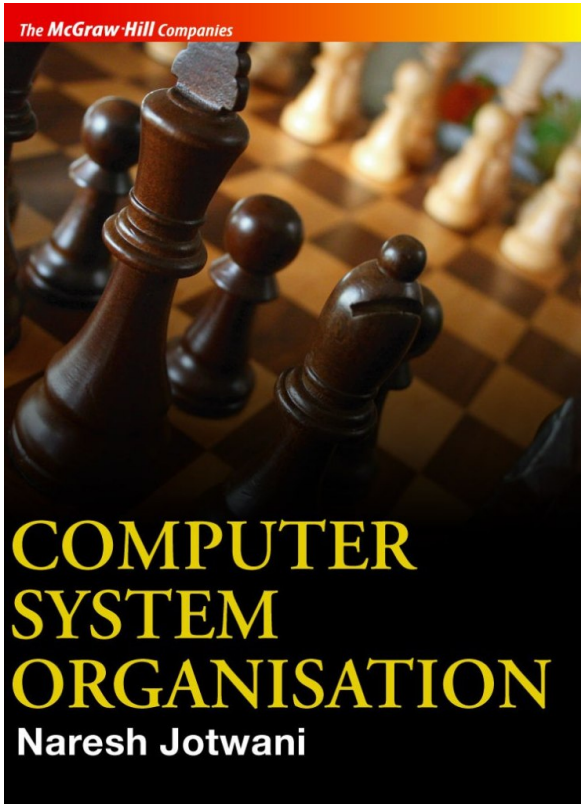
Mc Graw Hill Education

COMPUTER SYSTEM ORGANISATION

Naresh Jotwani

PowerPoint Slides



CHAPTER 13

PARALLELISM

Introduction

- In a computer system, *parallelism* in application or system programs is defined as any combination of:
  (i) two or more processors executing their machine instructions in parallel, and/or
  (ii) two or more machine instructions being executed in parallel on one processor.

- The multiple pipeline stages of a single processor, which operate in parallel, do satisfy the second part of the above definition.

- Multiprogramming on a single processor does not involve parallelism − unless of course the processor satisfies the second part of the definition.

- Multiprogramming is said to involve *pseudo-parallelism*, since the single processor on the system is shared between multiple running programs.

- As opposed to this, a system with multiple processors involves true parallel processing between them, and is referred to as *multiprocessing*.

- In a pipelined processor, only a single instruction stream passes through the multiple stages of the instruction pipeline.

  However, the *N* successive machine instructions occupying the *N* stages of the pipeline do get executed in parallel.

Instruction level parallelism

- A processor with an instruction pipeline presents opportunities for the exploitation of *instruction level parallelism* (ILP).

- Consider the following simple sequence of two instructions:

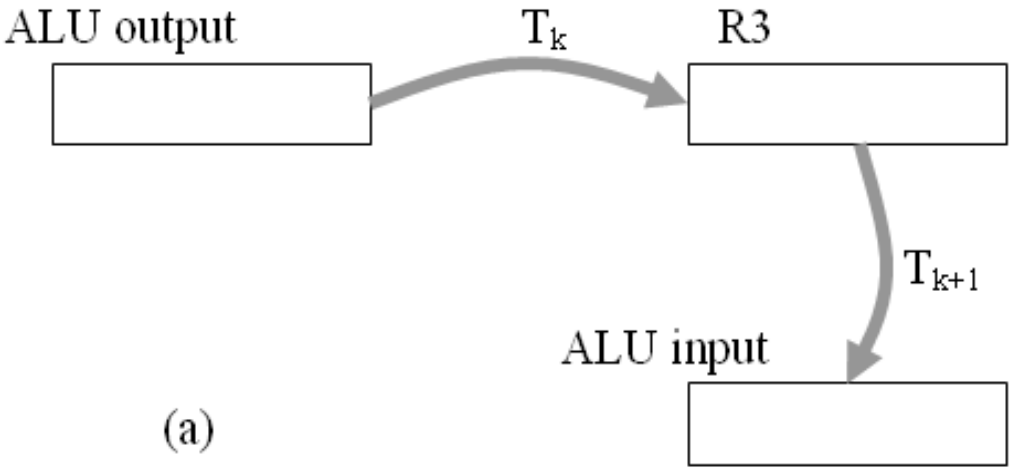
```
ADD       R1, R3
SHIFTR    #4, R3
```

- The result of ADD is stored in destination register R3, and then shifted right by four bits in the second instruction. Thus there is logical ***dependency*** between the two instructions – i.e. output of the first is input to the second.

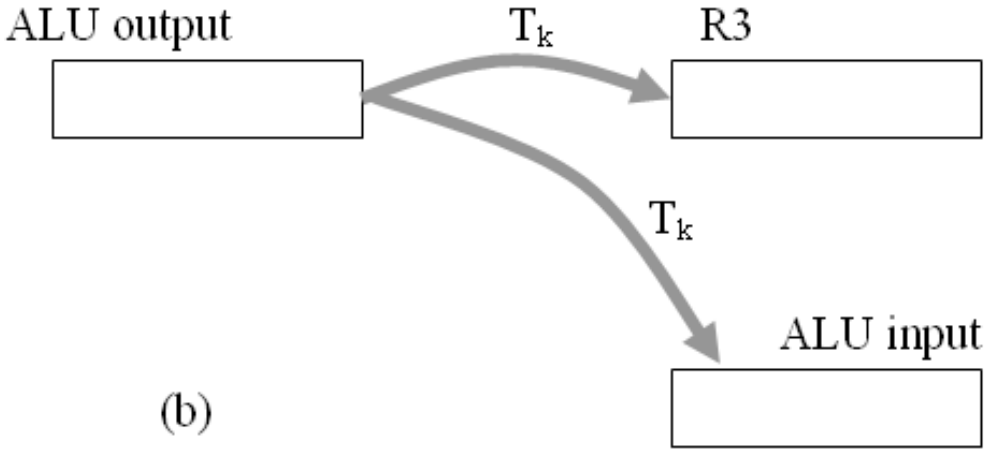
- In a pipelined processor, the second instruction should be executed just one stage – i.e. one clock cycle – behind the first.

- But it takes one clock cycle to transfer ALU output to R3, and then another clock cycle to transfer R3 content back to ALU input for the right shift – i.e. a total of two clock cycles.
- Therefore the second instruction above cannot be executed just one clock cycle behind the first.

- This is illustrated in part (a) of the next figure.
  - In clock cycle T_k, ALU output is transferred to R3 over an internal bus.
  - In the next clock cycle T_{k+1}, content of R3 is transferred to ALU input for the right shift.
  - The two data transfer operations take two clock cycles.



(a)



(b)

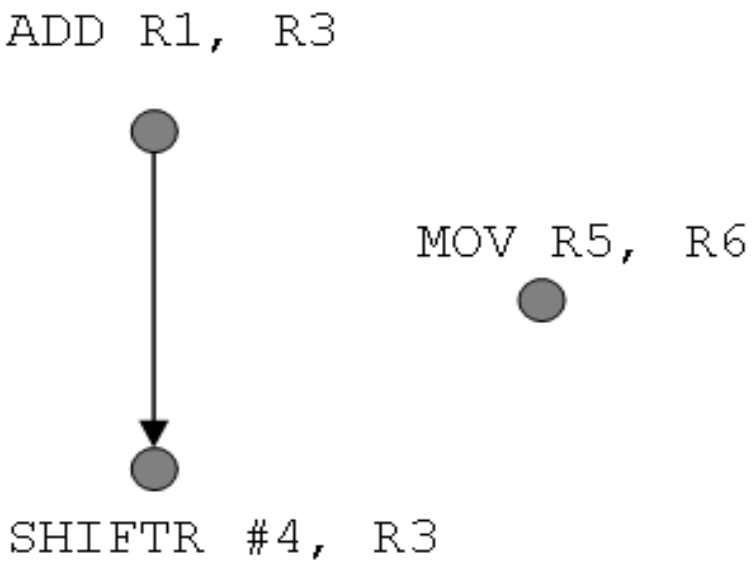
- Required data transfers can be achieved in only one cycle if ALU output is sent to both R3 and ALU input in the same clock cycle – see part (b) of the figure.
  - In general, if X is to be copied to Y, and in the next cycle Y is to be copied to Z, then we can just as well copy X to both Y and Z in one clock cycle.
- If this is done with the two instructions seen above, then the second is just one clock cycle behind the first, which is a basic requirement of an instruction pipeline.
- i.e. instead of performing two data transfers one after the other, we arrange to perform them in parallel.
  - This is parallelism at the level of elementary data transfer operations within the processor.
- The need for this type of parallelism shows up clearly in a pipelined processor.

- The above reasoning may apply even if there is an intervening instruction between **ADD** and **SHIFTR,** as in the following sequence:

```
ADD        R1, R3
MOV        R5, R6
SHIFTR     #4, R3
```

- **SHIFTR** must be executed after ADD, since it is the result of **ADD** which has to be shifted, i.e. there is a logical *dependency* between **ADD** and **SHIFTR**.

- But there is no such dependency between **MOV** and any of the other two instructions – which means **MOV** can be executed before **ADD**, or after **SHIFTR**.



- Figure shows these dependencies between instructions in the form of a graph.
  - *Node* in the graph represents an instruction
  - *Edge* between nodes represents logical dependency.
  - There is only one dependency in this instance amongst the three instructions
  - Therefore the graph has three nodes and one edge.

- MOV can be executed in a different order than in which machine instructions appear in the program, for better utilization of processor cycles.
  - Example: if for some reason ADD cannot be executed in a given clock cycle, MOV can be executed before it.
- i.e. the processor can perform *out of order execution* of instructions.
- Decisions such as

  (i) transfer of ALU output in parallel to both R3 and ALU input, and/ or

  (ii) out of order execution of MOV

  must be made by the hardware 'on-the-fly'. Control unit must detect any such possibilities and generate the required control signals.

- Data transfers make up a large part of the work of any processor.

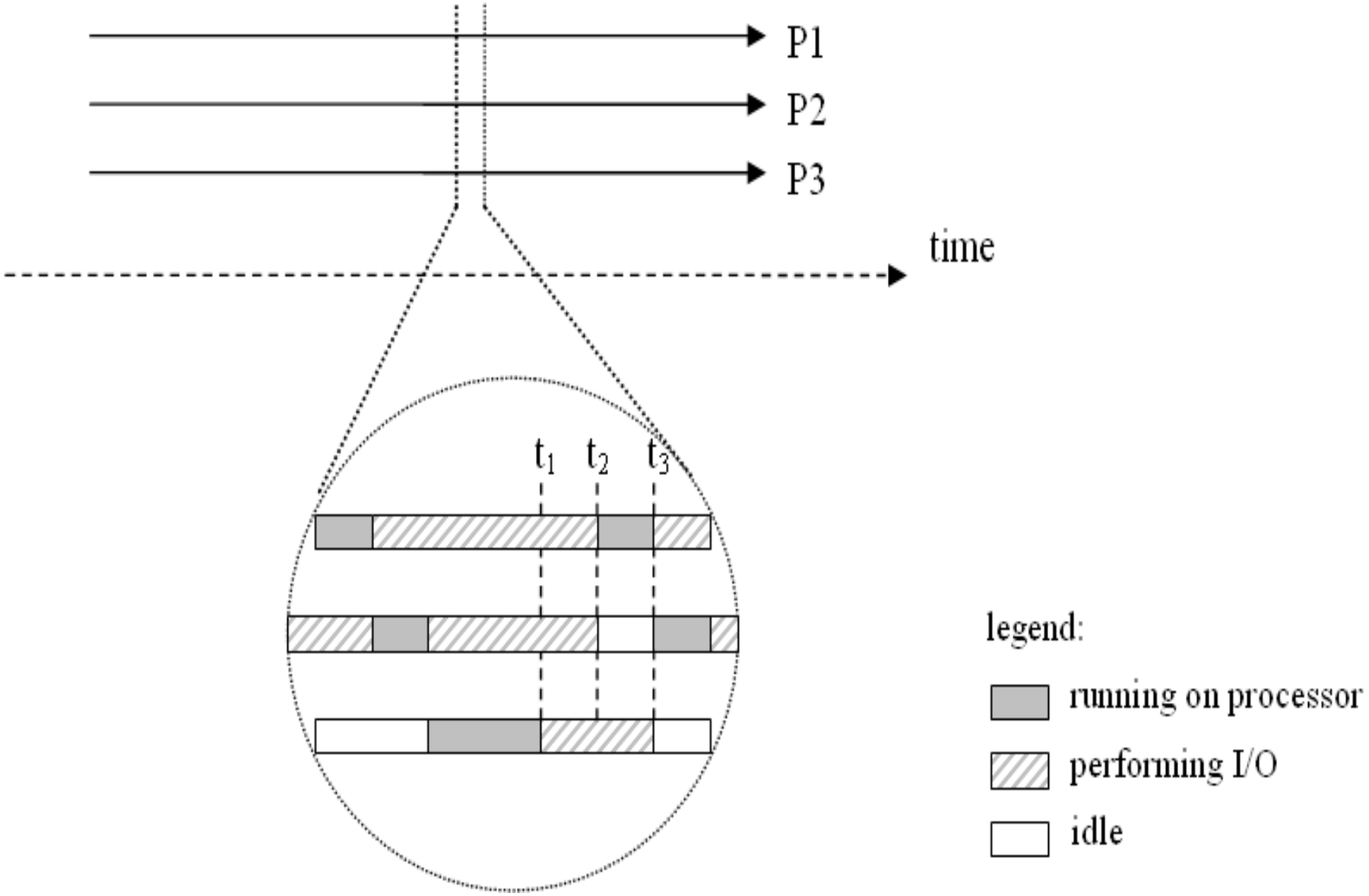
- To fully utilize the pipeline, designers must pay very close attention to the data transfers required in a single instruction and also across multiple instructions.

- High performance processors are often provided with multiple functional units.
  - With two FPUs, for example, two floating point operations can be carried out in parallel.
  - If each of the FPUs is itself designed with an internal pipeline, then each FPU can itself perform multiple floating point operations.

- Assume that a processor has multiple functional units.
- Also assume that, possibly through the use of an internal pipeline, each functional unit is capable of completing one instruction in every processor clock cycle.
- Then, at least in theory, the total throughput of the processor can exceed one instruction per clock cycle. Such a processor is said to have a *super-scalar* architecture.
- In practice, instruction dependencies and pipeline flushes and stalls result in reduced throughput.

Pseudo-parallelism

- Next figure shows a detailed view of the running of three programs – labeled P1, P2 and P3 – in multiprogramming mode.
  - In the top part, all three programs seem to be running in parallel.
  - i.e. each program makes progress in terms of instructions executed and input/output performed.
  - But, since the system has only one processor, at any one time only one of the three programs can execute instructions on the processor.
  - Thus, in terms of the actual execution of program instructions, there is no parallelism in this system.


P1
P2
P3
time
t1
t2
t3
legend:
running on processor
performing I/O
idle

- Bottom part has magnified view of a small time period.
- Grey-coloured slice of time indicates that a particular program's instructions are executed on the single processor. Note that, at any time instant, at most one of the three programs is in this *execute phase*.
- Slanted grey lines indicate that a program is performing an I/O operation. E.g. At the time instant t_1, both P1 and P2 are performing I/O.
- I/O operations can be done in parallel either because (a) the system has multiple I/O devices capable of operating in parallel, or (b) an I/O device allows multiple operations to proceed in parallel.
- In the time interval from t_1 to t_2, all three programs are performing I/O operations.

- In time interval t_2 to t_3, program P1 is running on the processor, P3 is performing I/O, and P2 is doing nothing.
  - This is because P2 needs to run on the processor, but the single processor is assigned to P1 during that interval.
  - At time t_3, program P1 'releases' the processor, and P2 resumes its execution.
- In terms of operation of the single processor, there is in fact NO true parallel processing between P1, P2 and P3.
- But, in a 'longer term' view, all three programs make progress in parallel, with better overall system utilization.
- Therefore this form of apparent parallelism between running programs is known as *pseudo-parallelism*.

- When programs running in parallel do not share any resources or data, then the execution of one program can have no bearing on any other program.
- However this is only a theoretical possibility, because the running programs do share the hardware and software resources of the system they are running on.
- Therefore complete independence between running programs is not always possible.
  - e.g. Two programs need to send their outputs to a common printer. If the system does not handle this requirement correctly, printed pages of two users may get garbled.
- Consider two programs, say P1 and P2, which are both working with a shared common item of data. Say that the data item shared by P1 and P2 is a bank balance.

- At a given time, P1 must add Rupees 5,000/- to the bank balance, and P2 must subtract Rupees 2,000/- from it.
- The combined effect of these two 'transactions' will be that the final bank balance increases by Rs. 3,000/-. If we assume that the initial bank balance is Rs. 20,000/-, then the correct final value is Rs. 23,000/-.
- Relevant parts of program P1 and P2, in algorithmic notation:

```
Program P1
.
.
read bank balance from memory
add 5000 to it
store bank balance in memory
.
.
```

```
Program P2
.
.
read bank balance from memory
subtract 2000 from it
store bank balance in memory
.
.
```

- It is possible that, after P1 completes the first step, the single processor shared by them begins to execute instructions of P2.
  - If this happens, P2 will also execute its first step before P1 moves on to execute the remaining two steps.

- The effect of this particular sequence of operations is that both P1 and P2 read the initial value of the shared data item, which we have assumed is 20,000.
  - At this point, both P1 and P2 have a separate copy of this one data item – each within its own 'program space'.
  - Both P1 and P2 then work on their own copy of the data, before writing it back in its shared location.

- The nett effect is that P1 changes its copy of the data item to 25,000, while P2 changes its copy to 17,000. Both then proceed to write their computed value back in the shared location.

- The result is that the final value of the bank balance is either 25,000 or 17,000, depending on which program was the last to write its value to memory.
- But both these answers are wrong. As we know, the correct answer is 23,000.
  - We must identify what goes wrong if steps of P1 and P2 happen to be executed in this particular sequence.
- Erroneous answer is produced because both P1 and P2 are changing the value in parallel – rather than one after the other.
  - The correct answer is achieved if P1 completes its three steps before P2 − or even the other way around.
  - i.e. An error can occur only if steps of P1 and P2 become 'interleaved' in terms of their execution times.
- To avoid erroneous answers, the three steps shown of P1 and P2 must not become interleaved in time.

- If either P1 or P2 is changing the shared value, the other program must be blocked from doing the same.

  Only after one program completes making the change is the other program allowed to do so.

- This synchronization between running programs can be achieved by ensuring the following condition:
  - While one program is executing a *critical section* of code, it must complete executing this critical section before another program is allowed to execute its own critical section.

- Need for such synchronization arises during parallel execution of two or more running programs.

- A 'running program' is referred to as a *process* – e.g. P1 and P2 above. This problem is therefore an instance of *inter-process synchronization*.

Thread Level Parallelism (TLP)

- For increased throughput, a program can be designed in the form of multiple *threads of execution* (i.e. threads) – i.e. as a a *multi-threaded program*.
- As a process runs, the successive values of its program counter (PC) define a *thread of execution*.
  - In a multi-threaded process, processor time is shared amongst the multiple threads.
- For this, the higher level programming language and operating system must provide necessary support.
- The computational work being done by a single process is arranged in the form of multiple *threads*.
- On a single processor system, multi-threading is a form of *pseudo-parallelism*.

- Threads run independently except when explicit synchro-nization is needed between two or more threads.

- Two approaches to support TLP in a processor:

(a)Coarse-grained multi-threading (CGMT)

Processor switches to another thread when the execution of one thread is held up for access to main memory.

- Access to main memory (on a cache miss) may take a few tens of processor clock cycles, during which time another thread runs on the processor.
- At a given time, processor pipeline is occupied by instructions of the one thread that is being executed.

(b) Fine-grained multi-threading (FGMT)

Based on sharing of processor cycles in a regular manner amongst multiple executing threads.

(continued)

  - Stages of processor pipeline are occupied by instructions from these threads;
  - Sharing of processor time is on the basis of individual clock cycles.

- Under CGMT and FGMT, data paths and functional units of the processor are shared amongst the threads.
  - Registers are provided in multiple sets for the multiple threads.

- In general, a high performance processor will have features to exploit both ILP and TLP.

- For either CGMT or FGMT, processor must provide a mechanism to switch quickly between the register sets of two threads.

Flynn's classification

- *Instruction stream* - Sequence of machine instructions getting executed on a processor; may correspond to a process, or to a single thread in a multi-threaded process.
- *Data stream* - Sequence of data items as they are brought into a processor from main memory, operated upon, and then stored back in main memory.
- On a uni-processor system, a single instruction stream operates over time upon a single data stream.
- In 1972, M J Flynn pointed out that parallelism in a computer system involves: (a) multiple instruction streams, or (b) multiple data streams, or (c) both.

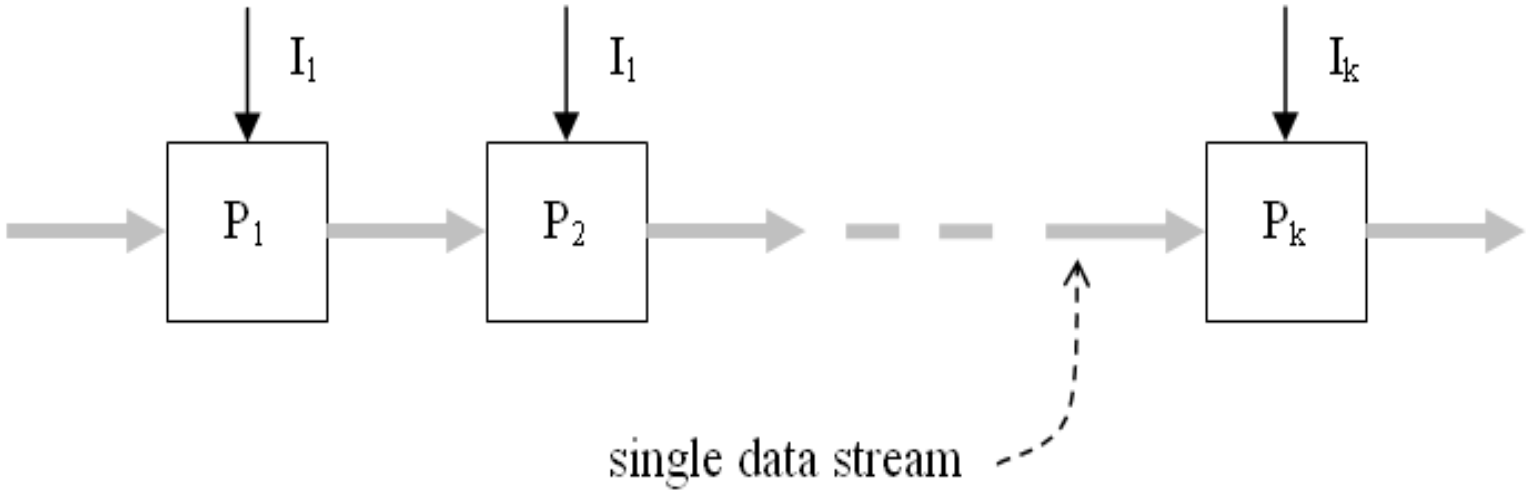
		Data Stream	
		Single	*Multiple*
Instruction Stream	*Single*	SISD	SIMD
	Multiple	MISD	MIMD

- The resulting classification of computer systems takes the form of the table above.

- The four possible types of systems resulting from this scheme are SISD, MISD, SIMD and MIMD.
- A single processor system, classified as SISD, has already been studied. The other three possibilities will now be examined.

(i) Multiple instruction stream, single data stream (MISD) system

- Such a computer system is difficult to visualize, because two or more instructions cannot operate simultaneously upon a single data item common to them.
- But if multiple instructions operate in sequence – i.e. one after the other – upon a single data item, then there no essential parallelism in the system.
- The next figure shows a single data stream – horizontal arrow going from left to right – being operated upon by *k* instruction streams on *k* different processors.



- But on any one item of data, *k* instruction streams are operating in sequence, i.e. the arrangement is *k* sequential stages in processing data.

- There is no essential parallelism in the processing architecture shown – in the sense that even a single processor can perform the *k* stages of processing on the data, one after the other.

(ii) *Single instruction stream, multiple data streams (SIMD)*

- Originally, this involved multiple processors operating in parallel in synchronized mode, each processor working on its own ‘slice’ of data.
- Newer variants have emerged since then.

(iii) *Multiple instruction stream, multiple data streams (MIMD)*

- True parallelism between multiple processors. Each *instruction stream* in the system operates upon its own *data stream*.
- Have been eveloped in several different basic forms: e.g. ***shared memory multiprocessors***, ***cluster computers***, and ***distributed computing systems***.

Key system design issues for MIMD systems are the following:

(a) whether multiple processors in the system share common main memory,

(b) whether the multiple processors are identical to each other, and

(c) the way in which the multiple processors communicate with one another.

These issues will be discussed in brief in the next chapter.

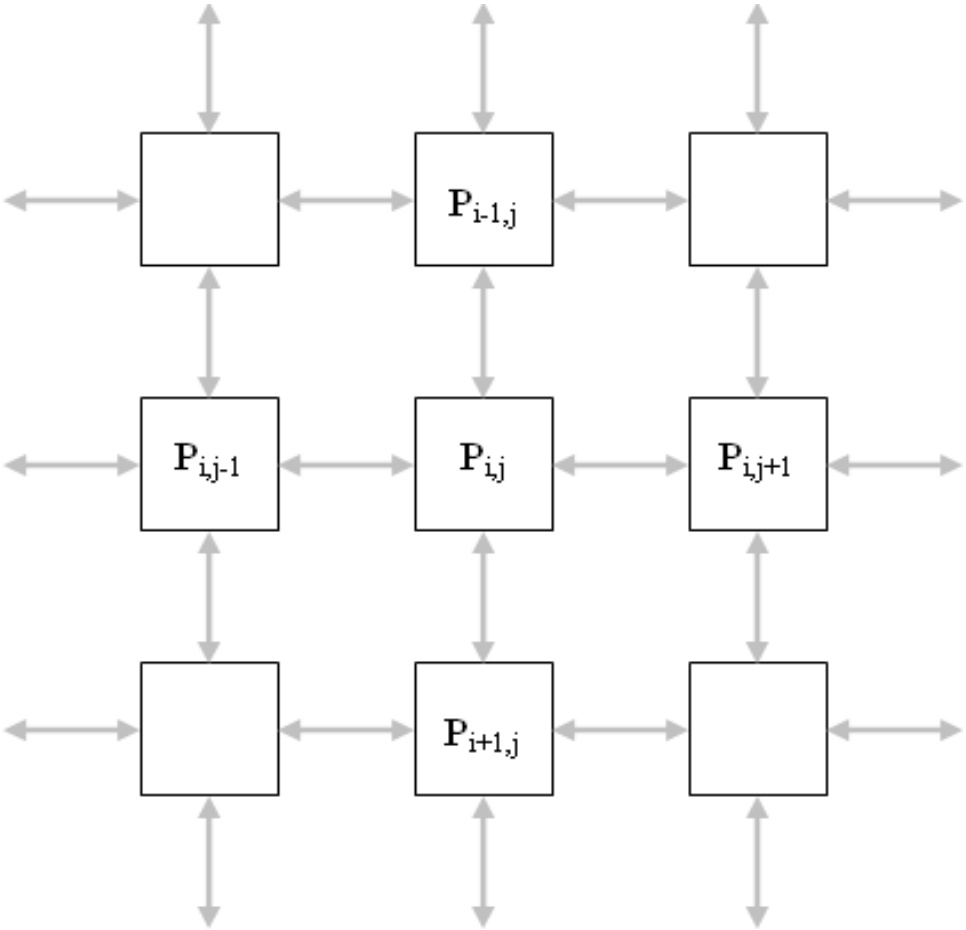
SIMD architecture

- SIMD originally had the form shown in the next figure. Each of the processing elements (PEs) in the system has its own processor and local memory.

- PEs are connected together with a regular *topology* –in the figure, this is a rectangular array. Hence such system is also known as *array processor.*

- All the processors execute the same program – i.e. a *single instruction stream* gets executed at each PE.

- Execution at PEs is fully synchronized – at any given time, processors in all the PEs execute the same instruction. Instructions may be broadcast to PEs, or pre-stored in local memories of PEs.



- Each PE processes – i.e. operates upon – a different slice of the aggregate data.
- This requires that the aggregate data on which the system operates must have a structure which matches the topology of the system.

- Heavy processing needs of certain types of applications can be met with an array of processors.

- Since each PE is processing a different slice of the aggregate data. Thus the aggregate data is divided into *multiple data streams*, one stream per PE.
  - Explains why such a system is known as *single instruction stream, multiple data stream* (SIMD) system.

- In a given instruction, each PE in the array will either
  - (a) operate on some data in a local register or memory, or
  - (b) send and receive data from its adjacent PEs along a row or along a column.

- If PEs are connected as a two-dimensional array, the four adjacent PEs of the PE at array position (i,j) are defined as follows:
  - (i) along row i, its two adjacent PEs are at array positions (i-1,j) and (i+1,j), and
  - (ii) along column j, its two adjacent PEs are at array positions (i,j-1) and (i,j+1).

- SIMD systems are not meant for general purpose computing applications.

- Only special problems which can be 'mapped' to the system topology are suitable for such systems.
  - In each case the aggregate data and the required software must be carefully mapped to the system. Because of difficulty in achieving this, the original model of SIMD has not achieved acceptance.

- On a parallel processing system, different applications have different requirements of inter-processor communication.
  - A major weakness of the architecture of the previous figure is that connections between processing nodes cannot be re-configured for a different application.
- The SIMD system described here represents the original *array processor* which emerged in the 1970s.
  - In this version, for the reasons mentioned, such a system has not found acceptance.
- Since then, processor and interconnect technologies have advanced to a great extent, and the basic concept of SIMD has evolved much further, as we shall see.

Vector Processing

- Each instruction of the processor we have seen operates on one operand, or a pair of operands.
- We often deal with *vectors* of numbers – usually floating point numbers – such that an operation must be performed on each number in the vector.
- If X and Y are vectors of 100 numbers each, we may denote this by writing:

$\mathbf{X = [\ x_1, x_2, \ldots\ldots x_{100}\ ]}$ **and**

$\mathbf{Y = [\ y_1, y_2, \ldots\ldots y_{100}\ ]}$

- Suppose vector X is to be modified by adding respective elements of vector Y to those of X.

- On a conventional processor, this requires the addition to be performed in an iterative loop which is executed 100 times, as shown below:

```
for values of i from 1 to 100
{
replace x_i by x_i + y_i
}
```

- Suppose we have a processor on which a single instruction causes these hundred additions to be performed. What does this instruction require?

- Assume that elements of both these vectors are located in consecutive memory locations.

- A *vector addition* instruction then requires the following three operand values to be specified:
  - the starting address in memory of vector X,
  - the starting address in memory of vector Y, and
  - the common length of X and Y.

- Therefore vector addition instruction – say VADD – may have a format such as:

```
VADD      X1, Y1, 100
```

- Note that VADD requires only a single *instruction fetch*. Once this instruction is fetched and decoded, all hundred additions take place within its *execute phase*.

- Clearly this results in faster processing of vectors.

- Another important factor leading to higher performance through vector processing is this:
- Vector operations can make full use of pipelined architecture, in the data paths and in the FPU.
- Reason: While the elements of a vector are being operated upon one after the other, no intervening conditional jumps can occur; therefore no costly pipeline flushes take place.
- Vector processing also has the advantage that it does not require a fundamentally different processor architecture.
- This means that vector instructions can be provided as enhancements on the basic architecture of a conventional pipelined processor.

- *Supercomputers* address the high processing needs of many applications in science and engineering.
  - Such applications involve intensive use of floating point operations on vectors and arrays.

- One approach has been to rely heavily on *vector processing*, *pipelining*, and *multiple FPUs*.
  - With VLSI technology, architectural features originally developed for supercomputers are now provided on processors used in workstations and PCs.
  - Special processors have also been developed for graphics and animated graphics.

- Vector processing does not resemble the original version of SIMD processing. But it can be classified as SIMD, since a single vector instruction processes the multiple data items of a vector.

SUMMARY

- Instruction level parallelism
- Pseudo-parallelism
- Thread level parallelism
- Flynn's classification of computer systems
- SIMD architecture
- Vector processing