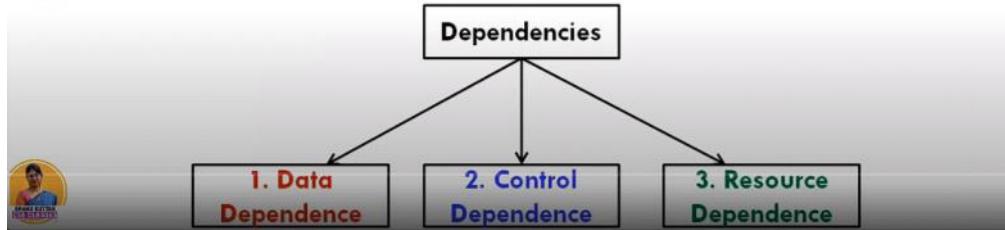
DOUBT :



DETECTING PARALLELISM :

Conditions of Parallelism

- The aim of parallel programmer is to exploit parallelism in a given program.
- To execute several program segments in parallel it is necessary that each segment is **independent** of other segments
- So, before executing parallelism, all the **conditions of parallelism** between the segments must be analyzed
- **Three types of dependencies** exist between the program segments:



Dependence graph

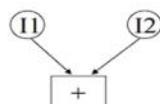
- A **dependence graph** is used to describe the **dependence relationships** between the statements.
 - **Nodes** corresponds to the **program statements (instructions)**, and
 - **Directed edges** with different **labels** shows the **ordered relation among the statements**.

3. Resource Dependence

- **Resource dependence** is concerned with conflicts in using shared resources, such as integer and floating point units (ALUs), registers and memory areas among parallel events, etc.
 - ALU conflicts are called **ALU dependence**.
 - Memory (storage) conflicts are called **Storage dependence**.
- **Example:**

I1 and I2 both are accessing addition unit, so ALU dependent

I1: $A = B + C$
I2: $G = D + H$



PROBLEMS ON DATA DEPENDANCE AND RESOURCE DEPENDANCE :

- a) S1: $A = B + D$
 S2: $C = A * 3$
 S3: $A = A + C$
 S4: $E = A / 2$

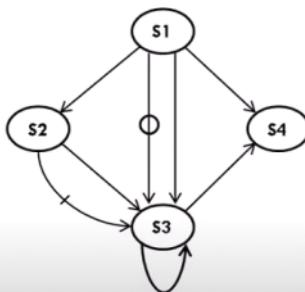


Data Dependencies:

- S1 to S2: S2 is flow dependent on S1
- S1 to S3: S3 is flow and output dependent on S1
- S1 to S4: S4 is flow dependent on S1

- S2 to S3: S3 is flow and anti dependent on S2
- S2 to S4: No dependency

- S3 to S3: S3 is Flow dependent on itself
- S3 to S4: S4 is Flow dependent on S3



Dependence Graph

- b) S1: $X = \sin(Y)$
 S2: $Z = X + W$
 S3: $Y = -2.5 * W$
 S4: $X = \cos(Z)$

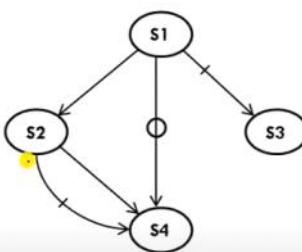


Data Dependencies:

- S1 to S2: S2 is flow dependent on S1
- S1 to S3: S3 is anti dependent on S1
- S1 to S4: S4 is output dependent on S1

- S2 to S3: No dependency
- S2 to S4: S3 is flow and anti dependent on S2

- S3 to S4: No dependency



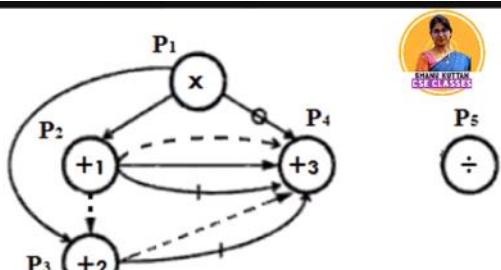
Dependence Graph

DO MENTION IN PAPER THAT ARROW AND A CUT { IS FOR ANTI DEPENDANCE }, GOLA WALA IS FOR OUTPUT DEPENDANCE AND REST NORMAL WALA IS FOR : FLOW DEPENDANCE !!

- P1: $C = D * E$
 P2: $M = G + C$
 P3: $A = B + C$
 P4: $C = L + M$
 P5: $F = G / E$

Solution:

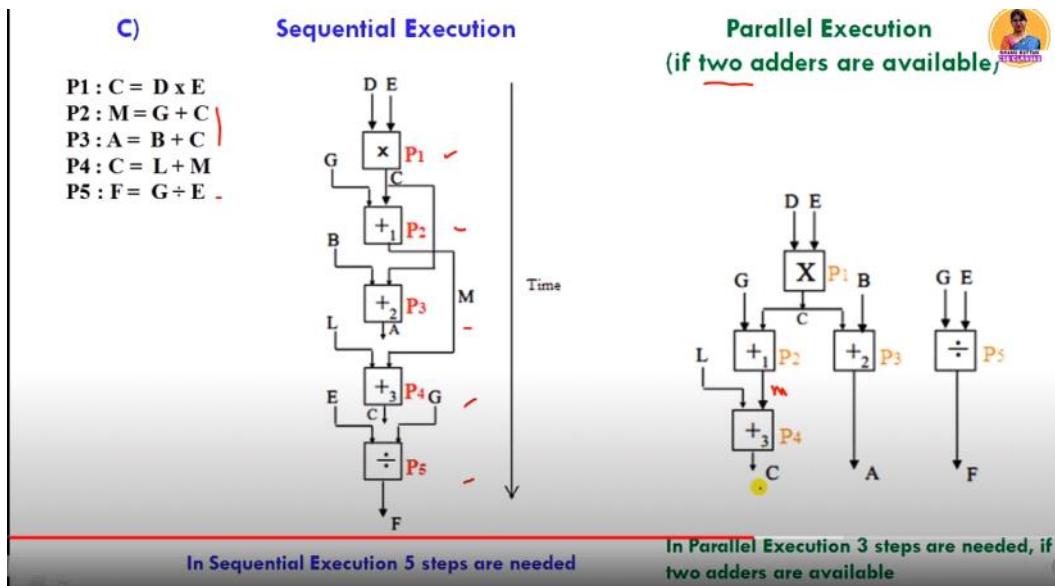
- **Data dependencies:**
 - P1 to P2: P2 is flow dependent on P1
 - P1 to P3: P3 is flow dependent on P1
 - P1 to P4: P4 is output dependent on P1
 - P1 to P5: No dependency
 - P2 to P3: No dependency
 - P2 to P4: P4 is flow dependent & anti dependent on P2
 - P2 to P5: No dependency
 - P3 to P4: P4 is anti dependent on P3
 - P3 to P5: No dependency
 - P4 to P5: No dependency



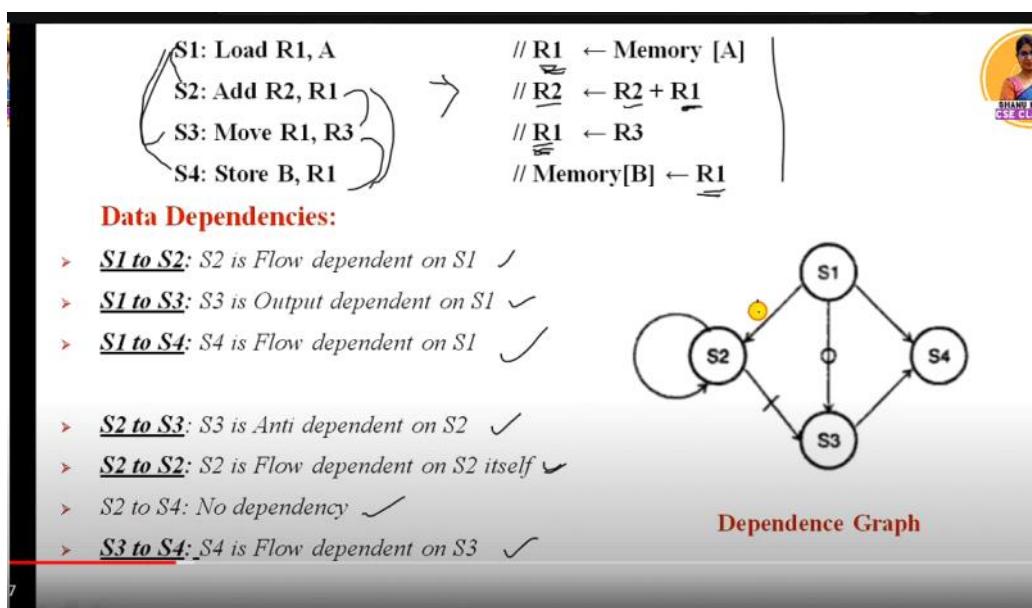
Dependence Graph:
Data dependence (Solid Lines)
Resource dependence (Dashed Lines)

Resource dependencies:

- P2 to P3: P3 is resource dependent on P2 (+)
- P2 to P4: P2 is resource dependent on P4 (+)
- P3 to P4: P4 is resource dependent on P3 (+)



{ SIMILAR OPERATION WALI INSTRUCTIONS WILL BE : RESOURCE DEPENDANT ! :THUMB RULE }



Question 2:

Analyze the data dependencies in following statements:

S1: LOAD R1, M(100)
S2: MOV R2, R1
S3: INC R1
S4: ADD R2, R1
S5: STORE M(100), R1

- Draw dependency graph to show all the dependencies
- Are there any resource dependencies if only one copy of each functional unit is available

Solution:

```

S1: LOAD R1, M(100)
S2: MOV R2, R1
S3: INCR1
S4: ADD R2, R1
S5: STORE M(100), R1

```

// $R_1 \leftarrow \text{MEM}(100)$
// $R_2 \leftarrow R_1$
// $R_1 \leftarrow R_1 + 1$
// $R_2 \leftarrow R_2 + R_1$
// $\text{MEM}(100) \leftarrow R_1$

Data Dependencies:

- S1 to S2: S2 is flow dependent on S1 ✓
- S1 to S3: S3 is flow and output dependent on S1 ✓
- S1 to S4: S4 is flow dependent on S1 ✓
- S1 to S5: S5 is flow dependent on S1 ✓
- S2 to S3: S3 is anti dependent on S2 ✓
- S2 to S4: S4 is flow and output dependent on S2 ✓
- S2 to S5: No dependency ✓
- S3 to S4: S4 is flow dependent on S3 ✓
- S3 to S3: S3 is flow dependent on S3 itself ✓
- S4 to S4: S4 is flow dependent on S4 itself ✓
- S3 to S5: S5 is flow dependent on S3 ✓
- S4 to S5: No dependency

Dependence Graph

Resource Dependency:

- S3 and S4: Both are using adders (+) ✓

Solution:

```

S1: LOAD R1, 1024      //  $R_1 \leftarrow 1024$ 
S2: LOAD R2, M(10)     //  $R_2 \leftarrow \text{MEMORY}(10)$ 
S3: ADD R1, R2          //  $R_1 \leftarrow R_1 + R_2$ 
S4: STORE M(1024), R1   //  $\text{MEMORY}(1024) \leftarrow R_1$ 
S5: STORE M(R2), 1024   //  $\text{MEMORY}(R_2) \leftarrow 1024$ 

```

Data Dependencies:

- S1 to S2: No dependency ✓
- S1 to S3: S3 is flow and output dependent on S1 ✓
- S1 to S4: S4 is flow dependent on S1 ✓
- S1 to S5: No dependency ✓
- S2 to S3: S3 is flow dependent on S2 ✓ ✓
- S2 to S4: No dependency ✓
- S2 to S5: S5 is flow dependent on S2 ✓ ✓
- S3 to S3: S3 is flow dependent on S3 itself ✓
- S3 to S4: S4 is flow dependent on S3 ✓
- S3 to S5: No dependency
- S4 to S5: No dependency

Dependence Graph

Resource Dependency:

- S1 and S2: both are using load unit
- S4 and S5: both are using store unit

LOAD AND STORE UNIT CAN BE BASICALLY : READ AND WRITE BUS !! { TO WAHI HAI !! }

VVIMP QUESTION :

Solution:

```

S1: A = B + C
S2: C = B * D
S3: S = 0
S4: DO I=A, 100
    S = S + X(I)
  END DO
S5: IF(S .GT. 1000) C = C * 2

```

Input Sets **Output Sets**

$I_1 = \{B, C\}$	$O_1 = \{A\}$
$I_2 = \{B, D\}$	$O_2 = \{C\}$
$I_3 = \emptyset$	$O_3 = \{S\}$ ✓
$I_4 = \{A, S, I, X\}$	$O_4 = \{S\}$
$I_5 = \{S, C\}$	$O_5 = \{C\}$

For S3 & S4:
 $I_3 \cap O_4 = \emptyset$, $I_4 \cap O_3 = \{S\}$, $O_3 \cap O_4 = \{S\}$
 $(S_3 \parallel S_4)$

For S3 & S5:
 $I_3 \cap O_5 = \emptyset$, $I_5 \cap O_3 = \{S\}$, $O_3 \cap O_5 = \emptyset$
 $(S_3 \parallel S_5)$

For S4 & S5:
 $I_4 \cap O_5 = \emptyset$, $I_5 \cap O_4 = \{S\}$, $O_4 \cap O_5 = \emptyset$
 $(S_4 \parallel S_5)$

So 3 pairs can execute in parallel ($S_1 \parallel S_3$), ($S_2 \parallel S_3$), ($S_2 \parallel S_4$)

```

S1: A = B + C ✓
S2: C = B * D ✓
S3: S = 0 ✓
S4: DO I=A, 100
    S = S + X(I) ✓
END DO
S5: IF (S .GT. 1000) C = C * 2 ✓

```

(b) Now Reconstructing the given program to maximize the parallelism:

Because 3 pairs can execute in parallel (S1 || S3), (S2 || S3), (S2 || S4) ✓

✓ P1:	A = B + C	S = 0 ✓
● P2:	C = B * D	DO I=A, 100 S = S + X(I) END DO } C = C * 2
P3:	IF (S .GT. 1000)	

Bernstein's Conditions

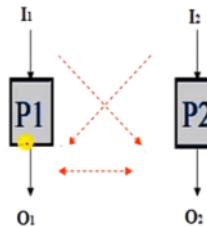


According to Bernstein:

Two processes P_1, P_2 with input sets I_1, I_2 and output sets O_1, O_2 can execute in parallel (denoted by $P_1 \parallel P_2$) if:

$$\left\{ \begin{array}{l} I_1 \cap O_2 = \emptyset \\ I_2 \cap O_1 = \emptyset \\ O_1 \cap O_2 = \emptyset \end{array} \right.$$

These three conditions are known as **Bernstein's Conditions**



Bernstein's Conditions...



□ *In terms of data dependencies,*

Bernstein's conditions imply that two processes can execute in parallel if they are:

- flow-independent, ✓
- anti-independent, and ✓
- output independent ✓

$I_1 \cap O_2 = \emptyset$ Anti-independent i.e no anti dependence

$I_2 \cap O_1 = \emptyset$ Flow-independent i.e no flow dependence

$O_1 \cap O_2 = \emptyset$ Output-independent i.e. no output dependence

Bernstein's Conditions...



- In general $P_1, P_2, P_3, \dots, P_k$ can execute in parallel if Bernstein's Conditions are satisfied on a pair-wise basis. This means that:

$$P_1 \parallel P_2 \parallel P_3 \parallel \dots \parallel P_k \quad \text{if and only if } P_i \parallel P_j \text{ for all } i \neq j$$

Example 1

Consider following statements:

$$a = \underline{x} + y;$$

$$b = \underline{x} + z;$$

Do they satisfy Bernstein's Conditions?

➤ **Solution:** $I_1 = \{x, y\}$ $O_1 = \{a\}$

$$I_2 = \{x, z\} \quad O_2 = \{b\}$$

Here, $I_1 \cap O_2 = \emptyset$

$$I_2 \cap O_1 = \emptyset$$

$$O_1 \cap O_2 = \emptyset$$

So, Bernstein's Conditions are satisfied.

Hence, these two statements can be executed in parallel.

Question 1:



- Consider the following instructions P_1, P_2, P_3, P_4, P_5 are in program order. Assume each instruction requires one step to execute, no pipelining is considered here.

A) Detect if parallelism is possible using Bernstein's Conditions

B) Draw Dependence graph and Resource Dependence graph

C) Find how many steps are needed in case of sequential execution and parallel execution if two adders are available.

$$P_1 : C = D \times E$$

$$P_2 : M = G + C$$

$$P_3 : A = B + C$$

$$P_4 : C = L + M$$

$$P_5 : F = G \div E$$

**See BERNSTIEN KE ANDAR : HAR 2 PROCESS KE BEECH ME RAW , WAR AND WAW ,
RESOURCE DEPENDANCY DEKHENG OFC !!**

$P1 : C = D \times E$ $P2 : M = G + C$ $P3 : A = B + C$ $\cancel{P4 : C = L + M}$ $\cancel{P5 : F = G \div E}$	For P3 & P4: $I_3 \cap O_4 = \{C\}, I_4 \cap O_3 = \emptyset, O_3 \cap O_4 = \emptyset$ Bernstein's Conditions does not satisfied. So P3 and P4 can't executed in parallel i.e. $(P_3 \parallel P_4)$
--	---

Input Sets $I_1 = \{D, E\}$ $I_2 = \{G, C\}$ $I_3 = \{B, C\}$ $\cancel{I_4 = \{L, M\}}$ $I_5 = \{G, E\}$	Output Sets $O_1 = \{C\}$ $O_2 = \{M\}$ $O_3 = \{A\}$ $\cancel{O_4 = \{C\}}$ $O_5 = \{F\}$
--	--

--	--

For P3 & P5: $I_3 \cap O_5 = \emptyset, I_5 \cap O_3 = \emptyset, O_3 \cap O_5 = \emptyset$ Bernstein's Conditions are satisfied. So P3 and P5 can be executed in parallel $(P_3 \parallel P_5)$	X ✓
--	---

For P4 & P5: $I_4 \cap O_5 = \emptyset, I_5 \cap O_4 = \emptyset, O_4 \cap O_5 = \emptyset$ Bernstein's Conditions are satisfied. So P4 and P5 can be executed in parallel $(P_4 \parallel P_5)$	X ✓
--	---

So 5 pairs can execute in parallel $(P_1 \parallel P_5), (P_2 \parallel P_3), (P_2 \parallel P_5), (P_3 \parallel P_5), (P_4 \parallel P_5)$

TYPES OF PROCESSORS :

Vector Processor:

- Deals with multiple data elements (vectors) simultaneously.
- Executes the same instruction on each element of the vector, achieving parallelism and potentially significant performance gains.
- Useful for tasks with inherent parallelism, like scientific calculations, image processing, and video encoding.
- Examples: GPUs, SIMD (Single Instruction, Multiple Data) architectures.

Scalar Processor:

- Processes data one element at a time.
- Executes instructions sequentially, focusing on single values.
- More common in traditional CPUs, suitable for general-purpose computing and tasks that don't benefit from parallelization.
- Examples: Most general-purpose CPUs, microcontrollers.

Key Differences:

Processor Types

□ Vector processor:

- ❖ Cray X1/T90; NEC SX#; Japan Earth Simulator; Early Cray machines; Japan Life Simulator (hybrid)

□ Scalar processor:

- ❖ **CISC:** Complex Instruction Set Computer
 - Intel 80x86 (IA32)
- ❖ **RISC:** Reduced Instruction Set Computer
 - Sun SPARC, IBM Power #, SGI MIPS

1. Scalar Processors:

- Scalar processors are traditional CPUs (Central Processing Units) that operate on scalar data, which means they process one data item at a time.
- These processors are optimized for executing a single instruction on a single piece of data.
- Scalar processors are well-suited for general-purpose computing tasks where each operation doesn't require simultaneous processing of multiple data elements.
- Examples of scalar processors include most CPUs found in personal computers, servers, and mobile devices.

2. Vector Processors:

- Vector processors, on the other hand, are designed to operate on vectors or arrays of data elements simultaneously.
- Instead of processing one data item at a time, vector processors can perform the same operation on multiple data elements in parallel.
- They excel in tasks that involve large amounts of data parallelism, such as multimedia processing, scientific simulations, and certain types of mathematical computations.
- Vector processors often have specialized hardware for efficiently executing vector operations.
- Examples of vector processors include GPUs (Graphics Processing Units), which are heavily used in graphics rendering, scientific computing, and machine learning due to their ability to handle parallel computations on large datasets.

In summary, scalar processors are optimized for handling individual data elements sequentially,
while vector processors are designed to process multiple data elements simultaneously, making

In summary, scalar processors are optimized for handling individual data elements sequentially,
while vector processors are designed to process multiple data elements simultaneously, making
them particularly effective for tasks with high levels of data parallelism.

RISC VS CISC PROCESSORS :

1. CISC (Complex Instruction Set Computer):

- CISC processors have a rich set of complex instructions that can perform multiple low-level operations within a single instruction.
- Instructions in CISC architectures can be quite powerful and versatile, capable of performing tasks that might require multiple simpler instructions in a RISC architecture.
- CISC processors typically have a larger and more diverse instruction set, which can include instructions for operations such as loading from memory, arithmetic and logic operations, and control flow instructions.
- Examples of CISC architectures include x86, which is widely used in personal computers and servers, as well as architectures like Motorola 68k and DEC VAX.

2. RISC (Reduced Instruction Set Computer):

- RISC processors, as the name suggests, have a simplified and streamlined instruction set, focusing on a smaller number of instructions that execute quickly.
- Instructions in RISC architectures are generally simpler and more atomic, performing basic operations with a single instruction.
- RISC architectures typically have a smaller, more uniform instruction set, which can lead to easier instruction decoding and more efficient pipelining.
- RISC processors often rely on optimizing compiler techniques to achieve high performance by breaking down complex tasks into sequences of simpler instructions.
- Examples of RISC architectures include ARM, MIPS, and PowerPC.

Key differences between RISC and CISC processors include:

Key differences between RISC and CISC processors include:

- **Instruction Complexity:** CISC processors have complex instructions capable of performing multiple tasks, while RISC processors have simpler, more atomic instructions.
- **Instruction Set Size:** CISC processors have larger instruction sets compared to RISC processors.
- **Pipeline Complexity:** RISC processors typically have simpler instruction pipelines, which can lead to more efficient instruction execution.
- **Compiler Dependence:** RISC processors often rely more heavily on compiler optimizations to achieve high performance, while CISC processors may rely on complex hardware features.

Both RISC and CISC architectures have their own advantages and trade-offs, and the choice between them depends on factors such as performance requirements, power consumption, and ease of programming.



CISC

- ❖ Complex instructions; Large number of instructions; Can complete more complicated functions at instruction level
 - ❖ Instruction actually invokes microcode. Microcodes are small programs in processor memory
 - ❖ Slower; Many instructions access memory; varying instruction length; allow no pipelining;
-

RISC Processor

- No microcode
- Simple instructions; Fewer instructions; Fast
- Only load and store instructions access memory
- Common instruction word length
- Allows pipelining

Almost all present-day high performance computers use RISC processors

L What is locality of reference ? What is spatial locality and temporal locality ?

Show drafts ▾ 🔍

Locality of reference refers to the tendency of a program to access memory locations that are close together in **both space (memory addresses) and time**. This principle forms the basis for optimizing cache performance in computer systems.

There are two main types of locality of reference:

1. **Spatial locality:** This refers to the tendency of a program to access memory locations that are **near** the one it recently accessed. For example, if a program reads a value from an array, it is likely to access the next elements in the array soon after.
2. **Temporal locality:** This refers to the tendency of a program to access the same memory location **multiple times within a short period**. For example, a loop that iterates over an array will repeatedly access the same memory locations for each iteration.

Locality of References

□ Spatial/Temporal locality

- ❖ If processor executes an instruction at time t , it is likely to execute an adjacent/next instruction at $(t+\delta_t)$;
- ❖ If processor accesses a memory location/data item x at time t , it is likely to access an adjacent memory location/data item $(x+\delta_x)$ at $(t+\delta_t)$;

Pipelining, Caching and many other techniques all based on the locality of references

L What do you mean by multiprocessors and multiComputers ?

Show drafts ▾ 🔍

Both multiprocessors and multicompilers are forms of parallel computing systems, but they differ in their architecture and design approach:

Multiprocessor:

- **Single system:** A single physical computer with multiple central processing units (CPUs) sharing memory and I/O devices.
- **Shared memory:** All CPUs have direct access to a single global memory space, enabling efficient data sharing and communication.
- **Tightly coupled:** CPUs communicate and synchronize closely due to shared resources and memory.
- **Examples:** Multi-core CPUs, servers with multiple CPUs.

Multicompiler:

- **Network of computers:** A collection of independent computers connected through a network. Each computer has its own memory and I/O devices.
- **Distributed memory:** Each computer has its own local memory, and communication between computers happens through message passing on the network.
- **Loosely coupled:** CPUs operate independently with less frequent communication compared to multiprocessors.
- **Examples:** Clusters of computers, large-scale computing systems.

Here's a table summarizing the key differences:

Feature	Multiprocessor	Multicompiler
Architecture	Single system	Network of computers
Memory	Shared memory	Distributed memory
Communication	Shared memory access	Message passing
Coupling	Tightly coupled	Loosely coupled

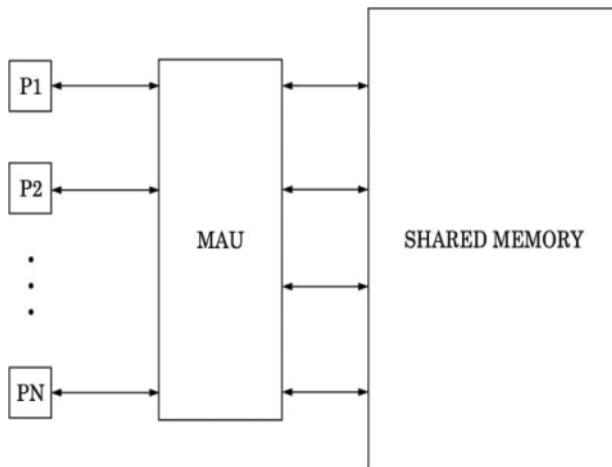
Feature	Multiprocessor	Multicomputer
Architecture	Single system	Network of computers
Memory	Shared memory	Distributed memory
Communication	Shared memory access	Message passing
Coupling	Tightly coupled	Loosely coupled
Scalability	Limited by shared resources	More scalable, can add more computers
Cost	Less expensive per CPU	More expensive due to additional network hardware
Complexity	Less complex hardware, more complex software	More complex hardware and software
Examples	Multi-core CPUs, servers	Clusters, supercomputers

 Export to Sheets

The Parallel Random Access Machine (PRAM)

The PRAM is one of the popular models for designing parallel algorithms, consists of the following:

- A set of $N(P_1, P_2, \dots, P_N)$ identical processors. In principle, N is unbounded.
- A memory with M locations which is shared by all the N processors. Again, in principle, M is unbounded.
- An MAU which allows the processors to access the shared memory.



The Parallel Random Access Machine (PRAM)

- The PRAM model can be subdivided into 4 categories based on the way simultaneous memory accesses are handled.
 - Exclusive Read Exclusive Write (**EREW**) PRAM
 - every access to a memory location (read or write) has to be exclusive.
 - Concurrent Read Exclusive Write (**CREW**) PRAM
 - Only write operations to a memory location are exclusive.
 - Exclusive Read Concurrent Write (**ERCW**) PRAM
 - Multiple processors can concurrently write into the same memory location.
 - Concurrent Read Concurrent Write (**CRCW**) PRAM
 - Allows both multiple read and multiple write operations to a memory location.

PRAM Variants

		Reads	
		Exclusive	Concurrent
Writes	Exclusive	EREW	CREW
	Least effective Most simple	Most common	
Concurrent	ERCW	CRCW	
	Practically no use	Most powerful Most complex	

The Parallel Random Access Machine (PRAM)

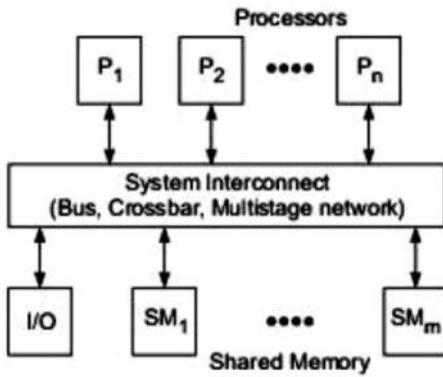
- There are many methods to implement the PRAM model, but the most prominent ones are:
 - Shared memory model
 - Message passing model
 - Data parallel model

Multiprocessors and Multicomputers

- These physical models are distinguished by having a **shared common memory** or **unshared distributed memories**.
- The processors in a multiprocessor system communicate with each other through shared variables in a common memory.
- Each computer node in a multicomputer system has a local memory, unshared with other nodes. Inter-processor communication is done through message passing among the nodes.
- There are **three types of shared memory multiprocessor:**
 - **UMA** (Uniform Memory Access)
 - **NUMA** (Non-uniform Memory Access)
 - **COMA** (Cache Only Memory)

Uniform Memory Access (UMA)

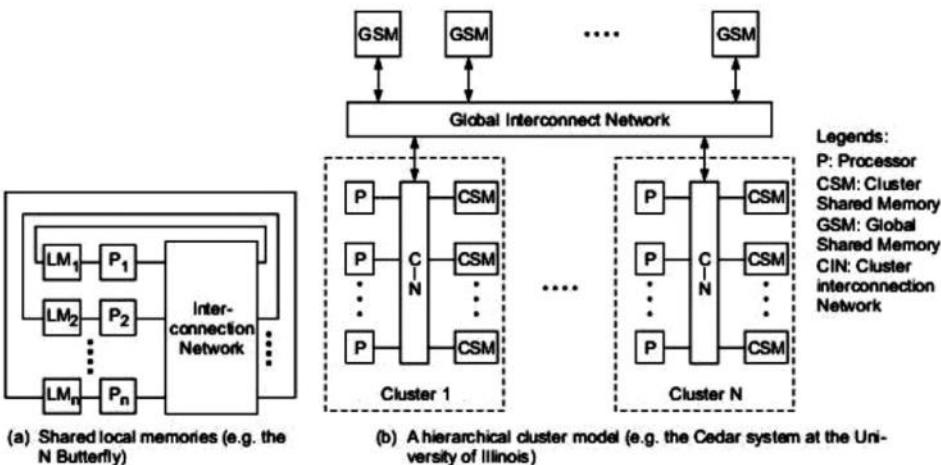
- Most commonly represented today by Symmetric Multiprocessor (SMP) machines.
- Identical processors.
- Equal access and access times to memory.
- Sometimes called CC-UMA - Cache Coherent UMA.
- Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.
- Multiprocessors are tightly coupled.
- The UMA model is suitable for general-purpose and times sharing applications by multiple users.



Non-Uniform Memory Access (NUMA)

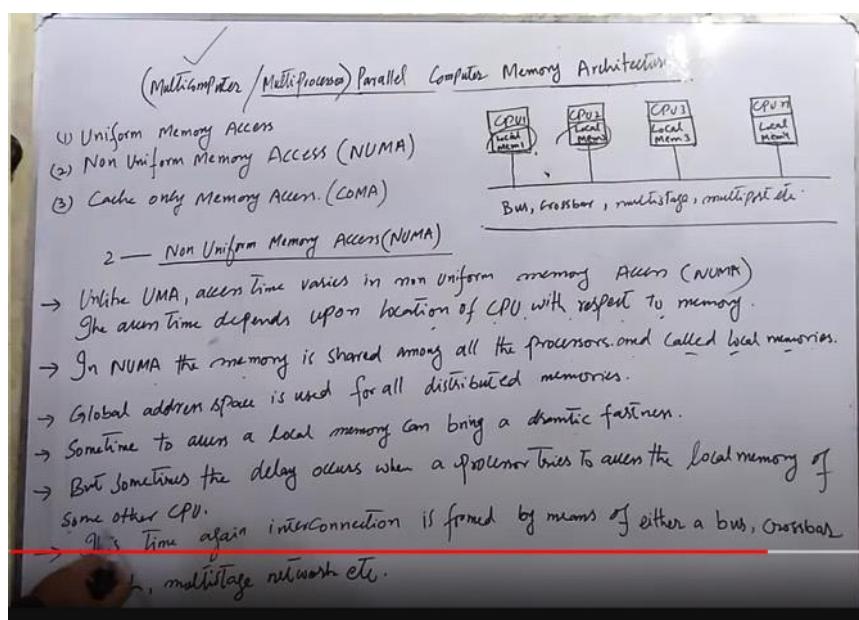
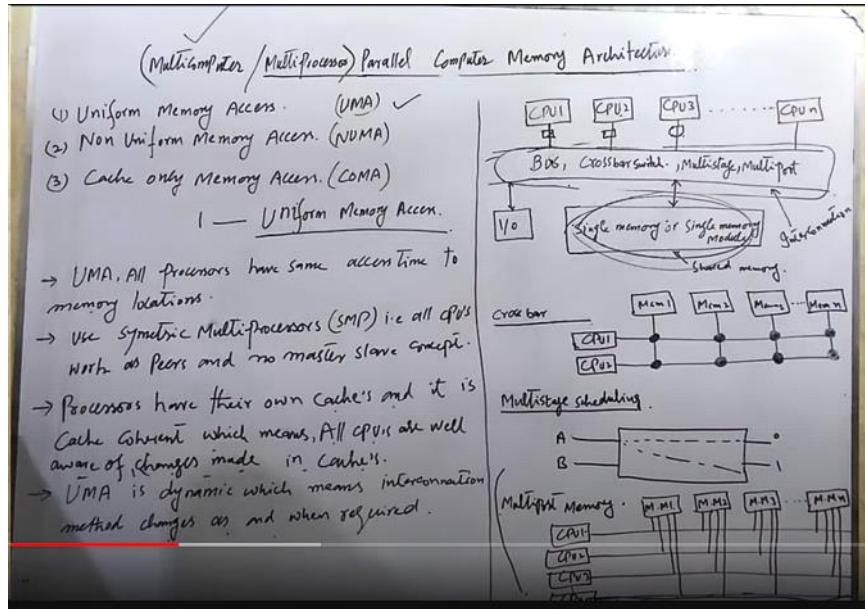
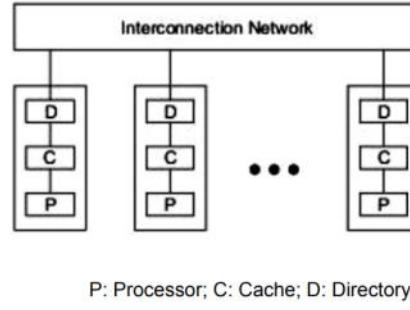
- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower.
- If cache coherency is maintained, then may also be called CC-NUMA
- Cache Coherent NUMA

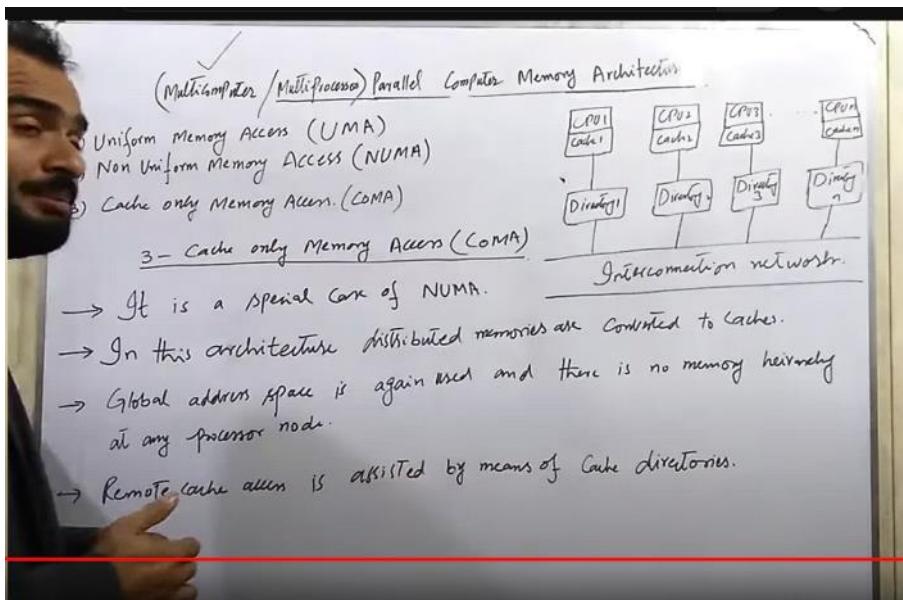
Non-Uniform Memory Access (NUMA)



The COMA model (Cache only Memory Access)

- The COMA model is a special case of NUMA machine in which the distributed main memories are converted to caches.
- All caches form a global address space and there is no memory hierarchy at each processor node.





1. **Structural hazards:** These occur when the hardware resources required by one instruction are not available because they are already being used by another instruction in the pipeline. For example, if two instructions require access to the same functional unit (such as the ALU), a structural hazard can occur, causing a stall until the resource becomes available.
2. **Data hazards:** Data hazards arise when an instruction depends on the result of a previous instruction that has not yet completed. There are three types of data hazards:
 - **Read-after-write (RAW) hazard:** A subsequent instruction attempts to read a register before a previous instruction writes to it. This can cause a stall until the required data is available.
 - **Write-after-read (WAR) hazard:** A subsequent instruction attempts to write to a register before a previous instruction reads from it. This can lead to incorrect program behavior but typically doesn't cause a stall.
 - **Write-after-write (WAW) hazard:** A subsequent instruction attempts to write to a register before a previous instruction writes to it. This can cause a stall or result in incorrect program behavior.
3. **Control hazards:** Control hazards occur when the pipeline incorrectly predicts the flow of control instructions (e.g., branches or jumps). If the prediction is incorrect, instructions following the branch must be flushed from the pipeline, causing a stall until the correct instruction address is known.

Unit -2 :

Hardware Parallelism

- This refers to the type of parallelism defined by the machine architecture and hardware multiplicity.
- Hardware parallelism is often a function of cost and performance tradeoffs.
- It displays the resource utilization patterns of simultaneously executable operations.
- It can also indicate the peak performance of the processor resource.
- One way to characterize the parallelism in a processor is by the number of instruction issues per machine cycle.
- If a processor issues k instructions per machine cycle, then it is called a **k -issue** processor.
- A conventional pipelined processor takes one machine cycle to issue a single instruction.
- These types of processors are called one-issue machines, with a single instruction pipeline in the processor.
- In a modern processor, two or more instructions can be issued per machine cycle.

K issue processors means : k diff processors !! Working together : to fetch : k instructions per machine cycle !!

Software Parallelism:

- Concerned with how **software is designed and written to take advantage of available hardware parallelism**.
- Achieved through techniques like:
 - **Parallelizing algorithms:** Breaking down large tasks into smaller, independent subtasks that can be executed concurrently.
 - **Using parallel programming constructs:** Languages like C++ and Python offer libraries and syntax for creating threads and processes to exploit parallel hardware.
 - **Optimizing compiler settings:** Compilers can sometimes restructure code to improve parallelism based on specific hardware capabilities.

Program Decomposition

• General Ideas:

- Identify the portions of code that can be done in parallel.
 - Mapping the code onto multiple processes.
 - Distributing the input, output, and intermediate data
 - Managing the access to shared resources.
 - Synchronizing the processes at various stages of the program.
- Data Hazards

L What is program decomposition in software parallelism ?

Show drafts ▾ 🔍

💡 In software parallelism, **program decomposition** is the crucial first step where you break down your program into smaller, independent tasks that can be executed concurrently on different processing units. It's the foundation upon which effective parallel programming is built.

Here's a closer look at program decomposition:

Why it's important:

- Modern processors have multiple cores/threads, and software needs to be designed to harness this potential. Decomposition allows you to split work across these units, achieving faster execution.
- Complex problems are often easier to solve when tackled in smaller, more manageable pieces. Decomposition aids in understanding and structuring the problem in a parallel-friendly way.

Key approaches to decomposition:

Key approaches to decomposition:

- **Functional decomposition:** Identify distinct functionalities within the program that can be executed independently. Imagine separating image processing tasks like filtering and resizing into individual modules.
- **Data decomposition:** Divide data sets into smaller chunks that can be processed concurrently. Think of distributing a large image file among multiple threads for parallel filtering.
- **Task-based decomposition:** Break down the program into well-defined tasks with clear inputs and outputs, suitable for independent execution on different threads or processes. Consider parallelizing individual calculations within a scientific simulation.

Code Decomposition

- **Decomposition:** the operation of dividing the computation into smaller parts, some of which may be executed in parallel.
- **Task:** programmer-defined units of code resulting from decomposition.
- **Granularity:** the number / size of the tasks.
- **Fine-grained decomposition:** a large number of tasks
- **Coarse-grained decomposition:** small number of tasks.
- **Degree of concurrency:** the maximum number of tasks that can be executed in the same time.

The correct formula for granularity is:

Granularity = Total amount of work / Number of tasks

L so if the program is decomposed into 1 sized instructions then : grainuality = 1 ? 

Show drafts  

◆ You're absolutely right! If the program is decomposed into 1-sized instructions, then the granularity would indeed be 1. This translates to a scenario where **every single instruction in the program becomes an independent task**. However, in practice, **decomposing a program into such fine-grained tasks is almost never beneficial** and often detrimental to performance.

ChatGPT

Sure, let's delve into how parallelism is achieved in each decomposition strategy:

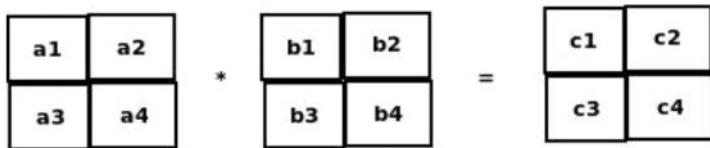
1. Recursive Decomposition:

- **Parallelism Achievement:** Parallelism in recursive decomposition is often achieved by exploiting the divide-and-conquer nature of the algorithm. Once the problem is divided into smaller sub-problems, these sub-problems can often be solved independently. This independence allows for parallel execution of sub-problems across multiple processing units. Each processing unit works on its assigned sub-problem concurrently with others, either on separate CPU cores, threads, or computing nodes in a distributed system.
- **Example:** In parallel merge sort, the original array is divided into smaller sub-arrays recursively. These sub-arrays can then be sorted independently in parallel. After sorting, the sorted sub-arrays are merged in parallel to produce the final sorted array.

2. Data Decomposition:

- **Parallelism Achievement:** Parallelism in data decomposition arises from the concurrent processing of different portions of the data by multiple processing units. The dataset is partitioned into smaller chunks, and each chunk is assigned to a different processing unit for computation. These units can execute simultaneously, each working on its assigned portion of the data. Once individual computations are completed, results may need to be combined or aggregated.
- **Example:** In parallel matrix multiplication, the input matrices are divided into smaller blocks or rows/columns. Each processor or computing node is responsible for computing a portion of the resulting matrix, which can be done simultaneously with other processors. Finally, the partial results are combined to produce the final output matrix.

Data Decomposition



- $c_1 = a_1 * b_1 + a_2 * b_3$
- $c_2 = a_1 * b_2 + a_2 * b_4$
- $c_3 = a_3 * b_1 + a_4 * b_3$
- $c_4 = a_3 * b_2 + a_4 * b_4$

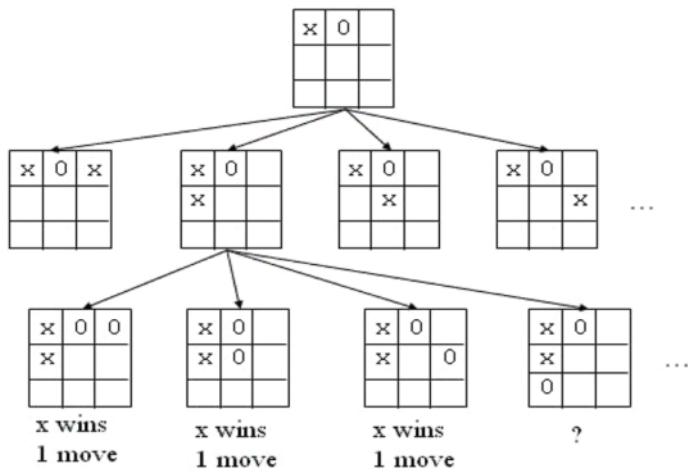
3. Functional Decomposition:

- **Parallelism Achievement:** Parallelism in functional decomposition typically involves executing different functions or procedures concurrently on multiple processing units. Functions or modules that are independent of each other can be executed in parallel, either within the same program instance (e.g., using threads) or across multiple instances (e.g., using distributed computing).
- **Example:** In a text processing application, different functions can be designed to handle distinct tasks such as reading input, tokenizing text, analyzing grammar, and generating output. These functions can be executed concurrently, with each function operating on its input data independently of others.

4. Exploratory Decomposition:

- **Parallelism Achievement:** Parallelism in exploratory decomposition is less about concurrent execution and more about parallelizing the exploration of different problem-solving approaches. Multiple strategies or algorithms may be explored simultaneously by different processing units or threads. Each unit evaluates a different approach, potentially in parallel, to identify the most effective solution.
- **Example:** In algorithm optimization, different approaches may be tested concurrently to determine the most efficient one. Each approach can be evaluated independently by different threads or processes, and the results compared to select the best-performing strategy.

Exploratory Decomposition



ALL BRANCHES OF THE ABOVE RECURSION CAN BE MADE TO RUN IN PARALLEL THEREBY SAVING TIME !!

Speculative Decomposition

- When a program may take one of many possible computationally significant branches depending on the output of other computations that precede it.
- In this situation, while one task is performing the computation whose output is used in deciding the next computation, other tasks can concurrently start the computations of the next stage.

Example of Speculative Execution

- Evaluating one or more of the branches of a *switch* statement in C in parallel before the input for the *switch* is available.
- While one task is performing the computation that will eventually resolve the *switch*, other tasks could pick up the multiple branches of the *switch* in parallel.
- When the input for the *switch* has finally been computed, the computation corresponding to the correct branch would be used while that corresponding to the other branches would be discarded.
- May result in wasteful computations, therefore, only the most promising branch is taken up a task in parallel with the preceding computation.

TERMINOLOGIES WRT CODE DECOMPOSITION TO ACHIEVE SOFTWARE PARALELLISM :

1. Task Generation:

- **Static Task Generation:** In static task generation, the tasks to be executed are known in advance, typically at compile time or before runtime. This approach is commonly used in data decomposition techniques, where the program is decomposed into a predefined set of tasks based on the partitioning of input data.
- **Dynamic Task Generation:** In dynamic task generation, the decision on which tasks to execute is made at runtime, based on the program's execution state or other runtime conditions. This approach is often used in recursive decomposition techniques, where the program recursively generates and executes tasks based on the results of previous computations.

2. Task Size:

- **Uniform Task Size:** In uniform task size, all tasks require approximately the same amount of computational effort or execution time. This uniformity simplifies load balancing and resource allocation in parallel processing systems.
- **Non-uniform Task Size:** In non-uniform task size, tasks vary significantly in their computational requirements or execution times. This variation may arise due to differences in input data, algorithmic complexity, or other factors. Handling non-uniform task sizes requires more sophisticated load balancing techniques to ensure efficient resource utilization.

2. Task Interaction:

3. Task Interaction:

- **Static Task Interaction:** In static task interaction, interactions between tasks occur at predetermined times, and the set of tasks involved in each interaction is known in advance. This static nature simplifies coordination and synchronization between tasks, as the communication patterns are predefined.
- **Dynamic Task Interaction:** In dynamic task interaction, the timing of interactions between tasks or the set of tasks involved in each interaction is unpredictable and may vary during program execution. Dynamic task interaction poses greater challenges for coordination and synchronization, as the communication patterns are not predetermined and may depend on runtime conditions.
- **Regular/Irregular Task Interaction:** Regular task interaction occurs when the interactions between tasks follow a predictable pattern that can be exploited for efficiency. For example, tasks may communicate in a regular grid pattern in some parallel algorithms. In contrast, irregular task interaction occurs when the communication patterns between tasks are unpredictable or irregular, making coordination more challenging.

These characteristics of tasks play a crucial role in determining the performance, scalability, and efficiency of parallel processing systems. Understanding these characteristics helps programmers and system designers choose appropriate decomposition techniques, develop efficient parallel algorithms, and implement effective coordination and synchronization mechanisms to maximize the benefits of parallelism.



L You

give examples of static and dynamic task interaction



ChatGPT

Certainly! Here are examples of static and dynamic task interaction:

1. Static Task Interaction:

- **Example 1: Matrix Multiplication:**
In matrix multiplication, each element of the resulting matrix is computed by taking the dot product of a row from the first matrix and a column from the second matrix. The computation of each element involves interactions between a fixed set of tasks corresponding to the elements of the input matrices. The communication pattern between tasks is static and predetermined based on the dimensions of the input matrices.
- **Example 2: Parallel Sorting:**
In parallel sorting algorithms such as parallel quicksort, the input array is divided into partitions, and each partition is sorted independently by separate tasks. Once all partitions are sorted, the results are combined using a merging operation. The interactions between tasks, such as merging sorted partitions, occur at predetermined points during the algorithm execution and involve a fixed set of tasks.

2. Dynamic Task Interaction:

- **Example 1: Parallel Tree Traversal:**

In parallel tree traversal algorithms, such as parallel depth-first search (DFS) or breadth-first search (BFS), tasks explore the nodes of a tree structure concurrently. The interactions between tasks are dynamic and depend on the structure of the tree and the order in which nodes are processed. For example, in DFS, tasks may dynamically interact to share information about discovered nodes and coordinate traversal progress.

- **Example 2: Parallel Monte Carlo Simulation:**

In parallel Monte Carlo simulations, tasks perform independent simulations of a stochastic process, such as option pricing or risk analysis. The interactions between tasks are dynamic and depend on the random outcomes of each simulation. For example, tasks may communicate to aggregate simulation results, update global statistics, or synchronize simulation progress dynamically based on runtime conditions.

Characteristics of Tasks

- **Task generation:**

- static - the tasks are known in advance (data decomposition)
- dynamic - decided at runtime (recursive decomposition)

- **Task size:**

- uniform (they require approximately the same amount of time) or
- Non-uniform
- known/not known.

- **Task Interaction**

- Static: it happens at predetermined times and the set of tasks to interact with is known in advance.
- Dynamic: the timing of the interaction or the set of tasks to interact with are unpredictable. Harder to implement.
- Regular/irregular: it is regular if the interaction follows a pattern that can be exploited for efficiency.

RELATION BTW GRANUALITY AND LATENCY :

LOWER THE GRANUALITY / GRAIN SIZE: MEANS : MORE ARE THE MACHINES IN PARALLEL ?

YES IG : MEANING : HIGHER LATENCY BCZ MORE IS THE INTER PROCESSOR COMMUNICATION THEREBY MORE IS THE LATENCY !!

Grain Sizes and Latency

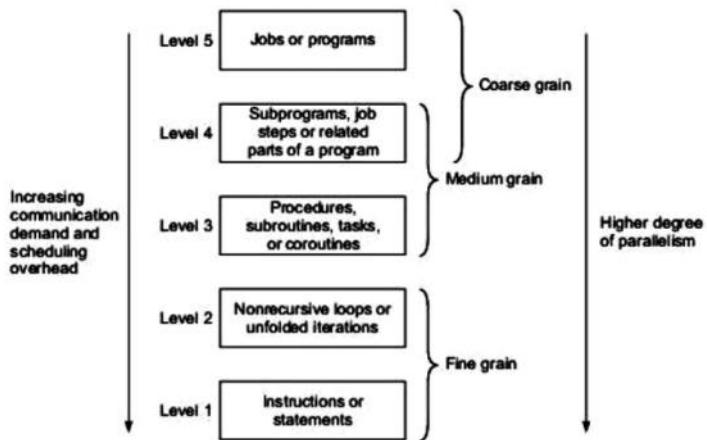
- **Grain size or granularity** is a measure of the amount of computation involved in a software process.

- The simplest measure is to count the number of instructions in a grain (program segment).
- Grain size determines the basic program segment chosen for parallel processing.
- Grain sizes are commonly described as fine, medium or coarse, depending on the processing levels involved.

- **Latency** is a time measure of the communication overhead incurred between machine subsystems.

- For example, the memory latency is the time required by a processor to access the memory.
- The time required for two processes to synchronise with each other is called synchronisation latency.
- Computational granularity and communication latency are closely related.

Levels of parallelism in program execution on modern computers



GRAIN PACKING AND SCHEDULING :

