Harvard Architecture → Separate data & instruction space

CACSC20 - HPC

Shared data & instruction space

Von-neuman Architecture → sequential
Program stored in [memory]
1 is shared
Instructions & operands ← [Processor]

*) **FLYNN'S CLASSIFICATION OF PARALLEL ARCHITECTURES** : Based on the multiplicity of instruction stream and data streams in a computer system. Given by Michael J Flynn in 1966

→ Two types of information flow into a processor → Instructions, Data

*) **Instruction Stream** : Sequence of instructions executed by the processing unit

*) **Data Stream** : Sequence of data including i/p, partial or temporal results, called by the instruction stream.

According to Flynn's classification, either of the instruction or data streams can be single or multiple. Computer architecture can be classified into four categories :-

Computer Architecture

Single-instruction stream Single-Data stream (SISD)

→ uniprocessor machine executes single instruction on a single data stream

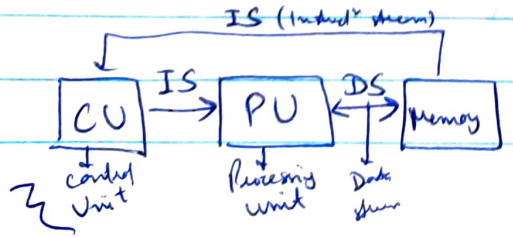
→ Conventional single-processor Von-neuman computers

→ Serial computer (Not II)

→ Pipelining present (Parallelism introduced)

→ More than one functional unit under a control unit

Eg : PCs, single CPU workstations, mainframes CDC-6600, VAX11, IBM 7001



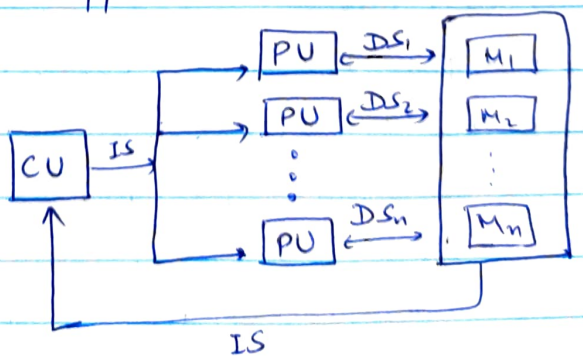
Single instruction stream Multiple-data stream (SIMD)

→ same instruction on multiple data streams

→ Single control unit (CU) but multiple processing unit (PU)

→ Scientific computation, vector & matrices

→ Eg: Array processors : ILLIAC-IV, MPP
Vector pipelines : IBM 9000, C90



Multiple-instruction stream Single-data stream (MISD)

→ Multiple different instructions can be executed on single data stream

→ Multiple CU & PU but they work on same DS

→ Practically useless

→ Eg : Systolic Arrays

Multiple instruction Multiple data stream (MIMD)

→ Diff IS can be issued on diff DS

→ Maximum Parallelism

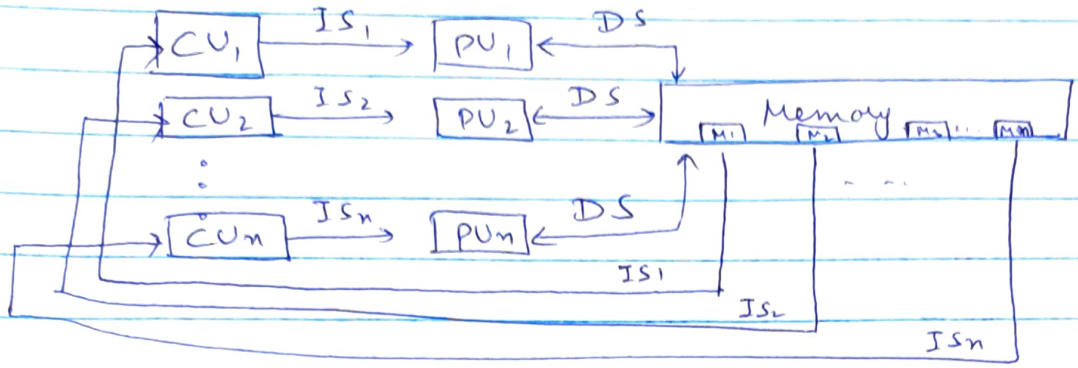
Systolic array

→ Loosely / Tightly coupled

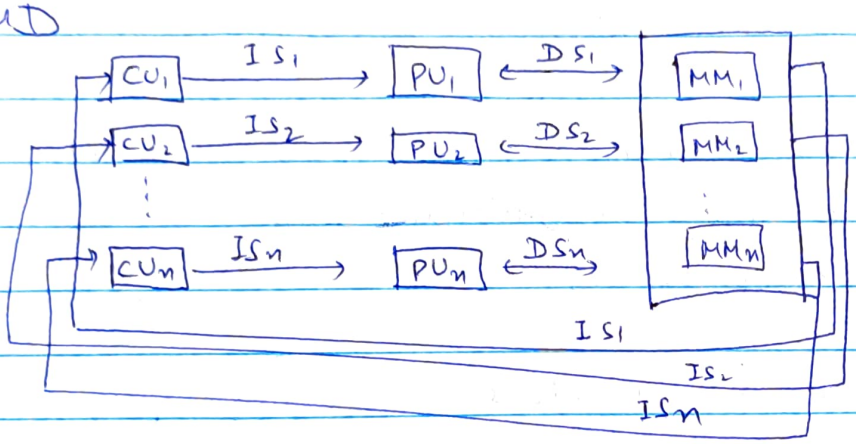
→ shared memory

→ Eg : multiprocessor Supercomputers IBM-370, "grids" UNIVAC-1100/80

MISD



MIMD



*) TEMPORAL PARALLELISM : Breaking up a job into a set of tasks to be executed overlapped in time. Also called assembly line processing, pipeline processing or vector processing.

Eg: 4 teacher checking a question paper, 1 question each
Job = checking paper Task = checking one answer

T_1	T_2	T_3	T_4
P_1			
P_2	P_1		
P_3	P_2	P_1	
P_4	P_3	P_2	P_1
P_5	P_4	P_3	P_2

Conditions for temporal parallelism :-

i) Jobs must be identical
ii) Job can be divided into independent tasks
iii) Each task takes same time
iv) Time to pass a task from 1 processor to other is neglible wrt task execution time
v) No. of tasks << No. of jobs

If no. of jobs = n, time to do a job = p. Each job divided into k tasks

Time for each task = p/k

Time to complete n jobs without pipeline = np

" " " " " " with " = $p + (n-1)p/k = p\frac{(k+n-1)}{k}$

Speed-up = $\frac{np}{p(\frac{k+n-1}{k})} = \frac{k}{1+\frac{k-1}{n}}$

No. of tasks ↗

*) Problems in temporal parallelism:

i) Synchronization: Time to do each task is equal, else holdup
ii) Bubbles in pipeline: If some tasks are missing in some jobs → Bubbles → Idle processors doing these tasks
iii) Fault tolerance: If 1 processor fails, all fail
iv) Inter-task communicat^n: Time to pass task must be negligible to task execut^n time
v) Scalability: The no. of processor to divide task = no. of individual tasks
↳ can't increase after certain limit

Pipelining is still used in vector supercomputers like CRAY as each processor expertizes its work

*) DATA PARALLELISM: Dividing input data into independent sets and processing simultaneously.

Eg: 100 answer sheets are divided among 4 teachers with 25 each and each teacher checks complete paper

T_1	T_2	T_3	T_4
P_1	P_{26}	P_{51}	P_{76}
P_2	P_{27}	P_{52}	P_{77}
⋮	⋮	⋮	⋮
P_{25}	P_{50}	P_{75}	P_{100}

Let no. of jobs = n, time to do a job = p
and there are k processors

Time to complete n jobs by single processor = np

" " " " " " k " = $kq + \frac{np}{k}$
↳ Time to distribute jobs to each processor

So, Speed-up = $\frac{np}{kq + \frac{np}{k}} = \frac{k}{1 + \frac{k^2 q}{np}} \longrightarrow \propto k$ for $k^2 q \ll np$

Efficiency = $\frac{\text{Speedup}}{k}$

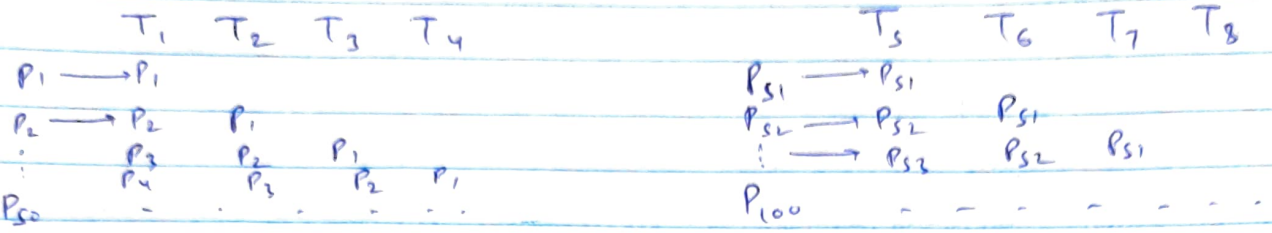
*) Advantages:

i) No synchroniz^n reqd
ii) No bubble problem
iii) Fault tolerant → If one fails, other can continue
iv) No commicat^n delay

*) Disadvantages:

i) Static assignment → pre-defined jobs, may have slow teach,
ii) Sets must be equal time taking & mutually independent
iii) Each processor must do all, no expertize
iv) Time to divide the jobs must be small
v) No of subsets << no. of jobs → Must hold

*) MIXED PARALLELISM : Both qualities of Temporal and Data parallelism



Example : Supercomputers like NEC-SX

*) CONDITIONS OF PARALLELISM : For ||ism, there must be 'independent' segments

Dependencies
- Data Dependence
- Control Dependence
- Resource Dependence

*) Dependence graph : Describes the relat^n b/w statements with nodes as statements & directed edges showing dependence, ordered relat^n

1) Data Dependence : Ordering relationship b/w statements

Data dependency is when a program statement refers to the data of a preceding statement.

Data dependencies

Flow dependence

o/p of S1 feeds as i/p to S2 ⇒ S2 flow dependent on S1

S1 → S2

Eg:

S1 : Load R1, A

S2 : Add R2, R1

Anti dependence

S2 follows S1 and o/p of S2 overlaps i/p of S1

S2 anti depend on S1

S1 ↛ S2

G:

S1 : Add R2, R1

S2 : Mov R1, R3

Output Dependence

If 2 statements produce same o/p variable

S1 ⟶ S2

G:

S1 : Load R1, A

S2 : Mov R1, R3

Both writing to R1

I/O Dependence

If same file is referenced by 2 I/O statements (Read or Write)

S1 —I/O→ S2

G:

S1 : Read (4), A[i]

S2 : Write (4), B[i]

Same tape unit 4

Unknown dependence

When none of these dependencies occur still can't run parallely

Due to
- Indirect addressing a(i[j])
- Loop variable missing a[]
- a[i] & a[j] in same loop
- Non-linear index variable

Random Access Machine

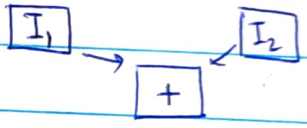
2) CONTROL DEPENDENCE : When order of execution of statements can't be determined before runtime. Eg: If condition (branch instruction)
Often prohibits parallelism.
Compilers are used to eliminate it & exploit the parallelism.
Eg:

```
for(i=0; i<n; i++)
{  a[i] = c[i];
   if(a[i]<0) a[i]=1;
}
```
Control- independent

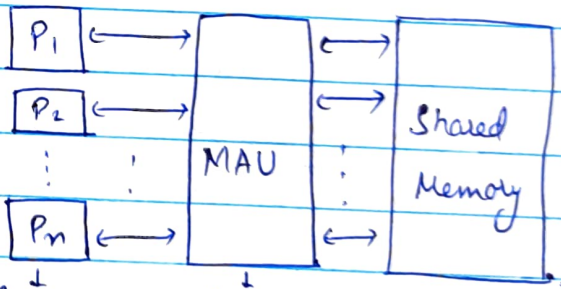
```
for(i=1; i<n; i++)
{  if(a[i-1]<0) a[i]=1;
}
```
Control dependent

3) RESOURCE DEPENDENCE : When conflicts in using shared resources like integer and float (ALUs), registers etc.
→ ALU conflicts are called ALU Dependence
→ Memory " " " Storage " → Load, Store
Eg: I_1 and I_2 both accessing addition unit, so ALU dependent

I_1 : A = B + C
I_2 : G = D + H



→ Parallel Random Access Machine
*) PRAM MODEL : → shared memory
→ Can be used in SIMD & MIMD both
→ It has each step of SIMD consist of :-
O(1) ← i) Read : Processors from memory (11dy)
O(1) ← ii) Compute : " perform computat" in their registers
O(1) ← iii) Write : " write (11dy) to memory



→ It is divided into 4 catagories :-
i) Exclusive Read Exclusive Write (EREW) : No sharing in reading or writing. Least concurrency
ii) Concurrent " ~~Concurrent~~ " (CREW) : Write is exclusive, ready is shared. Most common.
iii) Exclusive " Concurrent " (ERCW) : Write is shared, read exclusive. Not used greatly
iv) Concurrent " " " (CRCW) : Both shared. Most powerful for ||ism.

Interconnected Networks: No shared memory, processors communicate via interconnection network.

M → memory locat" } divided in
N → processors

Each processor in its local memory → M/N locat"

N/W Topology: Way of connect" b/w processors

→ Protocols to specify the value written in concurrent write:-

i) Priority CW: Highest priority processor writes
ii) Common CW: Write allowed only if all want to write same value.
iii) Arbitrary CW: Arbitrarily a processor succeeds in writing, others fail.
iv) Combining CW: A function of multiple values that want to be written is written in memory locat". Eg: ip1, ip2, ip3 wants to be written so f(ip1, ip2, ip3) is written.

) Garbage CW: A garbage value will be written

*) GRANULARITY: Also called grain size is the measure of amount of computation involved in a ~~S/w~~ software process. Simplest measure is count of no. of instructions in a grain (program segment). It determines the basic program segment chosen for II processing.

Grain size

Fine
Less no. of instructions in a grain.
A program is divided into large no. of very small sized grains

Medium
Medium no. of instructions in a grain

Coarse
Large no. of instructions in a grain.
A program is divided into small no. of large sized grains.

*) Latency: Time measure of the communication overhead incurred b/w machine subsystems.

Eg: Memory latency → Time reqd to access the memory
Synchronization " → " " for two processors to synchronize with each oth

*) Levels of Parallelism: Lower the level, finer the grain

↑ Communication demand & scheduling overhead	Level 5	Jobs or Programs	Coarse grain
	Level 4	Subprograms, job steps or related parts of a program	
	Level 3	Procedures, subroutines, tasks or coroutines	Medium grain
	Level 2	Non-recursive loops or unfolded iterations	Fine grain
	Level 1	Instruction or statement	

↑ Degree of II ism

For exploiting fine grain ||ism → optimized compiler → auto detect ||ism & translate source code to || form to be recognized by run-time system

1) Instruction Level: Lowest level, less than 20 instructions (grains contain), fine grain
Depending upon individual programs, fine grain ||ism varies from 2 to 1000s
Can be detected and exploited within processors.

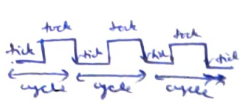
2) Loop Level: Iterative loop operations, contains less than 500 instructions
Most optimized to execute on || computers, pipeline can be used if "successive iter" are independent
Also fine grain. Recursive loops difficult to ||ize

3) Procedure Level: Medium grain ||ism (grain here), contains less than 2000 instruction
Detect" of ||ism is difficult, Dependence (inter-process) analysis important, Communicat" requirement < MIMD execution mode, SPMD → special case. Eg: Multitasking
Program (recording with compiler assistance)

4) Subprogram Level: Jobs steps / related subprograms, grain size in range of 10-100 thousand instructions. Overlapping job steps, ~~Eg~~:
Eg: Multiprogramming, ||ism exploited by designers (Algo designer) instead of compiles
Coarse/Medium grain ||ism → require suitably designed || programing language

5) Job or Program Level: || execut" of independent jobs in a || computer.
High grain size ~ Millions of instructions in a single program. Used in supercomputers having small no. of ↑ power processors, Coarse-grain ||ism
Handled by OS and program Loader, Eg: Time/Space sharing multiprocessors

*) COMMUNICATION LATENCY: Balance granularity (tradeoff) & latency → ↑ performance
Latencies attribute to machine architecture, implementing technology & communicat" pattern. Latency imposes a limiting factor on scalability of machine size. Eg: Memory latency ↑ with processor cycle time over the years
Latency in IPC also need to be minimized → also depend on comm" pattern
Eg: n ~~processor~~ (task) → $\frac{n(n-1)}{2}$ links → $O(n^2)$ → Limits no. of processors allowed in a large computer system
Frequent patterns → permut", broadcast, multicast etc.



Clk = tick-phase followed by tock-phase

Typical instruction : Fetch → Decode → Operand fetch → Execute → Memory reference

Processor

*) PROGRAM DECOMPOSITION TECHNIQUES : To introduce ||ism in program

Recursive Decomposition

→ General purpose

→ Divide & conquer strategy used

→ Problem → Sub problem, Sub problem, Sub problem, Sub problem → Convergence

Merged

Terminates when single element

→ Eg : Quick Sort

Data Decomposition

→ General purpose

→ Useful for algorithms operating on large data structures

→ i) Partition the i/p data

→ ii) This induces partioning of computat^n of tasks

→ Similar operations performed

Types

Partitioning o/p data

→ Each processor produces a portion of o/p

→ Independent of other i/p

Eg : Matrix Multiplicat^n

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Portion of o/p

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$

$C_{12} = A_{11}B_{12} + A_{12}B_{22}$

$C_{21} = A_{21}B_{11} + A_{22}B_{21}$

$C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Partitioning i/p data

→ Each processor is given a portion of i/p

→ Eg : SPMD

array add^n, Sorting

Partitioning intermediate data

→ Partially computed data is partitioned

*) Owner-Computes Rule : The process that owns a part of data, performs all computat^n related to it

For i/p partitioning → Process computes all computat^n involving that data

For o/p partitioning → Process computes all the o/p it is assigned to compute.

Exploratory Decomposition

→ Specific purpose

→ Computation involves searching a search-space

→ Program partitioned into small parts & each search concurrently

Eg : 15-Puzzle Problem

Speculative Decomposition

→ Specific purpose

→ Partition to a possible branch depending upon result from preceding comput^n

Eg : Topological Sort, Switch case

*) Performance Factor :-

*) Cycle Time : Time for one cycle (τ)

*) Clock rate (f) : No. of cycles per unit time in a processor clock. If Processor cycle time = τ then Clock rate = Inverse of τ

$$f = \frac{1}{\tau}$$

Book Pg 40 → Table

Average CPI = $\frac{\text{Total no. of cycles}}{\text{Total no. of instructions}}$

Memory cycle → Time for 1 memory reference
mem cycle = $k \cdot \tau$ → depends on cache speed, mem bus speed, mem processing tech, interface speed

★) Instruction Count (I_c): No. of machine instructions to be executed in the program. Determines program size. Diff. machine instructions take diff no. of clock cycles to execute.
Defines Program Size

★) Cycles per Instruction (CPI): Measure of time taken by an instruction to execute. No. of cycles reqd. by an instruction to execute.

★) Total No. of cycles reqd to execute a program = C

★) Performance Factor:

i) Execution Time (T): Time taken by a program to execute
or CPU Time

$$T = I_c \times CPI \times \tau$$

or

$$T = I_c \times (p + m \times k) \times \tau$$

or

$$T = C \times \tau$$

p = no. of processor cycles needed for instruction decode & execution

m = no. of memory references needed

k = ratio b/w memory cycle & process cycle

ii) MIPS rate (Million instructions per second): Processor speed in terms of MI/s. It varies wrt f, I_c and CPI

$$MIPS = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6} = \frac{f \times I_c}{C \times 10^6}$$

iii) Floating point operations per second: No. of floating point operations per unit time. Measured in mflops (megaflops), gigaflops, teraflops, petaflops.

1 flop = 1 floating point operation per second

iv) Throughput Rate:

W_s = System throughput rate = no. of programs a system can execute per unit time

W_p = Processor throughput rate = " " " " " " " " " "
based on only MIPS rate and I_c

Usually $W_s < W_p$ → system overheads I/O, OS overhead

$$W_p = \frac{f}{I_c \times CPI} = \frac{MIPS \times 10^6}{I_c}$$

System Attributes	I_c	p	m	k	τ
Instruction-set Architecture (ISA)	✓	✓			
Compiler technology	✓	✓	✓		
Processor Implement & control		✓			
Cache & Memory Hierarchy				✓	✓

System attributes influence Perform Factor

*) SHARED MEMORY PROGRAMMING : All processors share a common program and data stored in memory shared by all. Main program creates processes for each processor and provides them space and info to compute. After compute all processes rejoin. Main program executes after all created processes finish.

i) Fork : Process create statement

ii) Join : Invoking process needs results from invoked processes to continue

Process X

:

fork Y → creates Y

:

join Y ← Waits for Y to complete, then proceed

Process Y

end Y ← Both run simultaneously

To ensure exclusive write, we use lock & unlock :-

Process A

fork B

lock sum

Sum := Sum + A1

Unlock sum

join B

end A

Process B

lock sum

Sum := Sum + B1

Unlock sum

end B

↳ Produces Sum = (Sum + A1 + B1)

So

fork(1) Add-array (input: A[N/2+1:N], output: sum)

means fork & move to process 1, and start Add-array() with given parameters

Q: Program for Adding an array with 2-processor shared memory computer

Sol Main program for the 2-processor computer [Processor 0 (main), Processor 1 (slave)]

```
begin {main}
    global sum, Array A[1:N];
    for i := 1 to N do
        Read A[i]
    end for;
    Sum := 0;
    fork(1) Add-array (input: A[N/2+1:N], output: Sum);
    Add-array (input: A[1:N/2], output: Sum);
    join 1 ; // wait for Processor 1 to complete
    Write sum
end {main}
```

```
Add-array (input: A[P:M], output: Sum)
begin {Add-array}
    Sum_local := 0;
    for i := P to M do
        Sum_local := Sum_local + A[i]
    end for
    lock Sum ;
    Sum := Sum + Sum_local ;
    Unlock Sum;
end ; {Add-array}
```

*) Program for Adding an array with p-processor shared memory computer

Sol For Processor 0 :-

```
begin { main }
    global Sum, Array[1:N];
    for i := 1 to N do
        Read A[i]
    end for
    Sum := 0;
    increment := N div p;
    for j := 1 to (p-2) do
        fork(j) Add-array (input: A[j*increment+1 : (j+1)*increment], output: sum)
    end for;
    fork(p-1) Add-array (input: A[(p-1)*increment+1 : N], output: Sum);
    Add-array (input: A[1: increment], output: sum);
    for i := 1 to (p-1) do
        join i;
    end for;
    Write sum
end { main }
```

*) SPMD MODEL : Single Program Multiple Data – each instance of program working on its own data, and may follow different conditional branches or execute loops differently. Program written once → replicated many times

Barrier primitive makes process wait at this primitive for every other processes.

→ Program of SPMD add^n for array

Sol

```
begin
    global Sum, increment, Array A[1:N];
    if procid = 0 then
        for i := 1 to N do
            Read A[i]
        end for;
        Sum := 0;
        increment := N div p
    end if
    if procid ≠ p-1 then
        Add-array (input: A[procid*incent+1 : (procid+1)*incent], output: Sum)
    else Add-array (input: A[(p-1)*increment+1 : N], output: sum) end if
    end if
    barrier;
    if procid = 0 then
        Write sum
    end if
end
```

procid used for branch

*) **MESSAGE PASSING PROGRAMMING** : Processors send message to each other running parallely.

Asynchronous primitives : No order, no fixed time of message arrival

i) Send (destination processor address, variable name, size)

ii) Receive (source " " , " " , ")

Synchronous primitives : Order wise message delivery in given bounds. Message send-Block Receive-Idle Block

i) Blocked send → same argument as send → waits till message is delivered

i) Blocked receive → " " " receive → waits till specific message is arrived

Program for Two processor addition of array

Processor 0 :

```
Array A[1:N];
begin
  for i:= 1 to N do
     Read A[i]
  end for
  Send(1, A[N/2+1], N/2);
  Sum0 := 0;
  for i:= 1 to N/2 do
    Sum0 := Sum0 + A[i]
  end for
  Receive(1, sum1, 1);
  Sum := Sum0 + sum1
  Write Sum
end
```

Processor 1 :

```
Array B[1:N/2];
begin
  Receive(0, B[1], N/2);
  Sum1 := 0;
  for i:= 0 to N/2 do
    Sum1 := Sum1 + B[i]
  end for
  Send(0, Sum1, 1)
end
```

Addition using p-processors.

Processor 0 :

```
Array A[1:N], Sum[0:p-1];
begin
  for i:= 1 to N do
     Read A[i]
  end for;
  increment := N div p;
  last_inc = increment + N mod p;
  for j:= 1 to (p-2) do
     Send(j, A[j*increment+1], increment)
  end for
  Send(p-1, A[(p-1)*increment+1], last_inc);
  Sum[0] := 0;
  for i:= 1 to increment do
     sum[0] := sum[0] + A[i]
  end for
  for j := 1 to (p-1) do
     Receive(j, Sum[j], 1)
  end for;
  grand-sum := 0;
  for j := 0 to (p-1) do grand-sum := grand-sum + Sum[j]
  end for;
Write grand-sum end
```

```
Processor k = 1, 2 ... p-2
Array B [1 : increment];
begin
 Receive (0, B[1], increment);
 Sum[k] := 0 ;
 for i:=1 to increment do
     Sum[k] := Sum[k] + B[i]
 end for ;
 Send (0, Sum[k], 1)
end
```

```
Processor p-1
Array B[1 : last_inc];
begin
 Receive (0, B[1], last_inc);
 Sum[p-1] := 0 ;
 for i:=1 to last_inc do
     Sum[p-1] := Sum[p-1] + B[i]
 end for ;
 Send (0, Sum[p-1], 1)
end
```

*) SPMD Model - Message Passing Model : It is a loosely synchronous program. Conditional branching due to diff procid. Processors asynchronously execute same program with their own data.

→ Program for SPMD addition of array

```
begin
    Sum_local, Sum_rcvd, increment ;
    Array A[1:N];
    if procid = 0 then
        for i:=1 to N do
            Read A[i]
        end for
    end if
    if procid ≠ p-1 then
        increment := N div p
    else
        increment := N div p + N mod p
    end if
    if procid = 0 then
        for j:=1 to (p-2) do
            Send (j, A[j*increment+1], increment)
        end for
        Send (p-1, A[(p-1)*increment+1], increment+Nmodp)
    else
        Receive (0, A[1], increment)
    end if
    Sum_local := 0 ;
    for i:=1 to increment do
        Sum_local := Sum_local + A[i]
    end for
    if procid ≠ 0 then
        Send (0, sum_local, 1)
    else
        for j:=1 to p-1 do
            Receive (j, Sum_rcvd, 1);
            Sum_local := Sum_local + Sum_rcvd
        end for
        Write Sum_local
    end if
    end
```

MPI → Message Passing Interface
OpenMP → Open specifications for Multi-processing → shared memory parallel computing

Till Pg. 325

★) MPI (Message Passing Interface): It is a standard specification for a library of message passing functions. It specifies public-domain, platform-independent standard of MP functions → providing portability.
→ No feature specific to Hardware, OS or vendor
→ It is a library not a programming language
→ Can be called from FORTRAN 77, FORTRAN 95, C, C++ programs
→ Programs are compiled and then linked with MPI library
→ The design of MPI is based on :- (4 orthogonal concepts)
  i) message datatypes
  ii) Communicators
  iii) Communication Operations
  iv) Virtual topology → How ll computers are connected Eg: Mesh topology (Cartesian of)

★) Running MPI Program :
→ MPI assumes all processes are created when a ll program is loaded.
→ No ~~program~~ process terminated in middle of program execut^n
→ MPI_COMM_WORLD : default process group consisting of all ~~active~~ processes in a program
→ int MPI_INIT(int *argc, char ***argv) ⇒ first MPI call to initialize MPI environment
  (argc: no. of params, argv: actual params)
  → Must be called once
  → Multiple calls ⇒ erroneous
→ MPI_COMM_SIZE → to find out no. of processes in the program (n)
→ MPI_COMM_RANK → " " " rank of each process (0 to n-1)
→ MPI_FINALIZE → Terminates the MPI environment
Program on Pg. 303

★) Message → Message Buffer: The content of message
→ Message Envelope: Destinat^n of message

★) MPI_Send(&N, 1, MPI_INT, i, i, MPI_COMM_WORLD)
(Message Buffer: &N, 1, MPI_INT; Message envelope: i, i, MPI_COMM_WORLD)
- &N (Value sent) → Message address
- 1 → Message count
- MPI_INT → Message Datatype → supports heterogeneous computing → platform independent
- i → Destinat^n process id
- i → tag or msg type → integer used by programmer to label types of messages & restrict msg receipt^n
- MPI_COMM_WORLD → communicator → process group with a context

*) Message Buffer :

i) Message Address : Anything in sender's address space, refers to starting address of message buffer

ii) Message Count : No. of occurences of data items of the message datatype starting from message address

iii) Message Datatype : Heterogeneous computing → vendor, processor, OS independent
Eg: MPI-BYTE → 8 bits, MPI-PACKED, (Read from Book Eg. 205)
Derived datatypes →

*) Message Envelope :

i) Destinat" process id : Id of receiving process

ii) Message Tag : To order the messages

Eg:

Process X	Process Y
Send(M, 64, Y)	Recv(P, X, 64)
Send(N, 32, Y)	Recv(Q, X, 32)

We want to recieve P before Q

⟹

Process X	Process Y
send(M, 64, Y, tag1)	recv(P, X, 64, tag1)
send(N, 32, Y, tag2)	recv(Q, X, 32, tag2)

If N reaches earlier → buffered

iii) Communicator : Tags are allocated by user → mistakes prone,

Sol" ⇒ Content : Allocated at run-time by system in response to user requests & used for matching messages by machine

Eg : Process 0 :-
MPI-Send(m1, c1, MPI-INT, 1, tag1, comm1)
Parallel-mat-inv();
contains → MPI-Send(m2, c1, MPI-INT, 1, tag1, comm2)

Process 1 :-
MPI-Recv(m1, c1, MPI-INT, 0, tag1, comm1, ..)
parallel-mat-inv();
→ Might receive m2 here if no Comm

Communicator examples: MPI-COMM-WORLD, MPI-COMM-SELF → contains group of all processes + only the process it uses

*) MPI-RECV (Message address, Message count, Message Datatype, Dest' proc id, Tag, Communicator, Status)

→ Status: returns info on the data received, returns error code if any error like overflow occured

*) Types of Communicat"

- Point-to-Point
- Intra-Communicator → Broadcast, multicast
- Inter-Communicator

★) OPENMP : It is an API for programing in FORTRAN, C/C++ on shared memory ll computers

MPI → llize entire applicat" immideally

OpenMP → Incrementally llize the applicat" without major restructuring

OpenMP
- Compiler Directives
- Runtime Library Routine
- Environment Variables

→ Central entity is a thread, not a process

→ Creates multiple corporating threads running on multiple cores/processors

→ Start → Master Thread → Parallel Construct → Slave thread, Slave thread, Slave thread, Slave thread → Barrier (Join) → Master thread

Code inside → Parallel region → executed llly using SPMD

→ Master thread

Parallel region 1: 4 threads

Serial region

Parallel region 2: 5 threads

Team of threads

★) Running an OpenMP Program

→ A multi-threaded program where each thread is identified by its ID & master thread prints no. of threads present

→ Program on Pg. 318

→ llism in OpenMP is controlled by compiler directive

#pragma omp stuff (called sentinel)

a command or openMP directive

OpenMP Execution Model

→ Syntax of parallel construct :- #pragma omp parallel [clause [clause] ...]
{ structured block }

↓ ensures structured block executes llly

↳ Team of thread does llly (using SPMD)

↓ called parallel region

one entry & one exit point

one or multiple statements

→ omp_get_thread_num() → returns ID/rank of thread

→ omp_get_num_threads() → returns no. of threads in the team

→ #pragma omp barrier waits until all threads arrive at barrier

→ Variables declared in main → shared by all threads → shared scope

→ " " " structured block → only access to that thread → Private scope

→ 'private' clause changes variable scope from shared to Private for local changes in a block

Similarly 'shared' clause

Imp

*) Parallel Loops

→ # pragma omp parallel for → Creates a team of threads to execute following structured block which is a 'for' loop.
→ Loop is ||ized by dividing iterat^n of loop among threads
→ Loop-carried dependence should be absent (a[i] should not depend on a[i-1] or prev) as OpenMP compilers do not check this dependency
→ No ||ize for 'while' and 'do-while' in OpenMP convert to 'for' loop

→ # pragma omp for → Doesn't create team of threads
→ Partitions the iterat^n in for loop among threads in existing team

```
# pragma omp parallel
# pragma omp for
 for (...)
 { ...;
 }
```

equivalent to

```
# pragma omp parallel for
 for (...)
 { ...;
 }
```

*) Loop Scheduling: For distribut^n or mapping of loop iterat^n to thread, OpenMP supports scheduling strategy given by 'schedule' parameter.

→ schedule (static, chunk_size) → Static Scheduling: Assigns blocks of size chunk-size in Round-Robin to threads
Eg: <Thread 0: 0,1,2,9>
<Thread 1: 3,4,5>
<Thread 2: 6,7,8>
→ If comput^n per loop is not same → some thread may get overload & some get less load → Load imbalance
Solution: Dynamic Scheduling

→ schedule (dynamic, chunk-size) → Dynamic Scheduling: Assigns a new block of size chunk_size to thread as soon as it completes comput^n of previously assigned block
→ Overhead to keep track

→ schedule (guided, chunk_size) → Guided Scheduling → Similar to dynamic but chunk size is relative to no. of iterations left.

Chunk size = $\frac{\text{No. of iterations left}}{\text{No. of threads}}$

Eg: 100 iteration, 2 threads

Chunk sizes are 50, 25, 12, 6, 3, 1, 1

→ schedule (auto): Mapping decision is delegated to compiler and/or runtime system

★) Non-Iterative Constructs:

for construct → shares iteration of a loop across team → Data parallelism

sections construct → breaks work into separate independent sections to execute IIly → Temporal/Task IIism

```
# pragma omp sections
{
    # pragma omp section
    { structured block }   → independent of each other → executed IIly by diff threads
    # pragma omp section
    { structured block }   → Implicit synchroniz" at end of construct
    :
}
```

★) Synchronization Constructs: Protects critical sections (CS) or race condition as multiple threads IIly accessing shared data

1) # pragma omp critical → guarantees that structured block (CS) is executed by only 1 thread at a time
   { structured block }

2) # pragma omp atomic → Ensures single line of code is executed by only 1 thread at a time
   statement_expression
   ↓
   single line of code

   critical → protects all array elements
   atomic → " enclosing 1 array element

3) # pragma omp barrier → Synchronizes the threads at a specific point
It waits for all threads to arrive at barrier

4) nowait → Eliminates implicit synchronizat° occuring by default at the end of a parallel section
Eg: # pragma omp parallel for nowait

5) Lock functions
omp_init_lock() → initializes a lock variable
omp_set_lock() → set the lock
omp_unset_lock() → release the lock

6) # pragma omp flush [(list)] → Updates new values of elements in [(list)] to memory (flushes) so all threads can see correct values
→ No list → all thread visible variables update
→ OpenMP uses a relaxed memory consistency model

7) Reduction (operator : list) → reduces list of variables into one, using operator
→ neat way to combine results of pvt copies into a single result at the end of parallel region
i) By only 1 line addit°, Serial → Parallel code
ii) With reduct° clause, o/p
a) A Private copy of 'sum' is created for each thread
b) Initialized sum
c) At end, all copies are reduced to one by 'add' operator
iii) No Synchroniz° construct like 'critical' regd

Sequential code:

```
int A[MAX], i, sum = 0;
double avg;
for(i=0; i<MAX; i++)
{ sum = sum + A[i];
}
avg = sum / MAX;
```

Parallelized Code:

```
int A[MAX], i, sum=0;
double avg;
# pragma omp parallel for reduction(+:sum)
for (i=0; i<MAX; i++)
{   sum = sum + A[i];
}
    avg = sum / MAX
```

*) Environment Variables: To control II execut^n environment

i) OMP_SCHEDULE → alters 'schedule' clause when schedule(runtime) is specified in 'for' or 'parallel for' directive

ii) OMP_NUM_THREADS → Sets max no. of threads to use in a II region

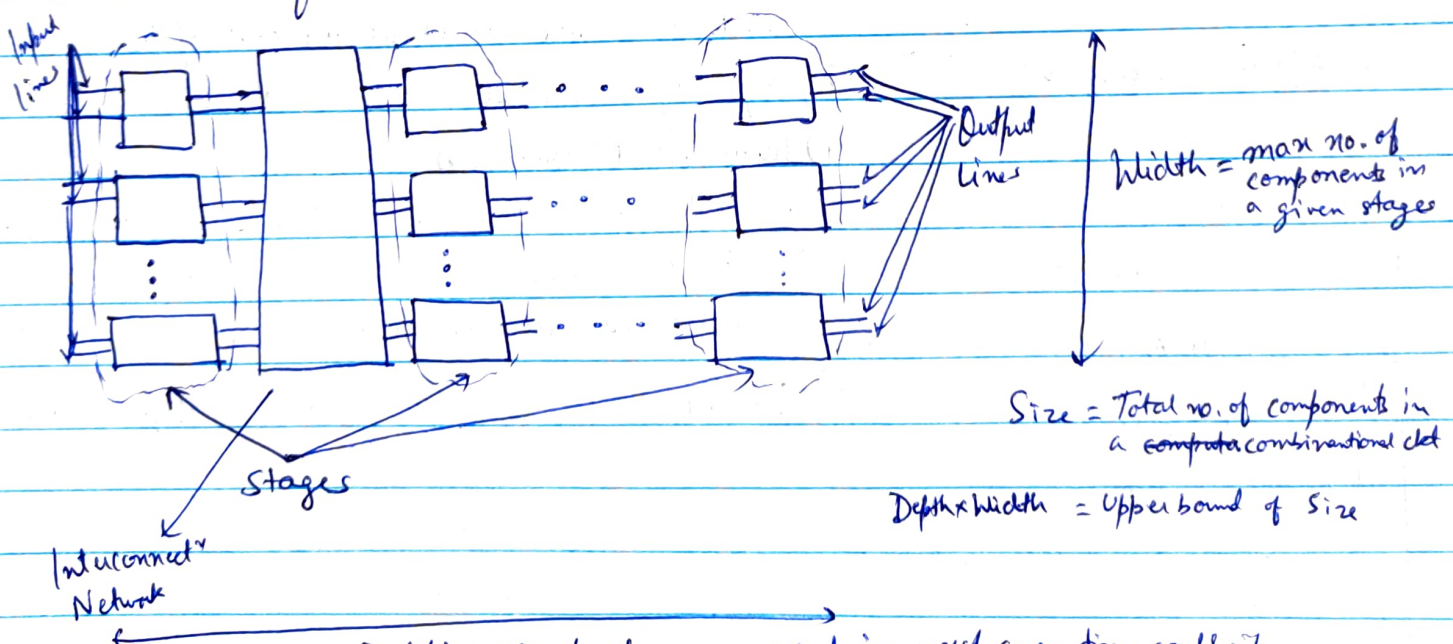
iii) OMP_DYNAMIC → Specifies whether runtime can adjust the no. of threads in II region

iv) OMP_STACKSIZE → Controls size of stack for slave threads

*) OpenMP 3.0 : 'task' declarative parallelizes recursive functions & while loops
'collapse' clause ,, ,, nested loops

*) OpenMP 4.0 (2013): SIMD constructs to support SIMD or vector-level IIism to exploit full potential of today's multicore architectures.

*) Combinational Circuit: It is a family of models of computation.
It has multiple components (processors) arranged in columns → Stages
No. of i/p lines to a component = Fan-in
,, ,, o/p ,, of ,, ,, = Fan-out
Each unit performs ALU operation in 'one time unit' & produces output.
Component is active only after receiving all necessary i/p
It has no feedback connection



Analysis of II → Running time → best case, worst case, avg case, asymptotic bounds

Speedup = Running time of best sequential Algo / Running time of given II algo

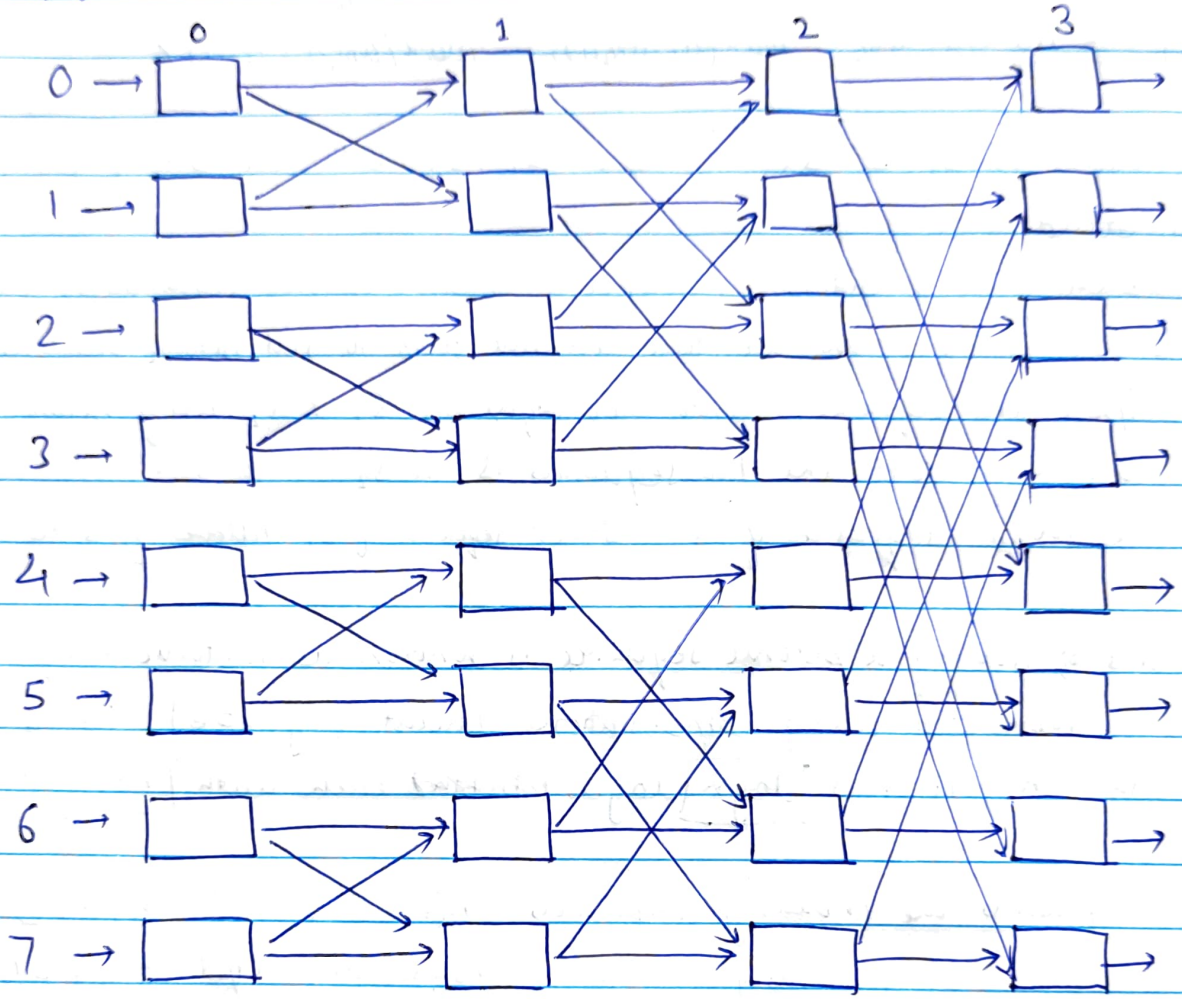
Algo → No. of processors → P(n) for i/p size n

→ Cost = Running time × No. of processors Efficiency = Worst case running time of best sequential algo / Cost of II algo

★) Butterfly circuit : Also a combinational circuit with n i/p and n o/p

Depth = 1 + log n and Width = n , Size = n(1 + log n) = n + n log n

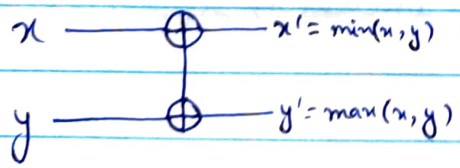
For n = 8 :-



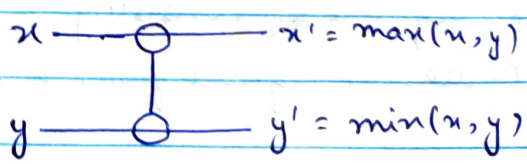
Butterfly Circuit

★) PARALLEL SORTING :

i) Using Combinational Circuit : Uses comparator :



Increasing comparator



Decreasing comparator

★) Bitonic Sorting Network : $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ is a bitonic sequence if :-

a) $\langle a_0, a_1, \ldots a_i \rangle$ is monotonically increasing & $\langle a_{i+1}, \ldots a_{n-1} \rangle$ is monotonically decreasing for some $0 \le i \le n-1$

b) OR The sequence is a cyclic shift of (a) Eg: <5,7,8,9,6,4,2,1> or <5,7,9,6,4,2,1,3>

Analysis of Bitonic Sort : If Sorting network, n = width, $\log n$ stages, last stage → (+)BM(n) → depth of $\log(n)$ all other stages sort $n/2$ element. So..

depth ← $d(n) = d(n/2) + \log n \Rightarrow d(n) = ((\log n)^2 + \log n)/2 = O(\log^2 n)$

Let $S = \langle a_0, a_1, \ldots a_{n-1} \rangle$ → Bitonic sequence where $a_0 \le a_1 \le \ldots \le a_{n/2-1}$ and $a_{n/2} \ge a_{n/2+1} \ge \ldots \ge a_{n-1}$

Consider two subsequences :-

$S_1 = \langle \min(a_0, a_{n/2}), \min(a_1, a_{n/2+1}), \ldots \min(a_{n/2-1}, a_{n-1}) \rangle$

$S_2 = \langle \max(a_0, a_{n/2}), \max(a_1, a_{n/2+1}), \ldots \max(a_{n/2-1}, a_{n-1}) \rangle$

Here,

All values in S_1 ≤ Values in S_2 and S_1, S_2 both are bitonic sequences

~~This is called~~

This splitting of a bitonic sequence into two bitonic sequences is called BITONIC SPLIT. We use this recursive algo to sort bitonic sequence :-

i) Split the sequence S into two sequences S_1 & S_2 using bitonic split

ii) Recursively sort the two sequences S_1 & S_2

iii) The sorted sequence of S = sorted sequence of S_1 followed by sorted seq of S_2

This process of sorting a bitonic sequence is known as Bitonic Merging. This can be implemented using combinational circuit. (Pg. 256). For a sequence of length = n, it has $\log n$ layers ~~[illegible]~~ each with $n/2$ comparators

If all comparators are increasing comparators → ascending output → (+) BM(n) (circuit representation)

" " " " decreasing " → descending output → (−) BM (n)

Depth = $\log n$, Width = n

(+) BM(n) and (−) BM(n) are building blocks for combinational ckt for sorting.

i) Any sequence of only 2 numbers forms a bitonic sequence → Trivial

ii) A sequence ~~of~~ consisting of a monotonically increasing sequence & a monotonically decreasing sequence forms a bitonic sequence.

So, we start with a sequence of size 2 and incrementally construct larger bitonic sequences

If we have two bitonic sequences of length $n/2$ each → (+) BM (n/2) to one sequence, (−) BM (n/2) to other sequence

Concatenate these sequences → Bitonic sequence of n length

Once we have a bitonic sequence of length n, we (+) BM(n) to sort. (Pg. 257)

*) Combination ckt for Sorting by Merging: Splits sequence in two halves, sorts & merges

→ Odd even merging ckt

Sequence $S_1 = \langle x_1, x_2 \dots x_m \rangle$ (sorted) and $S_2 = \langle y_1, y_2, \dots y_m \rangle$ (sorted) can be merged to sorted sequence $S = \langle z_1, z_2, \dots z_{2m} \rangle$ where $m = 2^n$.

So to merge two sequences we follow (m, m) Odd-even merging (length of each i/p) :-

i) Using an $(m/2, m/2)$ merging ckt, merge odd-indexed elements of two sequences: $\langle x_1, x_3, \dots x_{m-1} \rangle$ and $\langle y_1, y_3, \dots y_{m-1} \rangle \Rightarrow \langle u_1, u_2, \dots u_m \rangle$

ii) Similarly merge even-indexed elements $\langle x_2, x_4, \dots x_m \rangle$ & $\langle y_2, y_4 \dots y_m \rangle \Rightarrow \langle v_1, v_2, \dots$

iii) Output $\langle z_1, z_2, \dots z_{2m} \rangle$ is given as:-

$z_1 = u_1$ and $z_{2m} = v_m$

$z_{2i} = \min(u_{i+1}, v_i)$ and $z_{2i+1} = \max(u_{i+1}, v_i)$ $\forall i = 1, 2, \dots$

Pg. 259 → diagram

*) Analysis: Width = m = input size, Depth: $d(m) = \begin{cases} 1 & ; m = 1 \\ d(m/2) + 1 & ; m > 1 \end{cases}$

Size:

$p(1) = 1 \quad ; m = 1$

$p(m) = 2p(m/2) + m - 1 \quad ; m > 1$

So: $\boxed{p(m) = 1 + m \log m}$

$\boxed{d(m) = 1 + \log m}$ → very fast

→ Odd even Merge sorting ckt: For input of size = n :-

i) Split the unsorted i/p $\langle a_1, a_2, \dots a_n \rangle$ into two unsorted sequence $\langle a_1, a_2, \dots a_{n/2} \rangle$ and $\langle a_{n/2+1}, a_{n/2+2}, \dots a_n \rangle$

ii) Recursively sort these sequences

iii) Merge the two sorted sequences using $(n/2, n/2)$ merging ckt to get sorted output

*) Analysis:

Width: $O(n/2)$

Depth: $d_s(n) = d_s(n/2) + \underbrace{d_m(n/2)}_{\text{merging depth}} \Rightarrow \boxed{d_s(n) = O(\log^2 n)}$ → Time reqd. for sorting

(d_s: sorting depth; $\log(n/2)$)

Size: $P_s(n) = 2P_s(n/2) + P_m(n/2)$

$\hookrightarrow \boxed{P_s(n) = O(n \log^2 n)}$

ii) Using PRAM Models : Based on the idea of sorting by enumerations. Each element finds its own posit^n in sorted sequence by comparing its value with all other elements.

Rank of element s_i = posit^n of s_i in sorted sequence = $r_i + 1$

r_i = no. of elements smaller than s_i $\quad R = \langle r_1, r_2, \ldots r_n \rangle$

*) CRCW Sorting : If we have n^2 processors and if more than 1 processor tries to write in a memory location, then sum of all is written there. So P_{ij} = Processor in i^{th} row & j^{th} column (Total n rows & n columns $\rightarrow n^2$ processors). So, P_{ij} tries to write 1 in r_i if $s_i > s_j$ or $(s_i = s_j$ & $i > j) \rightarrow$ [s_j is smaller than s_i]

So if we place s_i at posit^n $r_i + 1 \rightarrow$ sorted

Procedure for CRCW Sorting :

```
for i := 1 to n do in parallel
    for j := 1 to n do in parallel
        if s_i > s_j or (s_i = s_j and i > j) then
            P_ij writes 1 to r_i
        end if
    end for
for i := 1 to n do in parallel
    P_i,1 puts s_i in (r_i + 1) position of S
end for
```

Time → O(1)
Space → O(n²)
↓
Processors
Unrealistic write conflict resolution

*) CREW Sorting :

Here process P_i computes r_i on complet^n of n iterations

In j^{th} iteration, all processors read s_j

P_i increments r_i if $s_i > s_j$ or $(s_i = s_j$ and $i > j)$

So in the end, we will have r_i = no. of elements smaller than s_i

So place s_i at posit^n $r_i + 1 \rightarrow$ Sorted sequence

Time → O(n)
Space → O(n)
↓
n processors

Procedure :

```
for i := 1 to n do in parallel
    for j := 1 to n do
        if (s_i > s_j) or (s_i = s_j and i > j) then
```

P_i adds 1 to r_i

end if

end for

~~for i:=1 to n do in parallel~~

$P_{i,1}$ puts s_i in (r_i+1) position

end for

★) EREW Sorting :

Time : $O(n)$

Space : $O(n)$

To avoid concurrent read, allow read in cyclic order

In first iteration $P_1, P_2, \ldots P_n$ reads $s_1, s_2, \ldots s_n$

" second " ——— reads $s_2, s_3, \ldots s_n, s_1$ and so on

" j^{th} " ——— reads $s_j, s_{j+1}, \ldots s_{j-1}$ and update corresponding r values

Procedure :

for i := 1 to n do in parallel

for j := 0 to n-1 do

k := (i+j) mod n

if ($s_i > s_k$) or ($s_i = s_k$ and $i > k$) then

P_i adds 1 to r_i

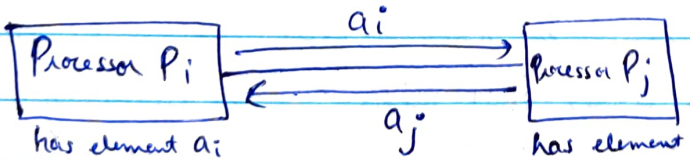
end if

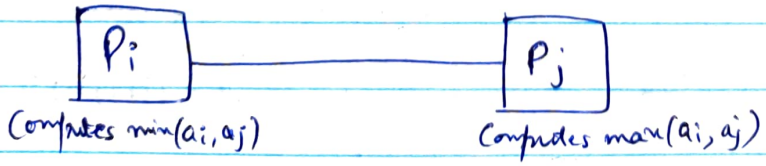
end for

$P_{i,1}$ puts s_i in (r_i+1) position

end for

ii) Sorting on Interconnection Networks





Parallel Compare Exchange Operation

Sorting on a Linear array

- Odd-even transposition sort and processors are connected to form a linear array
- Based on bubble sort

*) ODD EVEN TRANSPOSITION SORT: Time: $O(n^2)$

Let input be $\langle a_1, a_2, \ldots a_n \rangle$ unsorted.

i) Odd phase: $(a_1, a_2), (a_3, a_4), \ldots (a_{n-1}, a_n)$ compared & swapped if out of order

ii) Even phase: $(a_2, a_3), (a_4, a_5), \ldots (a_{n-2}, a_{n-1})$ " " " "

[a_1 & a_n not compared]

↳ Alternately applied

After $n/2$ odd & $n/2$ even phases → Sorted (Total n phases)

So in a linear array, P_i has element a_i

In odd phase, P_i does a compare-exchange with its right neighbour if i = odd

" even " , " " " " " " " " " " if i = even

Procedure

```
for i := 1 to n do
  if i is odd then
    for k := 1, 3, ..., 2⌊n/2⌋ - 1 do in parallel
      compare-exchange (Pk, Pk+1)
    end for
  else
    for k := 2, 4, ... 2⌊(n-1)/2⌋ do in parallel
      compare-exchange (Pk, Pk+1)
    end for
  end if
end for
```

*) Sorting on a Hypercube: Hypercube network can be used to optimize existing algorithms like Bitonic sort Algo

In bitonic sort let input lines be numbered 0000, 0001, ... 1110, 1111 (Binary)

Comparison is done b/w 2 wires that differ in 1 bit only

In a hypercube, such wires are neighbours. Thus each i/p is mapped to corresponding processor

Time Complexity still $O(n^2)$

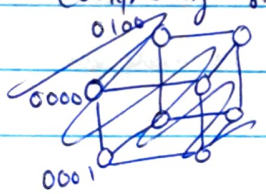


Diagram on Pg. 266

```
for i := 0 to d-1 do (n = 2^d, d = dimension of hypercube)
  for j := i down to 0 do
    for each Pk such that (i+1)st bit of k ≠ jth bit of k do in parallel
      compare-exchange (Pk, Pk(j))
    end for
  end for
```

↳ flipped bit at j^{th} position

*) MATRIX OPERATIONS ON PRAM MODELS :

i) **Matrix Multiplication** : If A is $m \times n$ and B is $n \times k$ then C is $m \times k$

$A \times B = C$

$$c_{ij} = \sum_{s=1}^{n} a_{is} * b_{sj} \quad (1 \le i \le m,\ 1 \le j \le k)$$

Sequential algorithm runs in $O(n^3)$ time, Strassen Algo: $O(n^{2.81})$ for $n \times n$ matrix

But n^2 values are to be produced, $O(n^\alpha)_{2 \le \alpha < 3}$

So min complexity will be around $O(n^2)$

Assume A, B, C all are $n \times n$

Sequential procedure :

```
for i := 1 to n do
  for j := 1 to k do
    c_ij = 0
    for s := 1 to n do
      c_ij = c_ij + a_is * b_sj
    end for
  end for
end for
```

*) **CREW Matrix Multiplication :**

n^2 processors arranged in $n \times n$ 2D array

P_{ij} computes c_{ij} in o/p matrix C

(one inner product)

Overall time complexity = $O(n)$

Procedure:

```
for i := 1 to n do parallel
  for j := 1 to n do parallel
    c_ij := 0;
    for k := 1 to n do
      c_ij := c_ij + a_ik * b_kj
    end for
  end for
end for
```

*) **EREW Matrix Multiplication** : Assume in k^{th} iteration, P_{ij} selects the pair $(a_{i l_k}, b_{l_k j})$ for multiplicat^n where $1 \le l_k \le n$. So :-

i) $1 < l_k < n$ ii) $l_1, l_2, \ldots l_n$ are all distinct

iii) $a_{i l_k}$ & $b_{l_k j}$ can be read only by Processor P_{ij} in iterat^n k

So $l_k = ((i+j+k) \bmod n) + 1$ (As if P_{ij} & P_{ab} read together :- then either $i=a$ or $j=b$

So $(i+j) \bmod n = (a+b) \bmod n$

Procedure for EREW :

Time : $O(n)$
n^2 processors

```
for i := 1 to n do in parallel
    for j := 1 to n do in parallel
        cij := 0;
        for k := 1 to n do
            lk := (i+j+k) mod n + 1;
            cij = cij + ai,lk * blk,j;
        end for
    end for
end for
```

*) CRCW Matrix Multiplication

Time : $O(1)$
n^3 processors

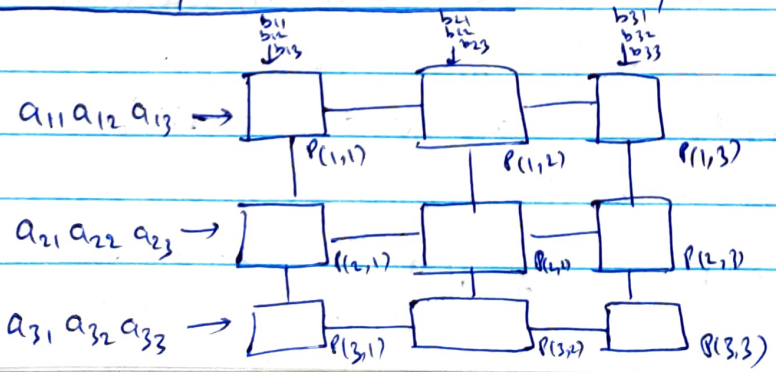
All write conflicts resolved by storing the sum of all values. Processors → conceptually → n×n×n array

Procedure

Cost of all three algorithm match with sequential

```
for i := 1 to n do in parallel
    for j := 1 to n do in parallel
        for s := 1 to n do in parallel
            cij = 0;
            cij = ais * bsj;
        end for
    end for
end for
```

Matrix Multiplication on a Mesh : n×n processors arranged in a mesh



When P_{ij} receives a & b :-

i) It multiplies them
ii) Adds the result to c_{ij}
iii) Sends 'a' to P_{ij+1} if $j \neq n$
iv) Sends 'b' to P_{i+1j} if $i \neq n$

Finally P_{ij} has c_{ij}

Parallelism property

i) Commutative : $P_i \parallel P_j \iff P_j \parallel P_i$

ii) Non-transitive : $P_i \parallel P_j$ and $P_j \parallel P_k \not\Rightarrow P_i \parallel P_k$

iii) Associative : $(P_i \parallel P_j) \parallel P_k \Rightarrow P_i \parallel (P_j \parallel P_k)$

Procedure

```
for i:=1 to n do in parallel
    for j:=1 to n do in parallel
        cij := 0
        while Pij receives two inputs a and b do
            cij := cij + a*b
            if i<n then send b to Pi+1j
            end if
            if j<n then send a to Pij+1
            end if
        end while
    end for
end for
```

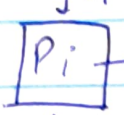
*) Analysis : P_{ij} receives its input after $(i-1)+(j-1)$ steps from beginning and it takes n steps to calculate c_{ij}. So c_{ij} is computed after $(i-1)+(j-1)+n$ steps.

So Time complexity $= O(n-1+n-1+n) = O(n)$

No. of processors $= O(n^2)$

So, Cost $= O(n^3)$

*) Bernstein's Condition: To check if two processes can execute parallely

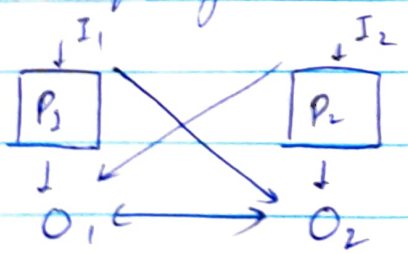


I_i → Input set or Read set or Domain of P_i

P_i → Process P_i

O_i → Output Set or Write Set or Range of P_i

Two processes P_1 and P_2 can execute in $\parallel$ if they are independent & do not create confusing results $\therefore (P_1 \parallel P_2)$ if :-



$I_1 \cap O_2 \neq \phi$ → Anti-independent

$I_2 \cap O_1 \neq \phi$ → Flow-independent

$O_1 \cap O_2 = \phi$ → O/p-independent

★) Pipeline Ideal Conditions :

i) Each instruct^n can be broken into independent tasks each taking equal time.
ii) All instruct^n in order → no branch or jump (Locality in instruct^n execn^n)
iii) Successive iterat^n are independent i.e. i^{th} iteral^n is independent of $(i-1)^{th}$ or previous
iv) Sufficient resources are available readily

Delays in pipeline ⇒ Pipeline Hazards due to non-ideal conditions

★) HAZARDS

i) Structural Hazards

→ Delays due to resource constraints, non-availability of resource
→ one cycle takes longer than other
→ **Avoid:**
More H/W Resource
Compute delay beforehand & decide whether to tolerate or not

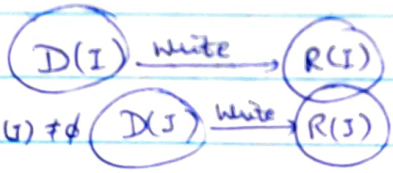
ii) Data Hazard

→ Delays due to data dependency
→ If instruction J is followed by I and D(J) = Domain of I & R(I) = Range of J

So :-

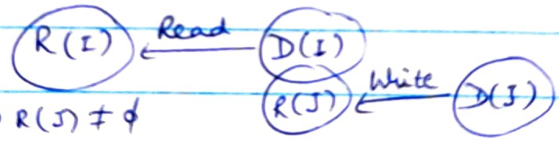
$R(I) \cap D(J) \neq \phi$

Read-after-Write (RAW) Hazard



$R(I) \cap R(J) \neq \phi$

Write-after-Write (WAW) Hazard



$D(I) \cap R(J) \neq \phi$

Write-after-Read (WAR) Hazard

→ **Avoidance**

i) Operand Forwarding : Special circuitry to give less time in a pipeline segment
ii) Code-reordering : Special s/w → H/W-dependent Compilers
iii) Stall-insertion : Insert NOP instruct^n to provide delay till operand is not read (efficiency still ↓)

iii) Control Hazard

→ Delays due to branch or control dependency
→ Jump instruct^n is found only in decode stage
→ **Avoidance**
i) Stall until branch outcome is known or fetch again
ii) Predict branch behaviour
→ Predict always taken
→ Predict never taken
→ Delay

(ILPP)

Unit-3 : Instruction Level Parallel Processing

→ How to use ILPP to design high performance PE (Processing elements)
→ Earliest use was Pipelining → widely used in RISC
→ After RISC, Superscalar processors → execute multiple instructions in 1 clk cycle
  ↳ uses ILPP by ↑ no. of arithmetic & functional units in PE
→ Further in VLIW one instruction word encodes more than one operation
→ Recent processors schedule no. of instructions on processors to execute parallelly
→ No. of transistors on a chip doubles every 24 months (Moore's Law)
→ We need to effectively use these multiple processors ('cores') on a single chip
→ We will discuss pipelining & superscalar here

★) **Pipelining of PEs** : Pipelining uses temporal parallelism to ↑ speed
effective method of ↑ execut^{n} speed given conditions (Ideal) :-

i) It is possible to break instruction into no. of independent tasks, each almost same time-taking to execute
ii) Locality in instruction execution ⇒ Instructions executed sequentially without JUMP
iii) Execution of one instruction helps execution of successive instruction & instructions are independent
iv) Sufficient resources are available in a processor for each instruction in pipeline

But these are rarely achieved!

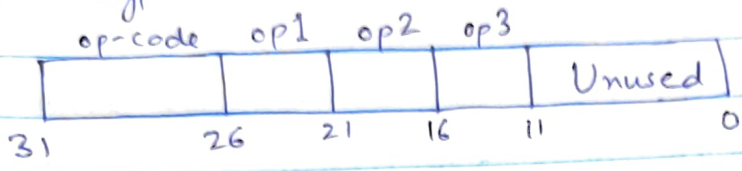
We will try to achieve pipelining under non-ideal conditions. Let's describe a hypothetical computer architectural model :- SMAC2P → Small Computer 2 Parallel

→ RISC Computer (Similar to SMAC2 (Small Computer 2))
→ Data memory : Data to be read / written stored here → Register file of 32 registers. General purpose reg to store operands & index value
→ Instruct^{n} " : Instruct^{n} " " executed " "
→ IMAR : Instruct^{n} memory address register
→ DMAR : Data " " "
→ IR : Data register of Instruction memory
→ MDR : " " " Data "
→ PC : Program counter → contains address of next instruct^{n} to execute
→ Machine is word addressable. A word = 32 bits

→ All instructions are of same length and relative posit^n of operands are fixed

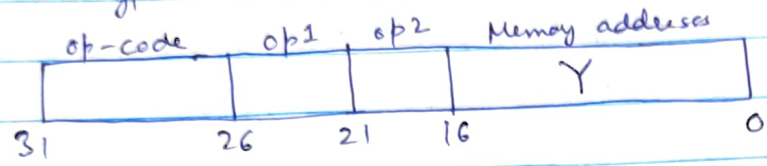
3 types of instructions :-

i) R-type



op1, op2, op3 } Address of registers in register file

ii) M-type

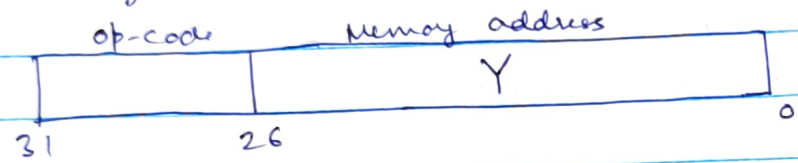


Y → address of operand stored in memory

op-code → 6 bit → $2^6 = 64$ operations possible

However we consider only a small set (Pg. 58) Rajaraman

iii) J-type



→ The only instructions which access memory → Load & Store instructions

Load: Reads words from data memory & stores it in specified register

Store: stores the contents of register in data memory

→ ALU carries out 1 arithmatic or 1 logic operat^n in 1 clk cycle

Stages in pipeline → Pg. 59 to 64 (See tables & diagrams only)

Buffer registers are needed to store the intermediate results so that they are not overwrited by next instruction

In ideal case, if total no. of instructions to be executed = m

& no. of clk cycles per instruction = n

No time taken in non-pipelining = mn cycles

& " " " Pipelining = $n + m - 1$ cycles

speedup due to pipeline = $\frac{mn}{n+m-1}$ if $m >> n$

$= \frac{n}{\frac{n}{m} + \frac{m-1}{m}} \approx n$

Due to this, each stage takes $(1+e)$ cycles

So, speedup = $\frac{mn}{n + (m-1)(1+e)} \approx \frac{n}{1+e}$

*) Delays in pipeline Execution : Delays in pipeline execution of instructions due to non-ideal conditions are called Pipeline Hazards

i) Delays due to resource constraints → Structural Hazards
ii) " " " data dependency b/w instructions → Data Hazard
iii) " " " branch instructions or control dependency → Control Hazard

1) Delays due to Resource Constraints : Suppose one instruction is reading data from memory and other wants to fetch instruction. If there is a common memory for data & instruction → only one instruction can execute. ↳ Not in SMIAC2P → No stalls ← Pipeline Stall
Forced waiting of an instruction in pipeline processing is called

Delay can also occur if execution of one instruction takes longer than others
Eg: A floating point division takes longer than integer addition.
Eg: (i+1) instruct^n takes 3 clk cycles. So (i+2) can't start at 5 → needs to wait till 7.

Clk cycles → / Instructions	1	2	3	4	5	6	7	8	9	10
i	FI	DE	EX	MEM	SR					
i+1		FI	DE	EX	EX	EX	MEM	SR		
i+2			FI	DE	X	X	EX	MEM	SR	
i+3				FI	DE	X	X	EX	MEM	SR

X = idle period or stall of an instruction due to resource constraint
How to avoid stall? → Speed up floating point to take only 1 clk cycle by using extra H/W
But let's see if the delay is too much to be avoided?
Let a fraction 'f' of total instruct^n are floating point taking n+k clk cycles each
Let total no. of instructions = m
So time to execute m instructions without pipelining = $m(1-f)n + mf(n+k)$ clk cycles
" " " " " with " = $n + (m-1)(1+kf)$ clk cycles

$$\text{Speedup} = \frac{m(1-f)n + mf(n+k)}{n + (m-1)(1+kf)}$$

$$\text{Speedup} = \frac{n + kf}{\frac{n}{m} + \frac{(m-1)}{m}(1+kf)} \approx \frac{n}{1+kf}$$ (Assuming $m \gg n$ & $n \gg kf$)

For $k=2$ and $f=0.1$ $\quad \text{Speedup} = \frac{n}{1+0.2} = 0.833n$

So loss of speedup = 16.6% ⟶ Not worth spending for extra H/W

Locking Pipeline: In some designs, whenever work cannot continue in a particular cycle, all stages except the one executing, are stalled → Locking

So :-

Clk cycle → / Instruction	1	2	3	4	5	6	7	8	9	10
i	FI	DE	EX	MEM	SR					
i+1		FI	DE	EX	EX	EX	MEM	SR		
i+2			FI	DE	X	X	EX	MEM	SR	
i+3				FI	X	X	DE	EX	MEM	SR

Commonly used in many machines

Here DE could have been done because only EX unit is busy But due to Locking → No work

Advantage of Locking: Ease of H/W Implementation

Ensures instructions executed in the order in which they are issued

Some machines do not lock & allow instructions without resource constraints to execute. They may get 'out of order' but if it is logically accepted by the program ⟶ No need to Lock.

2) **Delay Due to Data Dependency**: Delay might be produced because successive iterations are not always independent of one another. Eg:

Clk Cycle →	1	2	3	4	5	6	7	8	9	10
(R3←R1+R2) ADD R1, R2, R3	FI	DE	EX	MEM	SR					
(R5←R4×R3) MUL R3, R4, R5		FI	DE	X	X	EX	MEM	SR		
(R6←R7−R2) SUB R7, R2, R6			FI	DE	EX	MEM	SR			
(R3←R3+1) INC R3				FI	DE	X	EX	MEM	SR	

Stall due to data dependency → Data Hazard

ALU is busy

Here, third instruct^n completes before the second one. This is called 'out-of-order completion' → maybe unacceptable under some situations.
To avoid this we can use locking :- → preserves order but takes 1 more cycle

Clk cycle →	1	2	3	4	5	6	7	8	9	10
ADD R1, R2, R3	FI	DE	EX	MEM	SR					
MUL R3, R4, R5		FI	DE	X	X	EX	MEM	SR		
SUB R7, R2, R6			FI	X	X	DE	EX	MEM	SR	
INC R3				X	X	FI	DE	EX	MEM	SR

How to avoid delay due to data dependency?

i) H/W Method → Register forwarding → Consider ADD R1, R2, R3
after EX step we will have the result of R1+R2 stored in buffer register B3.
B3 ← B1 op B2 (B1 ← R1, B2 ← R2 (DE step))

So we provide a path from B3 to ALU i/p and bypass MEM & SR stages, so instead of waiting it to be written to R3, the result can be used via B3. This is register forwarding. It also requires H/W to detect that next instruct^n requires o/p of current instruction and should be fed back as i/p to ALU. Hardware should have facility to forward the o/p to all instructions that require it.

Eg: LD R2, R3, Y — C(R2) ← C(Y + C(R3))
ADD R1, R2, R3 — C(R3) ← C(R1) + C(R2)
MUL R4, R3, R1 — C(R1) ← C(R4) * C(R3)
SUB R7, R8, R9 — C(R9) ← C(R7) − C(R8)

Clk cycles →	1	2	3	4	5	6	7	8	9	10
LD R2, R3, Y	FI	DE	EX	MEM	SR					
ADD R1, R2, R3		FI	DE	X	X	EX	MEM	SR		
MUL R4, R3, R1			FI	DE	X	X	EX	MEM	SR	
SUB R7, R8, R9				FI	DE	X	X	EX	MEM	SR

Using register forwarding

Here we can see that SUB instruction is independent of above instructions but still it is delayed so much → Solution is software scheduling

ii) Software scheduling → Though H/W register forwarding reduced delay, it has not eliminated it completly. So we can re-order some independent instructions to reduce delays without changing the meanig of program It can further help to reduce or in some cases eliminate delays. For eg. if we reorder the instructions :-

```
LD   R2, R3, Y
SUB  R7, R8, R9
ADD  R1, R2, R3
MUL  R4, R3, R1
```

Clk cycle →	1	2	3	4	5	6	7	8	9
LD R2,R3,Y	FI	DE	EX	MEM	SR				
SUB R7,R8,R9		FI	DE	EX	MEM	SR			
ADD R1,R2,R3			FI	DE	X	EX	MEM	SR	
MUL R4,R3,R1				FI	DE	X	EX	MEM	SR

R2 not loaded → EX is busy

↳ Can be removed by adding more independent instructions in b/w

3) Delay due to Branch Instruction (disrupts normal flow of control): Last & most important non-ideal condition → Branch

If an instruction is branch instruction (known after DE stage)

↓ Next instruction will be

- Next sequential instruction (Branch not taken)
- Instruction specified in branch (Branch taken)

In SMAC2P, JMP → unconditional jump → jump address known at DE stage

JMI, JEQ, BCT → conditional ,,

- jump address known after DE
- jump address known only after EX stage

But we have defined PC set in stage 4 for all → MEM

So the next instruction to be executed will be available only after MEM stage.
Eg: (i+1) instruction is branch. But it will be known only after DE stage.
Meanwhile (i+2)th is fetched. So, it needs to be stalled till MEM of (i+1)
DE of (i+2)

If Branch not taken → DE of i+2 → delay of 2 cycles
" " taken → Fetch will be for new branch instruction
So later this instruction will be fetched again → delay of 3 cycles

Clk cycle → / Instruction ↓	1	2	3	4	5	6	7	8	9	10
i	FI	DE	EX	MEM	SR					
i+1 branch		FI	DE	EX	MEM	SR				
i+2			FI	X	X	FI	DE	EX	MEM	SR

If branch taken

Now if maximum ideal speedup of pipeline = 5
& let % of unconditional branches in a set of programs = 5 %
" " " conditional " " " = 15 %
" " " " " taken in program = 80 %

So, Ideal no. of cycles per instruction = 1
Avg delay due to unconditional branches = 3 x 5 % = 3 x 0.05 = 0.15
" " " " " " = Delay due to taken branch + " " " non-" "

So speedup with branches = $\frac{5}{1 + 0.15 + 0.42}$
≈ 3.18

= 3 x (15% x 80%) + 2 x (15% x 20%)
= 0.36 + 0.06
= 0.42

% loss of speedup = 36.4 % → High
Need to resolve it

There can be H/W methods & S/W method to reduce branch delays.

H/W Method to reduce branch delay

In SMAC2P, The branch address of JMP & JMI is available at the end of DE stage. But address of JEQ & BCT " " only after EX stage.
If we add an extra ALU in DE stage to calculate B1 - B2, then all addresses will be available after DE stage.

This way delay = 1 cycle if branch taken
= 0 cycle if " not taken

Now if we calculate the speedup :-

Avg delay cycles = $1 \times 0.05 + 1 \times 0.15 \times 0.8 + 0 \cdot 0.15 \times 0.2$
= $0.05 + 0.12$
= 0.17

So speedup = $5/1.17 \approx 4.27$

and % loss of speedup = 14.6% (from 36.4%)

So by adding H/W we get a gain of 22% → H/W is worth it because of less instructions

Adding this H/W in SMAC2P was not expensive But in commercial processors there are large no. of branch instructions → Expensive to add H/W (varied)

Some other technologies which are more cost-effective in certain processors are :-

i) Branch Prediction Buffer (BPB)
ii) Branch Target Buffer (BTB)

1) Branch Prediction Buffer (BPB)

→ less expensive H/W
→ less effective
→ small memory

Address of BPB memory

Address	Contents	Prediction bits
Low order bits of branch instr^n addr^s	Address where branch will jump	2 bits

Address of branch instruction → Branch address (Target)

Prediction → Not taken

At DE step → We will know that instruction is a branch instr^n or not

Use lower order bits as address to find in BPB memory (if branch)

00 → Strongly Not taken
01 → Weakly " "
10 → Weakly Taken
11 → Strongly " } Taken

if pred bits = 10, 11 → Branch taken → Predict Taken

if pred bits = 00, 01 → Branch not taken → Sequential instruction

Initially prediction bits = 00, following table is used to change the prediction bits :-

Current prediction bits	00	00	01	01	10	10	11	11
Branch taken?	Y	N	Y	N	Y	N	Y	N
New prediction bits	01	00	10	00	11	01	11	10

Experimentally, the predictions are 90% correct

With 1000 entries in BPB, estimated probability of finding a branch instruction in the buffer is 95%.

∴ Prob. of finding the branch address is at least $0.9 \times 0.95 = 0.855$

How many clk cycles do we gain?

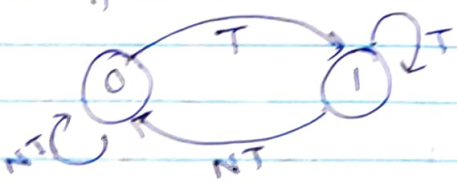
→ In case of SMAC2P with H/W modification → No gain as branch is ~~predicted~~ calculated at DE only

→ In case of SMAC2P without H/W, we gain two cycles (MEM → DE) if predictions are correct

→ Very useful in machines with slow computations

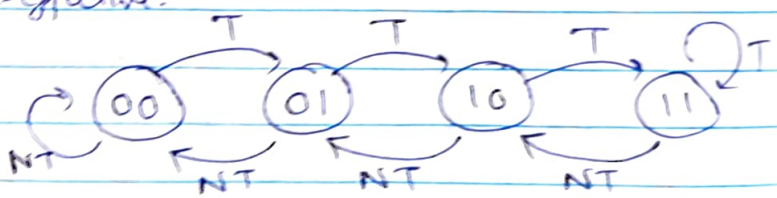
Why we need two bits for prediction and not one bit?

→ Single-bit predictor incorrectly predicts branches more often, particularly in most loops, in comparison to a 2-bit predictor. So 2-bit predictors are found to be more cost-effective.



0: NT, 1: T

1-bit predictor



00: Strongly Not taken

01: Weakly Not taken

10: Weakly taken

11: Strongly taken

BPB & BTB can be combined!

→ More expensive H/W
→ more effective
→ larger memory

2) **Branch target Buffer (BTB)**: Unlike BPB, it can be used at FI stage.

Address	Contents	Prediction bits
Address of branch instruction	Address where branch will jump	1 or 2 bits (optional)

Instructn address ← Address

↓ target address

At the end of DE stage → we know the instruction is branch

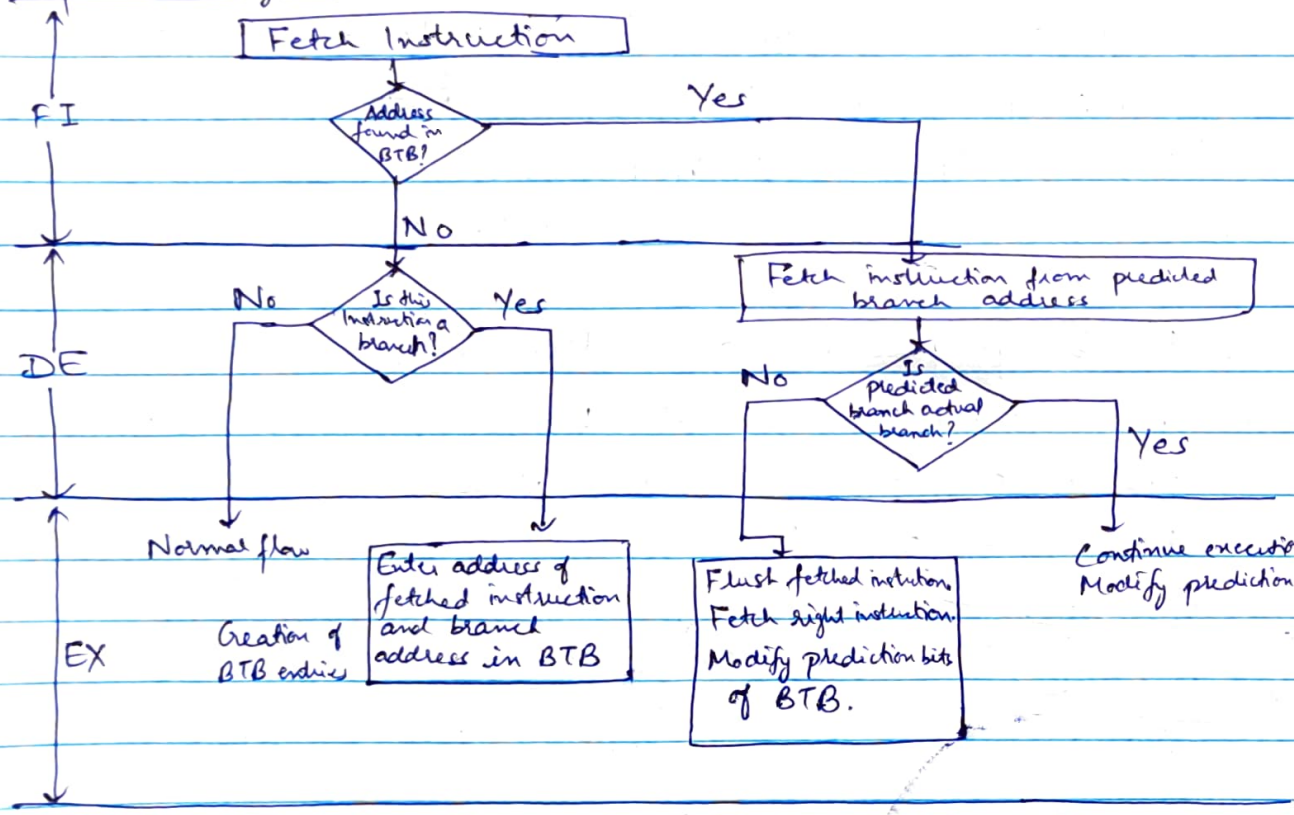
" " " " EX " → " " " target address of branch

↳ After this BTB entry is made & predictn initialized to 01 bits

Entry is made in a loop in 1st iteratn & used by later iterations

↳ saves 3 clk cycles

Pipeline stages



Here it is assumed that branch target address is found at DE → SMAC2P with H/W

If No H/W → Match of predicted & target address → possible after EX stage

BTB needs to be searched for instruction address → should not be too large ~1000 entries

Now assume Unconditional branch = 5% (0.05), Conditional branch = 15% (0.15), Taken branch = 80% = 0.8

Prob. that instruction found in BTB = 95% = 0.95

& " " prediction is correct = 90% = 0.9

So, by having BTB the avg delay cycle when unconditional branch found in BTB = 0 → 95%

" " " " " " NOT " " " = 3 → 5%

So Avg. delay due to unconditional branch = $0 \times 0.95 + 3 \times 0.05$

$= 0.15$

(No misprediction for unconditional branch)

Avg delay due to conditional branch if found in BTB = 0 → 95%

" " " " " " NOT " " " = (3×0.8) → Branch taken + (2×0.2) → Not taken

$= 2.8 \longrightarrow 5\%$

So, avg delay due to conditional branch = $0 \times 0.95 + 0.05 \times 2.8$

$= 0.14$

Avg delay due to misprediction of conditional branch = $0.1 \times 3 \times 0.95$

(0.1: misprediction (10%); 3: delay cycle for conditional branch; 0.95: present in BTB)

$= 0.285$

So,

Avg delay due to branch = $\underbrace{5\% \times 0.15}_{\text{Unconditional}} + \underbrace{15\% \times (0.14 + 0.285)}_{\text{Conditional}}$

$= 0.05 \times 0.15 + 0.15 \times (0.14 + 0.285)$

$= 0.0075 + 0.064$

$= 0.0715$

So. Speedup with BTB = $5 / 1 + 0.0715 = 4.666$

% loss of speedup = 6.8 %. (compare to 36.4%) → very good

BTB is extremly useful!

*) S/w Method to reduce delay due to branch: In absence of H/W methods, we need to rearrange the statements in such a way that statement following the branch statement (called delay slot) is always executed once it is fetched with compilers of program

Often successful

Eg1

Original program

ADD R4, R5, R6

ADD R1, R2, R3

JMP X

DELAY SLOT

X

Rearranged Program

ADD R4, R5, R6

JMP X

ADD R1, R2, R3 → Allowed to compute while JMP is decoded if it do not distrub program meaning 1 Else NOP inserted

X

Eg:2

Y: ADD R4, R5, R6

ADD R1, R2, R3

JMI Y

Y+1: ADD R1, R2, R3

JMI Y+1

ADD R4, R5, R6 Delay slot → executed while JMI decoding

Eg:3

ADD R4, R5, R6

JMI Z

Z: ADD R1, R2, R3

ADD R4, R5, R6

JMI Z+1

ADD R1, R2, R3

Z+1:

Eg: Loop unrolling

*) SUPERPIPELINING: In pipelined processor we assume that all stages take same time of 1 clk cycle. But in practice, the DE & MEM stage takes less than 1 clk cycle. So we can allocate them half a cycle & one cycle to all other stages. Further we divide each clk cycle into two phases of diff resource requirements → then no resource conflicts even for same stage! This is called superpipelining. → speeds up execution, adopted in many commercial processors like MIPS R4000

EX1 & EX2 have diff resource req → No conflict

Superpipelined Processing → completes in 5 cycles

clk cycle →	1	2	3	4	5	6	7						
	FI1	FI2	DE	EX1	EX2	MEM1	MEM2	SR					
		FI1	FI2	DE	EX1	EX2	MEM1	MEM2	SR				
			FI1	FI2	DE	EX1	EX2	MEM1	MEM2	SR			

Pipelined processing → completes in 7 cycles

FI	DE	EX	MEM	SR		
	FI	DE	EX	MEM	SR	
		FI	DE	EX	MEM	SR

★) **SUPERSCALAR PROCESSING**: Combines temporal parallelism of pipeline with data parallelism by issuing several instructions in one cycle. The instructions may not be in order but maximizes H/W resources. This is called Superscalar processing.

Superscalar processing with 2 instructions simultaneously :-

clk cycle →	1	2	3	4	5	6	7	→ 6 instructions completed in 7 seconds
	FI	DE	EX	MEM	SR			
	FI	DE	EX	MEM	SR			
		FI	DE	EX	MEM	SR		
		FI	DE	EX	MEM	SR		
			FI	DE	EX	MEM	SR	
			FI	DE	EX	MEM	SR	

Here, in steady state → two instructions will be completed every cycle (Ideal case)

For successful superscalar processing :-

i) H/W should permit fetching several instructions simultaneously → several FI

ii) Several independent ports for Read/Write in data cache → several DE
If Instruction = 32 bit, 2 fetch together → 64 bit path from Instruction cache reqd & 2 IR reqd

iii) Multiple EX units reqd to avoid resource conflicts → several EX
Floating pt + Integer add^{n}

Assume two integer execution units & one floating pt. unit & superscalar with 2 instruction fetched together :-

	Instruction	No. of cycles for execⁿ	ALU needed.
I1	R1 ← R1 / R5	2	INT
I2	R3 ← R1 + R2	1	INT
I3	R2 ← R5 + 3	1	INT
I4	R7 ← R1 - R11	1	INT
I5	R6 ← R4 × R8	2	FLOATING PT.
I6	R5 ← R1 + 6	1	INT
I7	R1 ← R2 + 1	1	INT
I8	R10 ← R9 × R8	2	FLOATING PT.

As I1 → I2

Cycles	1	2	3	4	5	6	7
	FI	DE	EX1	EX2	MEM	SR	
	FI	DE			EX1	MEM	SR
		FI	DE	EX1	MEM	SR	
		FI	DE		EX1	MEM	SR

I4 ← I1

Idle cycle

To take care of dependencies :-

Cycles →	1	2	3	4	5	6	7	8	9	10
I1	FI	DE	EX1	EX2	MEM	SR				
I2	FI	DE			EX1	MEM	SR			
I3		FI	DE			EX1	MEM	SR		
I4		FI	DE		EX1	MEM	SR			
I5			FI	DE	EXF	EXF	MEM	SR		
I6			FI	DE			EX1	MEM	SR	
I7				FI	DE			EX1	MEM	SR
I8				FI	DE		EXF	EXF	MEM	SR

Instruction Cycle

I2 3,4 ⇒ I1 → I2 flow dependency (Read After Write (RAW) Hazard)

I3 4,5 ⇒ I2 ↛ I3 anti dependency (Write After Read (WAR) ——)

I4 4 ⇒ I1 → I4 flow dependency (RAW)

I6 5,6 ⇒ I1 → I6 → 5th cell (RAW) & I3 ↛ I6 6th cell (WAR)

I7 6,7 ⇒ I3 → I7 → 6th cell (RAW) & I6 ↛ I7 7th cell, I1 ↛ I7 (Output dependency / Write after Write Hazard WAW)

I8 6 ⇒ EXF is only one & is used by I5 → Resource constraint

*)) Register Scoreboarding : To keep track of dependencies, we use a separate register called Scoreboard. If a computer has 32 registers, then a 32-bit scoreboard indicates busy / free status of registers.

When an instruction uses a set of registers → their scoreboard bits set to 1

When execution is completed → scoreboard bits reset to 0

With scoreboard instructions can check if a register is free or not. Now, flow dependency delay can't be reduced but output and anti-dpnd delays can be reduced using register renaming.

We allocate extra registers for this.

Eg:

	Old	Register renamed
I1	R1 ← R1 / R5	R1 ← R1 / R5
I2	R3 ← R1 + R2	R3 ← R1 + R2
I3	R2 ← R5 + 3	R2N ← R5 + 3
I4	R7 ← R1 - R11	R7 ← R1 - R11
I5	R6 ← R4 x R8	R6 ← R4 x R8
I6	R5 ← R1 + 6	R5N ← R1 + 6
I7	R1 ← R2 + 1	R1N ← R2N + 1
I8	R10 ← R9 x R8	R10 ← R9 x R8

→ This removes anti-dependency & output-dependency
↓
Thus delays are also reduced
↓
All instructions complete in 9 cycles

Is it possible to reschedule issue of instructions to reduce delay? → Yes

We can use an instruction window containing several instructions. From this window, our system picks 2 instructions & schedules them to minimize time.

Larger window size → Better optimization

Eg: Window size = 4 ⇒ { I1, I2, I3, I4 }

I1 → I2 so let's swap it with I3

⇓ Picks

Instructions picked ← {I1, I3}

↓ Next cycle

⇒ { I2, I4, I5, I6 }

I1 → I4 → delay I4

⇓

{ I2, I5 }

↓ Next cycle

{ I4, I6, I7, I8 }

I6, I7 → INT unit busy

{I6, I7}

⇓

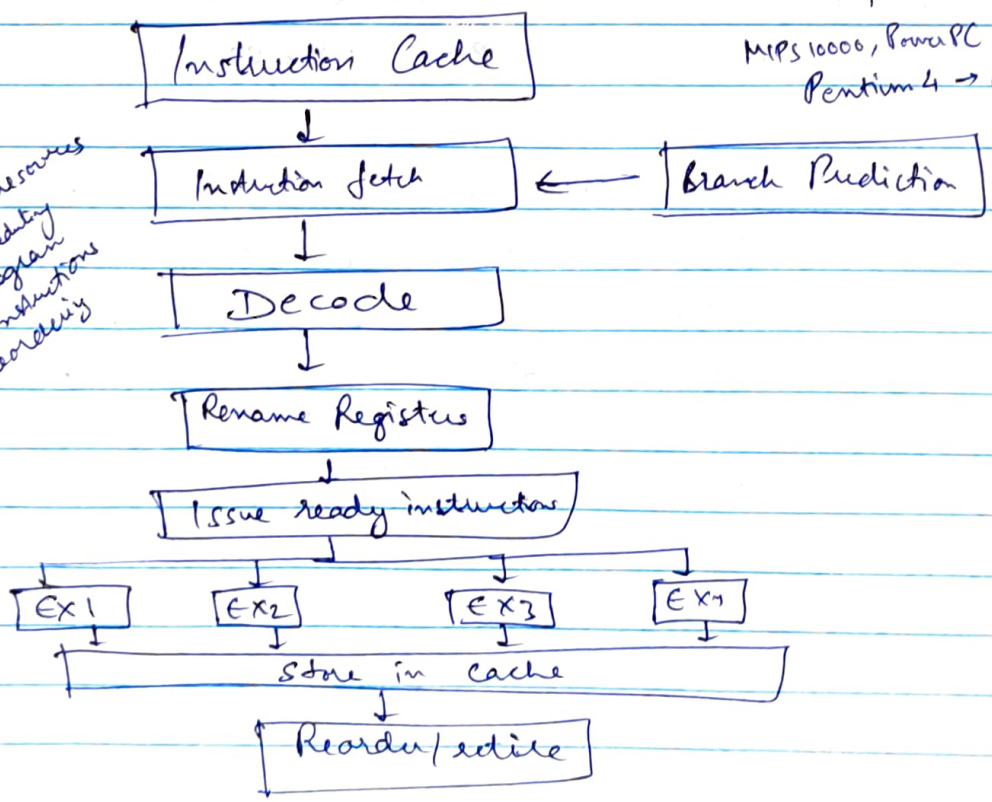
{I4, I8}

Rescheduling instructions in superscalar

Cycles	1	2	3	4	5	6	7	8	9	10
I1	FI	DE	EX1	EX2	MEM	SR				
I3	FI	DE	EX1	MEM	SR					
I2		FI	DE	~~X~~	EX1	MEM	SR			
I5		FI	DE	EXF	EXF	MEM	SR			
I4			FI	DE	EX1	MEM	SR			
I8			FI	DE	EXF	EXF	MEM	SR		
I6				FI	DE	EX1	MEM	SR		
I7				FI	DE	EX1	MEM	SR		

Now I3 completes before I1 and I2. Now due to branch prediction there may be some other instructions that need to be executed & these instructions might be abandoned. Thus it is incorrect to update memory with these instructions. Thus results must be stored in buffers, registers. These buffers will later be permanently committed to appropriate storage taking into account the sequential program correctness. This is called Committing or retiring an instruction.

Ex:-



Both RISC & CISC are superscalar

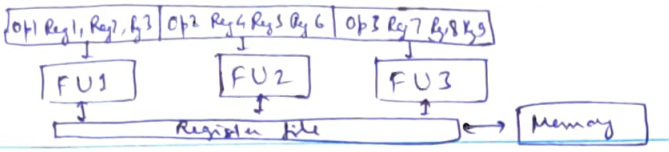
MIPS 10000, PowerPC → RISC superscalar

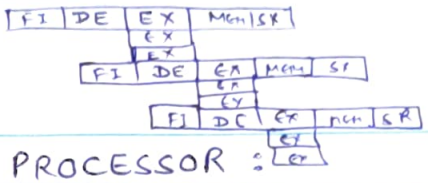
Pentium 4 → CISC superscalar

Many machines have been built to execute 4-5 instr per cycle

Machine Parallelism → H/W resources
Instruction level Parallelism → scheduling program instructions Reordering

Best Reducing → Loop Unrolling
Branch delays method





★) VERY LONG INSTRUCTION WORD (VLIW) PROCESSOR :

Problems with Superscalar → Need to duplicate IR, Decoder & ALU
→ Difficult to dynamically schedule instructions

Alternative method to use compilers to expose a sequence of instructions with no dependency & diff resource requirements because Compilers can take a more global view of program & rearrange code to better utilize the resources.
Here, a single word incorporates many independent operations → pipelined easily
Eg: 2 Integer operation, 2 floating pt. operation, → if not found NO-OP inserted
2 load/store " , 1 branch " → All packed in 1 word

→ The word size varies from 128 bits to 256 bits
→ It requires processor to have enough resources to execute all operations simultaneously
→ Challenging task : Maintaining enough parallelism in instructions to keep all units busy
→ Parallelism in programs is exposed by :-
i) Loop unrolling
ii) Scheduling instructions by examining program globally
iii) Trace scheduling : Predicting path taken by a branch operat" at compile time.
It uses some heuristics / hints given by programer.
If branches follow predicted path → straight line code

→ Challenges :-
i) Lack of sufficient instruct" level parallelism : For 1 FLOP (floating point operation) with 4 stages in pipeline, it requires 10 FLOP instruction to effectively use. + 2 Integer operations.
Instruct" level parallelism may not be this high.
ii) Difficulties in building H/W : Requires high memory & large register file. Extra read / write ports → requires large silicon area on chip.
iii) Inefficient use of bits in a very long instruction word : It is difficult to find this many independent instructions. Atleast half of each word is filled with NO-OP

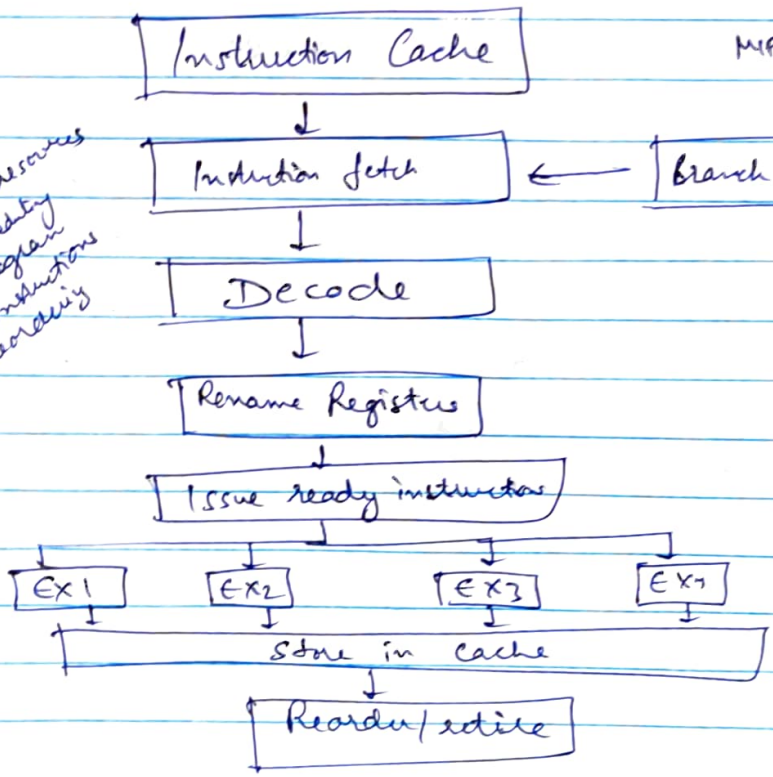
Thus VLIW is not very popular commercially.

Rescheduling instructions in superscalar

Cycles	1	2	3	4	5	6	7	8	9	10
I1	FI	DE	EX1	EX2	MEM	SR				
I3	FI	DE	EX1	MEM	SR					
I2		FI	DE	✕	EX1	MEM	SR			
I5		FI	DE	EXF	EXF	MEM	SR			
I4			FI	DE	EX1	MEM	SR			
I8			FI	DE	EXF	EXF	MEM	SR		
I6				FI	DE	EX1	MEM	SR		
I7				FI	DE	EX1	MEM	SR		

Now I3 completes before I1 and I2. Now due to branch prediction there may be some other instructions that need to be executed & these instructions might be abandoned. Thus it is incorrect to update memory with these instructions. Thus results must be stored in buffers (registers). These buffers will later be permanently committed to appropriate storage taking into account the sequential program correctness. This is called Committing or retiring an instruction.

Eg :-



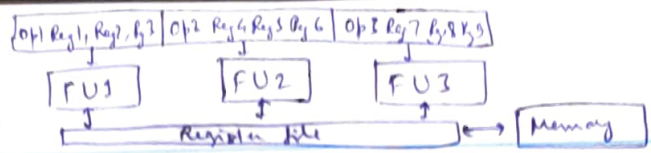
Both RISC & CISC are superscalar

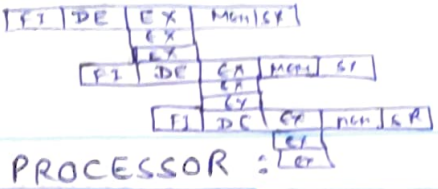
MIPS 10000, PowerPC → RISC superscalar
Pentium 4 → CISC superscalar

Many machines have been built to execute 4-5 instr per cycle

Machine parallelism → H/W resources
Instruction level Parallelism → scheduling program instructions Reordering

Best Reducing Branch delays method → Loop Unrolling





*) VERY LONG INSTRUCTION WORD (VLIW) PROCESSOR :

Problems with Superscalar → Need to duplicate IR, Decoder & ALU
→ Difficult to dynamically schedule instructions

Alternative method to use compilers to expose a sequence of instructions with no dependency & diff resource requirements because Compilers can take a more global view of program & rearrange code to better utilize the resources.
Here, a single word incorporates many independent operations → pipelined easily
Eg: 2 Integer operation, 2 floating pt. operation, → if not found NO-OP inserted
2 load/store " , 1 branch " → All packed in 1 word

→ The word size varies from 128 bits to 256 bits
→ It requires processor to have enough resources to execute all operations simultaneously
→ Challenging task : Maintaining enough parallelism in instructions to keep all units busy
→ Parallelism in programs is exposed by :-
i) Loop unrolling
ii) Scheduling instructions by examining program globally
iii) Trace scheduling : Predicting path taken by a branch operat^n at compile time.
It uses some heuristics / hints given by programer.
If branches follow predicted path → straight line code

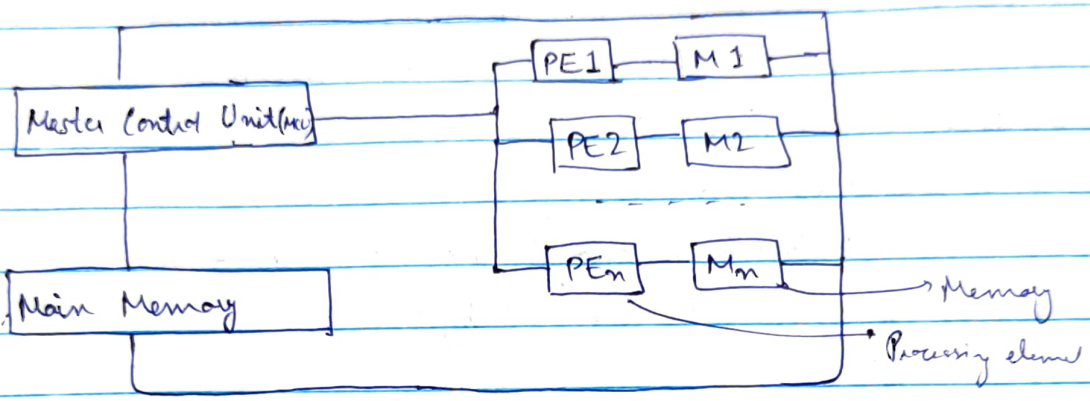
→ Challenges :-
i) Lack of sufficient instruct^n level parallelism : For 1 FLOP (floating point operation) with 4 stages in pipeline, it requires 10 FLOP instruction to effectively use. → 2 Integer operations.
Instruct^n level parallelism may not be this high.
ii) Difficulties in building H/W : Requires high memory & large register file. Extra read / write ports → requires large silicon area on chip.
iii) Inefficient use of bits in a very long instruction word : It is difficult to find this many independent instructions. Atleast half of each word is filled with NO-OP

Thus VLIW is not very popular commercially.

Scalar instruction → handles one piece of info at a time → like superscalar
Vector " → handles multiple piece of info at a time → like array / vector processors

*) Array Processors : Performs computations on large array of data.
There are two types of array processors :-
i) Attached array processor → Not in syllabus → auxillary processor is attached to ↑ comput^n speed
ii) SIMD Array Processor

→ SIMD Array Processor :
i) It has multiple process unit operating in parallel
ii) It also manipulate vectors but internal organizat^n is diff from attached array processor



→ PEs are sync together to perform same operat^n under control of Master Control Unit
→ Thus Master Control provides single instruction that is applied to Multiple data (SIMD)
→ Each PE includes :-
i) ALU ii) Floating pt. arithmatic unit iii) Working registers

→ MCU decodes the instruction & determines how to execute it
→ If instruction is scalar/program control instruction → executed within MCU
→ Main memory stores the program, each PE uses operands stored in their local memory

Advantages of Array Processors :
i) Increases overall instruction processing speed
ii) Improves overall capacity of system as most Array Processors operate async to Comp system
iii) Array processors have their own local memory, thus provide extra memory for systems with low memory.
iv) Exploits significant level of data parallelism
v) More energy efficient than MIMD → One fetch for all data
vi) Higher speed up than MIMD (2x)

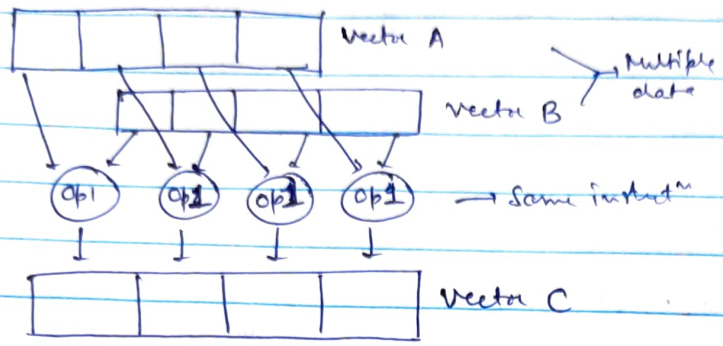
Scalar operat^{n} (1 operat^{n})

add R3, R1, R2

Vector operat^{n} (n operat^{n})

add VV V3, V1, V2

→ SIMD Units : The H/W Units performing operations parallely
It receives two vectors as input, each with a set of operands
It perform same operat^{n} on both set of operands (one from each vector) & Outputs a vector with result



★) Vector Processors (VP) : It is a central processing unit that can perform the complete vector input in individual instruction.
→ Used in scientific & research computations → high power requirement
→ Scientific problems can be specified in terms of vectors or matrices → use vector processing

→ Features :

i) Vector is a structured set of elements. The elements are scalar instructions, size of vector is called its length
ii) In one clk period, 2 successive pairs of vector elements are processed.
The vector has dual vector pipes & dual set of vector functional units → Thus 2 pairs of elements
iii) In parallel vector processing, more than 2 results generated per clk cycle.
It automatically starts when :-
→ Successive vector instructions use diffent functional unit & multiple vector registers
→ Successive " " " the resulting flow from one vector reg as operand of other utilizing diff functional unit. This is called chaining.
iv) VP implents better with higher vector due to fundal^{n} delay in pipeline
v) VP decreases overhead to maintain loop control variables

★) Vector processor Classification

Memory to Memory Architecture

Register to Register Architecture

Three catagories to exploit parallism

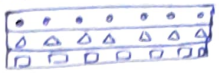
→ Instruct level parallelism

→ Stages with diff tim

→ Thread level parallelism

→ Multiple instruct parallely

→ Vector data parallism



i) Memory-to-Memory Architecture

→ Source operands, intermediate & final results are read directly from memory

→ Base address, offset, increment, vector length must be specified

→ Eg: TI-ASC, CDC STAR-100, Cyber-205

→ No limitation of Size

→ Speed is comparatively slow

ii) Register-to-register Architecture

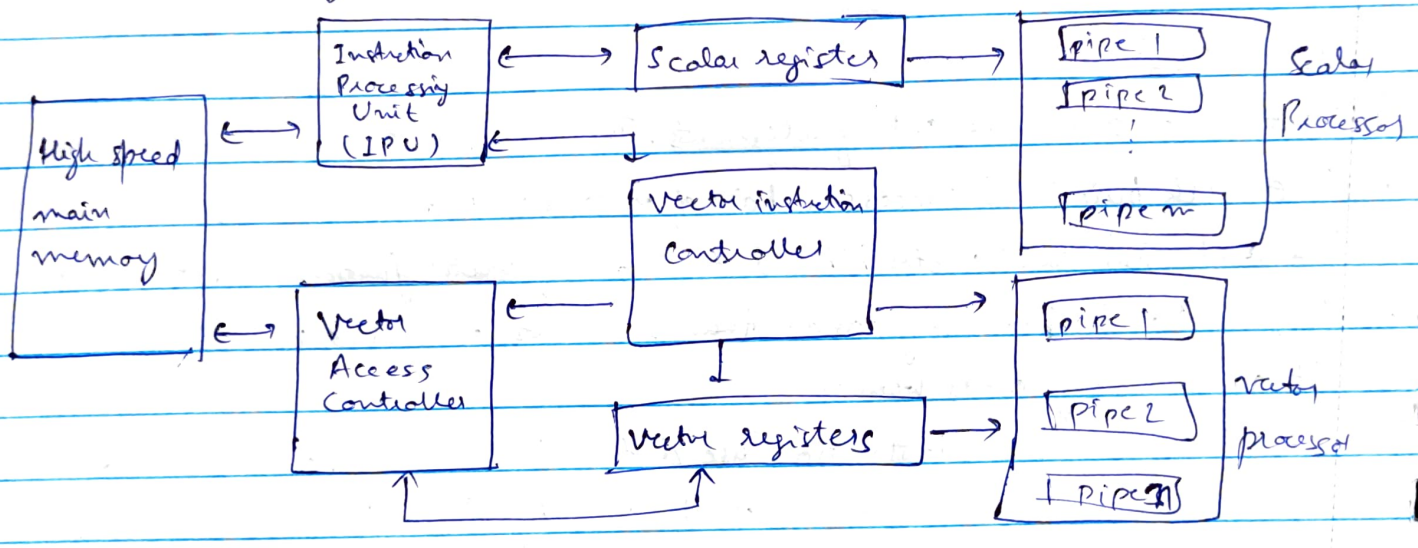
→ Operands and results read indirectly from memory via large no. of scalar or vector registers

→ Eg: Cray-1, Fujitsu VP-200

→ Limited Size

→ Speed is very high compared to memory to memory

→ H/W cost is high



) Shared Memory Symmetric multiprocessing:

Very common parallel computer architecture (Uses 4 to 32 processors)

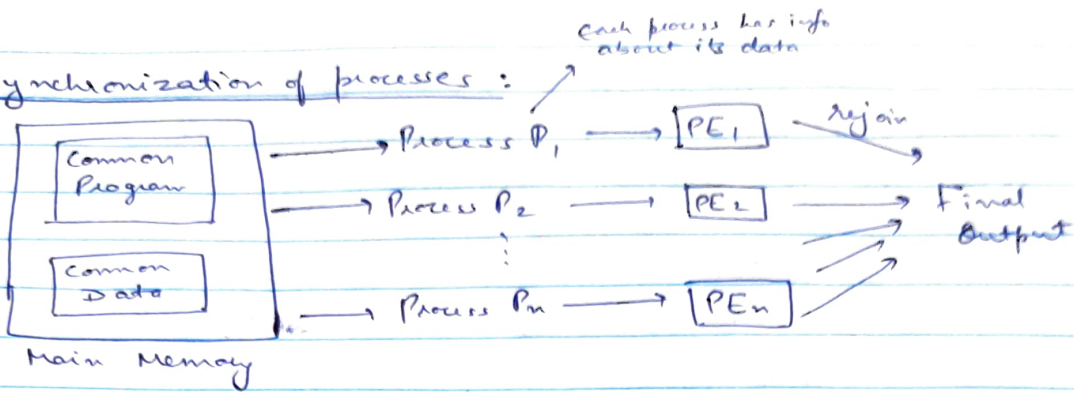
Provides global address space for writing || programs → easy to program

Each processor has a pvt cache

Each processor is connected to main memory (shared) either with shared bus or using Interconnection Network

The avg time to access main memory from any processor is same → Symmetric Multiprocessing (SMP)

*) Synchronization of processes :



i) fork : Statement used to create a process

ii) join : " " when parent process needs the result of child process

Eg:

Process A	Process B
:	:
fork B	Sum ← Sum + f(B)
:	!
Sum ← Sum + f(A)	end B
:	
join B	
!	
end A	

When A forks B, then till join, both A and B are working concurrently. Thus data maybe misread or miswritten.

Suppose Sum = 0 → A read
↳ B also read simultaneously

A did Sum ← Sum + f(A)
B did Sum ← Sum + f(B)

Now A writes Sum as f(A) & then B writes Sum as f(B). Final Sum → f(B)

This is because B should have read sum after A has done its work.

↓

Need for Lock

Locking

Process A	Process B
:	:
fork B	lock sum
:	Sum ← Sum + f(B)
lock sum;	unlock sum
Sum ← Sum + f(A)	:
unlock sum;	end B
:	
join B	
:	
end A	

We have the following hardware for lock :-

```
lock:   LD    R1, 0, L    // C(R1) ← C(L)
        CMP   R1, #0      // Compare R1 with 0
        BNZ   lock        // If R1 ≠ 0 try again
        ST    L, #1       // L ← 1
        RET               // Return
```

goes on waiting if L=1 ← busy-wait

```
unlock: ST    L, #0       // Store 0 in L
```

↑ immediate operand

L=0 → unlocked
L=1 → locked

However the problem is that lock is not an atomic instruction. Between the instructions of lock, if other process is reading lock then there may be dirty read. So for atomic instruction:-

Test & Set → atomic read-modify-write

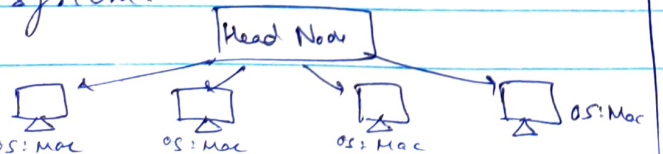
```
lock:  TST  R1, 0, L  // C(R1)←L, L←1      unlock: ST L,#0 //L←0
       CMP  R1, #0    // Compare R1 with 0          RET
       BNZ  lock      // If R1≠0 try again
```

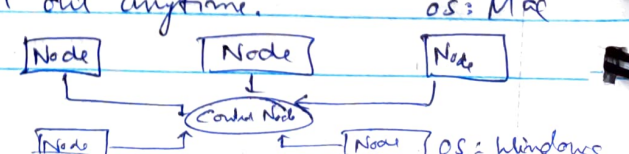
Barrier → ensures all processes complete before proceeding further
↳ also requires atomic instruct^n

Sequential Consistency: A multiprocessor is sequentially consistent if the result of any execution is the same as if the operation of all processor were executed in some sequential order and the operations of each individual processor occurs in this sequence in the order specified by its program.

*) Difference between Grid & Cluster Computing.

~~Grid~~ Cluster Computing (Homogenous)	Grid Computing
i) Collection of tightly or loosely coupled computers that work together to act as a single entity.	i) A network of ~~const~~ homogeneous or heterogeneous computers working together to perform a task difficult for a single machine.
ii) Computers are dedicated to same work & perform no other task	ii) Computers contribute their unused processing resources to grid network.
iii) Computers are located close to each other.	iii) Computers may be located at hyr distance from each other.
iv) Computers are connected by high speed LAN.	iv) Computers are connected by slow speed bus or internet.
v) Centralized Network Topology - central server	v) Decentralized Network Topology - less server control
vi) Whole system functions as a single system.	vi) Each node is autonomous & can opt out anytime.





*) Interconnection Networks: Each system needs to interact to other system via connection links.

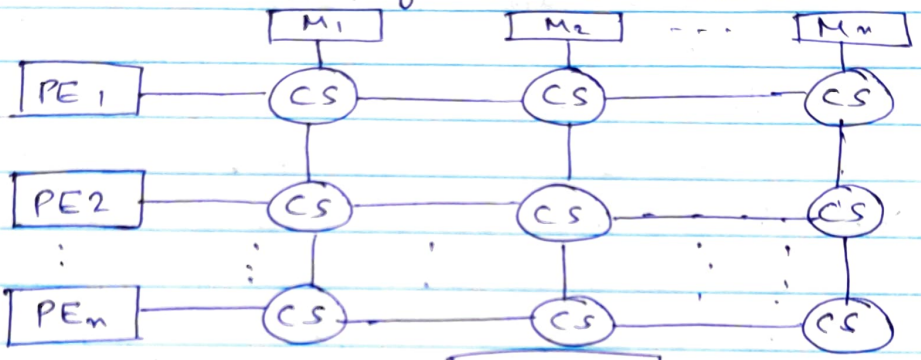
Direct
Each node connected to a neighbour node by dedicated communicatⁿ line

Indirect
Each node is connected to many other nodes using one or more switching elements

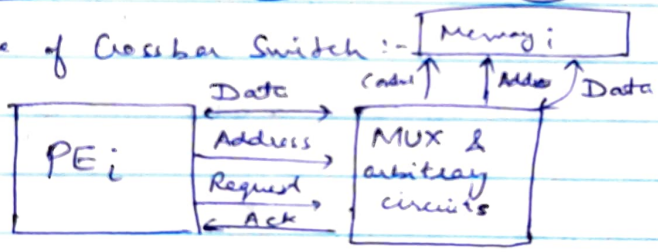
Processor → Memory

→ Crossbar Switch: Simultaneously connects (P_i, M_i)
Non-blocking N/W
Routing mechanism is called Crosspoint Switch
For n Processors & n Memory modules, n^2 switches are reqd

At any time
$P_i \leftrightarrow M_j$
$P_q \leftrightarrow M_q$
$(i \neq q, j \neq j)$

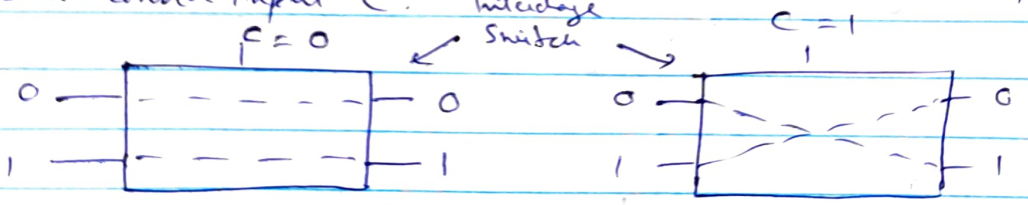


Structure of Crossbar Switch:-



→ Multistage Interconnectⁿ N/W & Generalized Cube N/W: It's basic component has a control input C.

Blocking N/W
$P_i \leftrightarrow M_j$
$P_q \leftrightarrow M_q$ ✗
Not possible



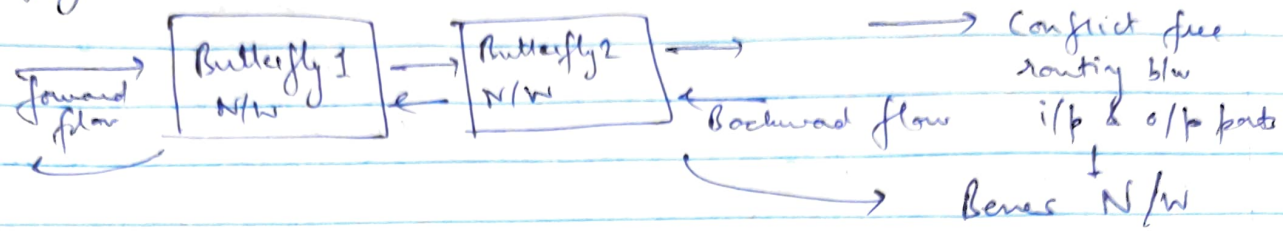
Straight connection — Cross connection

A general multistage N/W has N i/p, N o/p, $N = 2^m$ → no. of stages
Each stage → N/2 Interchange switches. So No. of switch = $m(N/2) = \frac{N}{2}\log_2 n$
(See figure in Book-Rajaraman Pg. 154)

→ Shuffle Interconnectⁿ: Shuffle$(a_m a_{m-1} \ldots a_2 a_1) = a_{m-1} a_{m-2} \ldots a_2 a_1 a_m$
↳ Circular left shift by 1 bit

→ Omega N/W : $\log_2 N$ cascaded switched using shuffle connect^n
See fig. Pg : 156

→ Butterfly N/W : Does not allow N i/p to N o/p connection without conflicts



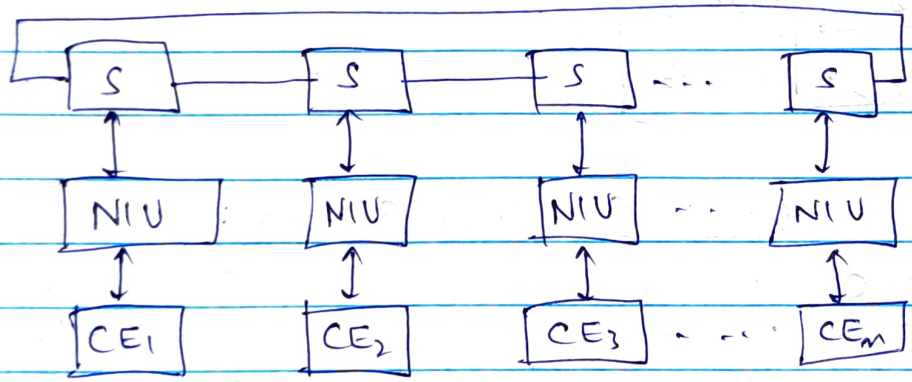
*) Direct (interconnect^n N/W : Characteristics of Interconnect^n N/W (CE = PE + Memory)

i) Total N/W Bandwidth : Bytes/second that N/W can support
For a ring of n CEs with individual link
BW = B, total BW = nB, for Bus = B

ii) Bisection Bandwidth : Imagine a cut dividing N/W into 2 parts & calculating B/W across the cut. In case of ring of n CEs, it is 2B. For Bus = B

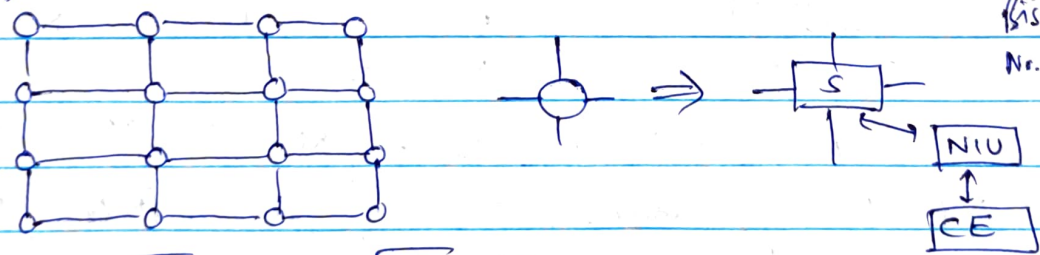
iii) No. of ports each N/W switch has. For ring, it is 3. For Bus = 1

iv) Total no. of links b/w switches & b/w switches & CEs : In ring it is 2n



NIU : N/W Interface Unit

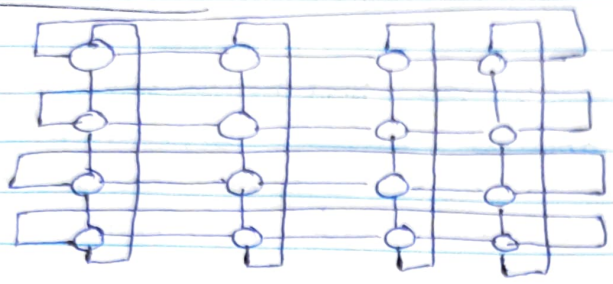
→ 2D grid :



Bisect^n B/W = $\sqrt{n}B$
No. of port per switch = 5 (4 to PEs, 1 to NIU)

n processors, $\sqrt{n}$ rows, $\sqrt{n}$ columns,
Each row → $(\sqrt{n}-1)$ links
Each col → $(\sqrt{n}-1)$ links
So, total no. of link = $2\sqrt{n}(\sqrt{n}-1)+n$
↓ Each switch to CE
If BW of each link = B
So total BW = $2\sqrt{n}(\sqrt{n}-1)B$

→ 2D Toroid: Switches at edges are also connected by links.



→ Max symmetric N/W

→ Disadvantage: Non-planar

If n CE, total B/W = 2nB

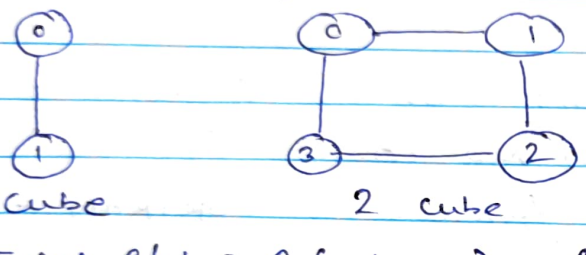
Bisect^n B/W = 2√n B

No. of links = 3n

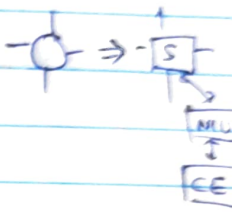
Ports per switch = 5 (4+1)

→ N cube or Hypercube: Interconnects 2^N CEs one at each corner of a cube.

N = 1 N = 2 N = 3



1 cube 2 cube 3 cube



Total B/W = $\frac{B(n\log_2 n)}{2}$ $\left(n = 2^N\right)$

Bisection B/W = $\frac{Bn}{2}$

No. of ports per switch = $1 + \log_2 n$

Total no. of links = $n + n\log_2\left(\frac{n}{2}\right)$

★) Routing Mechanisms for directly connected systems:

→ Each node connected to other node via switch → sophisticated to → Router

→ Router: Examine incoming messages & decided where it should be sent (for which CE it is destined)

→ Messages —broken to→ packets → | Header | Data |

→ Time taken by data at packet → determines time

Info of Destinat^n CE & order of packet (Seq. No.)

→ Packet switching

Source → Router → Destinat^n

packets re-ordered

→ Routing requires storage at each router to buffer packets → to order them

→ Virtual Cut through Routing: Router buffers a ~~packet~~ packet at a node only if the next link it is to take is occupied

→ Wormhole Routing: Packet sub-divided into flits (flow control digits)

→ First flit = Header info, rest → data

Header flit → block, other flits → buffered

→ All flits follow the header flit

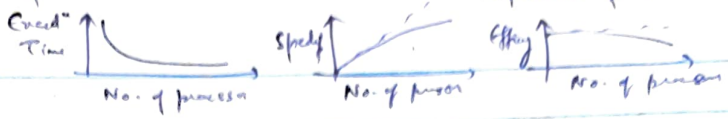
→ Very fast, space efficient

→ Flits of diff packet → Not interleaved

→ Successive flits of a packet → async pipelining w/ H/W

Speedup = $\frac{\text{Time taken by 1 processor}}{\text{Time taken by n processor}} = \frac{T(1)}{T(n)}$ — Superlinear speedup, when Speedup > n — no. of processors

Efficiency = $\frac{\text{speedup}}{n}$



★) Performance Benchmarks:

i) Peak-performance: Theoritical maximum based on best possible utiliz^n of resources

ii) Sustained performance: Based on running applicat^n-oriented benchmarks

Indicate instruct^n exec^n rate ↔ In terms of MIPS or Mflops

Indicate floating pt. capability ↔ Floating pt. operat^n

Benchmark: Set of programs or Program fragments used to compare performance of various machines.

↓ Used to decide which system to use in which applicat^n

i) Synthetic benchmarks: Small programs constructed for benchmarking but don't represent real computat^n (workload)

Eg: Whetstone formulated for FORTRAN to measure flop performance,

Dhrystone to measure integer performance in C

→ Don't reflect actual behaviour of complex computat^n

ii) Kernel benchmarks: Program fragments extracted from real programs. Heavily used core of the programs.

Eg: Livermore Loops for FORTRAN, Linpack for FORTRAN

Drawback: Performance results (in Mflops) are very large for non-scientific computations

iii) Application benchmarks: Several complete applications/programs that reflect the workload of a standard user, called (benchmark suits)

Implement^n is very time & resource intensive

Produces meaningful results

Eg: ~~SPEC~~ (System performance evalut^n corporat^n group)

SPEC2006, SPEC MPI2007, SPEC OMP2012 for parallel computing

For Sum & Ni :- → ↑ in time or ↑ memo

$$S(n) = \frac{T_s + G(n)T_p(n,W)}{T_s + \frac{G(n)}{n}T_p(n,W)} = \frac{\alpha + G(n)(1-\alpha)}{\alpha + \frac{G(n)}{n}(1-\alpha)}$$

i) $G(n) = 1$ → Amdahl's Law
ii) $G(n) = n$ → Gustafson's Law

★) Overheads in Parallel Processing: Parallel overhead is the time reqd to cordinate || tasks as opposed to doing useful work. Reasons for this are :-

i) IPC: If each processor (p processor) spends t_{comm} time for data transfer, then IPC time = $p \times t_{comm}$ (or $p \times t_{com} + \alpha$) ($0 < \alpha < 1$) → comput^n/commicat^n overlap factor
If ↑ high ⇒ ↑ parallel overhead

ii) Load Imbalance: Always not possible to determine size of sub-problem for each processor. Load may not be uniform. Some maybe idle, some overloaded → parallel overhead

iii) Inter-task Synchronization: If subtask B on processor P_B depends on subtask A runing on processor P_A, then execut^n of B is delayed till A is complete. For that time, B is idle → parallel overhead (wait a join or barrier)

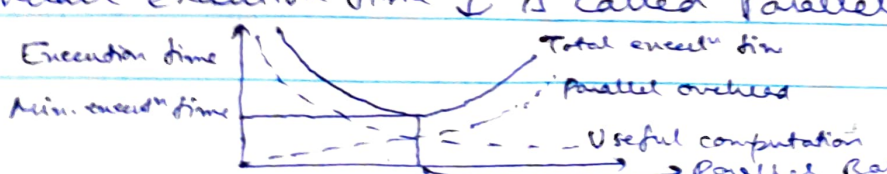
iv) Extra Computat^n: If the best sequential algorithm does not have higher degree of parallelism, then we maybe forced to use a parallel algo with more operations count. Eg: Bidirectional Gaussian elimination
The extra comput^n accounts to parallel overhead.

v) Other overheads: Task creation, Task scheduling, Task termination, memory latency, cache coherence enforcement etc.
Coarse grain task → Less || overhead but less degree of parallelism & Fine " " → Large " " " Large " " "

★) Parallel Balance Point: On ↑ no. of processors → execution time of prog ↓
If problem size is fixed, " " " " " → comput^n time per processor ↓ (Workload)

Execution time starts ↑ ← Workload ⇌ After some time → Parallel overhead

So an optimal no. of processor where on ↑ processors without ↑ problem size, the overall execution time ↓ is called Parallel Balance point.



Workload same, Time ↓ → Strong scaling → efficiency not fixed on ↑ processors

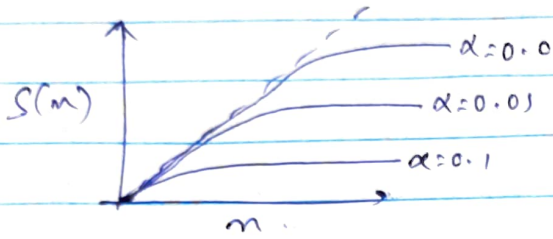
*) Speedup performance laws → sequential operations → bottleneck

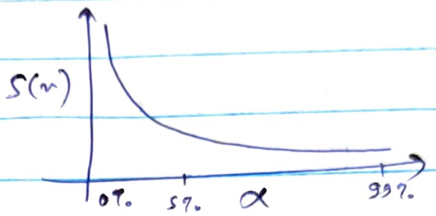
i) Amdahl's Law: Let a program has α fract^n of non-parallizable oper^n & $1-\alpha$ is parallelizable. If serial implementat^n takes T_s time. Problem size is fixed

So, $T_s = \alpha T_s + (1-\alpha) T_s$

and $T_p = \alpha T_s + (1-\alpha)\frac{T_s}{n}$ for n processors

So Speedup = $S(n) = \frac{T(1)}{T(n)} = \frac{T_s}{T_p} = \frac{T_s}{\alpha T_s + (1-\alpha)\frac{T_s}{n}} = \frac{1}{\alpha + \frac{(1-\alpha)}{n}}$





Workload is constant but execution time decreases.

ii) Gustafson's Law: When accuracy is more important than response time. As machine size ↑ to obtain more computing power, problem size can ↑ to create a greater work load to produce more accurate sol^n & execut^n time is unchanged. On ↑ problem size, sequential problem → no longer bottleneck. → Weak Scaling → efficiency fixed on ↑ processors

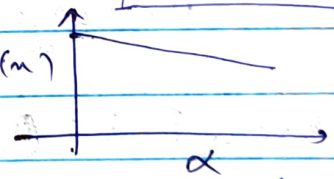
Here $S(n) = \frac{T_s + T_p(1,W)}{T_s + T_p(n,W)}$ (W = workload, n = processors)

Now $T_p(n,W) = n\,T_p(1,W)$ and $\alpha = \frac{T_s}{T_s + T_p(n,W)}$

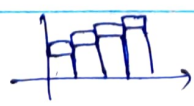
So, $S(n) = \alpha + n(1-\alpha)$ if n is large $\Rightarrow S(n) \approx n(1-\alpha)$

$= n - \alpha(n-1)$

Time same, Workload ↑



iii) Sun and Ni's Law: Generalizes Amdahl & Gustafson's law by (memory-bounded speedup) Solve maximum ~~size~~ possible size of problem limited by memory capacity. If we n processors of small memory → we get a total large memory. Now using this memory we can scale up problem size then if time limit satisfies Gustafson law then ↑ size → better sol^n, better efficiency. Execution time may increase



Time $T = g(M)$ → memory

~~$S(n) = \frac{T_s + T_p(n)}{T_s + T_p(n)(1-\alpha)}$~~