ARCHITECTURES FOR PARALLEL COMPUTATION

1. Why Parallel Computation
2. Parallel Programs
3. A Classification of Computer Architectures
4. Performance of Parallel Architectures
5. The Interconnection Network
6. SIMD Computers: Array Processors
7. MIMD Computers
9. Multicore Architectures
10. Multithreading
11. General Purpose Graphic Processing Units
12. Vector Processors
13. Multimedia Extensions to Microprocessors

The Need for High Performance

Two main factors contribute to high performance of modern processors:

- Fast circuit technology
- Architectural features:
  - large caches
  - multiple fast buses
  - pipelining
  - superscalar architectures (multiple functional units)

The Need for High Performance

Two main factors contribute to high performance of modern processors:

- Fast circuit technology
- Architectural features:
  - large caches
  - multiple fast buses
  - pipelining
  - superscalar architectures (multiple functional units)

However

- Computers running with a single CPU, often are not able to meet performance needs in certain areas:
  - Fluid flow analysis and aerodynamics
  - Simulation of large complex systems, for example in physics, economy, biology, technic
  - Computer aided design
  - Multimedia
  - Machine learning

A Solution: Parallel Computers

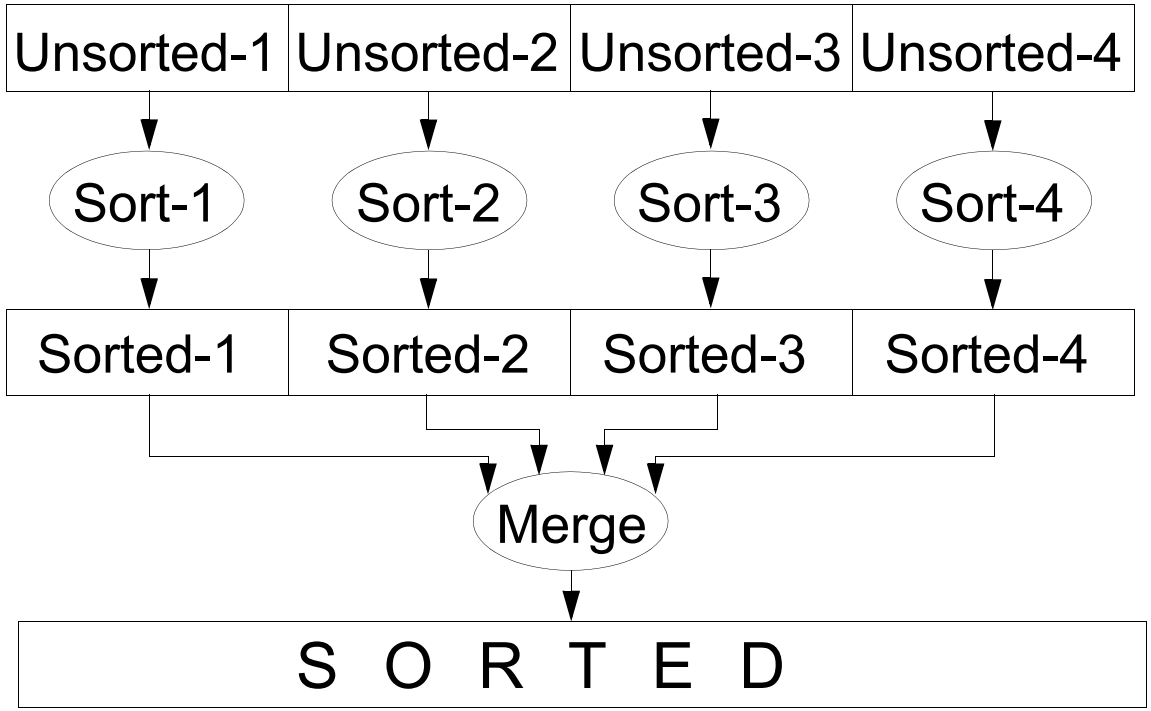
- One solution to the need for high performance: architectures in which *several CPUs* are running in order to solve *a certain application*.

- Such computers have been organized in different ways. Some key features:
  - number and complexity of individual CPUs
  - availability of common (shared memory)
  - interconnection topology
  - performance of interconnection network
  - I/O devices
  - - - - - - - - - - - - -

- To efficiently use parallel computers you need to write parallel programs.

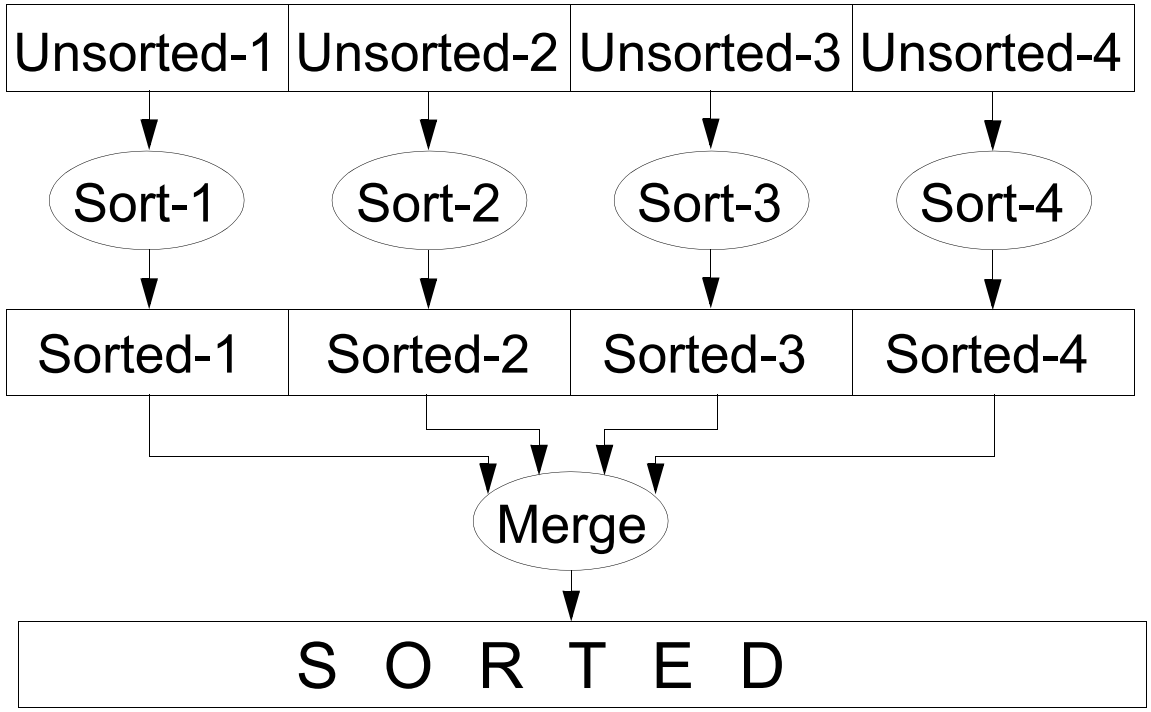
Parallel Programs

Parallel sorting



Parallel Programs

Parallel sorting




```
var t: array [1..1000] of integer;
- - - - - - - - - - -
procedure sort (i, j:integer);
    - - sort elements between t[i] and t[j] - -
end sort;

procedure merge;
    - - merge the four sub-arrays - -
end merge;
- - - - - - - - - - -

begin
    - - - - - - - -
    cobegin
        sort (1,250) |
        sort (251,500) |
        sort (501,750) |
        sort (751,1000)
    coend;
    merge;
    - - - - - - - -
end;
```

Parallel Programs

Matrix addition:

$$\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ a_{31} & a_{32} & \cdots & a_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{vmatrix} + \begin{vmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ b_{31} & b_{32} & \cdots & b_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{vmatrix} = \begin{vmatrix} c_{11} & c_{12} & \cdots & c_{1m} \\ c_{21} & c_{22} & \cdots & c_{2m} \\ c_{31} & c_{32} & \cdots & c_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ c_{n1} & c_{n2} & \cdots & c_{nm} \end{vmatrix}$$

Parallel Programs

Matrix addition:

$$\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ a_{31} & a_{32} & \cdots & a_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{vmatrix} + \begin{vmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ b_{31} & b_{32} & \cdots & b_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{vmatrix} = \begin{vmatrix} c_{11} & c_{12} & \cdots & c_{1m} \\ c_{21} & c_{22} & \cdots & c_{2m} \\ c_{31} & c_{32} & \cdots & c_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ c_{n1} & c_{n2} & \cdots & c_{nm} \end{vmatrix}$$

Sequential version:

```
var  a: array [1..n, 1..m] of integer;
     b: array [1..n, 1..m] of integer;
     c: array [1..n, 1..m] of integer;
     i: integer
- - - - - - - - - - -
begin
    - - - - - - - -
    for i:=1 to n do
        for j:= 1 to m do
            c[i,j]:=a[i, j] + b[i, j];
        end for
    end for
    - - - - - - - -
end;
```

Parallel Programs

Matrix addition:

$$\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ a_{31} & a_{32} & \cdots & a_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{vmatrix} + \begin{vmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ b_{31} & b_{32} & \cdots & b_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{vmatrix} = \begin{vmatrix} c_{11} & c_{12} & \cdots & c_{1m} \\ c_{21} & c_{22} & \cdots & c_{2m} \\ c_{31} & c_{32} & \cdots & c_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ c_{n1} & c_{n2} & \cdots & c_{nm} \end{vmatrix}$$

Sequential version:

```
var   a: array [1..n, 1..m] of integer;
      b: array [1..n, 1..m] of integer;
      c: array [1..n, 1..m] of integer;
      i: integer
- - - - - - - - - - -
begin
    - - - - - - - -
    for i:=1 to n do
        for j:= 1 to m do
            c[i,j]:=a[i, j] + b[i, j];
        end for
    end for
    - - - - - - - -
end;
```

Parallel version:

```
var   a: array [1..n, 1..m] of integer;
      b: array [1..n, 1..m] of integer;
      c: array [1..n, 1..m] of integer;
      i: integer
- - - - - - - - - - -
procedure add_vector(n_ln: integer);
    var j: integer
begin
    for j:=1 to m do
        c[n_ln, j]:=a[n_ln, j] + b[n_ln, j];
    end for
end add_vector;
begin
    - - - - - - - -
    cobegin for i:=1 to n do
        add_vector(i);
    coend;
    - - - - - - - -
end;
```

Parallel Programs

Matrix addition:

$$\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ a_{31} & a_{32} & \cdots & a_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{vmatrix} + \begin{vmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ b_{31} & b_{32} & \cdots & b_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{vmatrix} = \begin{vmatrix} c_{11} & c_{12} & \cdots & c_{1m} \\ c_{21} & c_{22} & \cdots & c_{2m} \\ c_{31} & c_{32} & \cdots & c_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ c_{n1} & c_{n2} & \cdots & c_{nm} \end{vmatrix}$$

Vector computation version 1:

```
var  a: array [1..n, 1..m] of integer;
     b: array [1..n, 1..m] of integer;
     c: array [1..n, 1..m] of integer;
     i: integer
- - - - - - - - - - -
begin
    - - - - - - - -
    for i:=1 to n do
        c[i,1:m]:=a[i,1:m] +b [i,1:m];
    end for;
    - - - - - - - -
end;
```

Parallel version:

```
var  a: array [1..n, 1..m] of integer;
     b: array [1..n, 1..m] of integer;
     c: array [1..n, 1..m] of integer;
     i: integer
- - - - - - - - - - -
procedure add_vector(n_ln: integer);
    var j: integer
begin
    for j:=1 to m do
        c[n_ln, j]:=a[n_ln, j] + b[n_ln, j];
    end for
end add_vector;

begin
    - - - - - - - -
    cobegin for i:=1 to n do
        add_vector(i);
    coend;
    - - - - - - - -
end;
```

Parallel Programs

Matrix addition:

$$\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ a_{31} & a_{32} & \cdots & a_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{vmatrix} + \begin{vmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ b_{31} & b_{32} & \cdots & b_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{vmatrix} = \begin{vmatrix} c_{11} & c_{12} & \cdots & c_{1m} \\ c_{21} & c_{22} & \cdots & c_{2m} \\ c_{31} & c_{32} & \cdots & c_{3m} \\ \cdots & \cdots & \cdots & \cdots \\ c_{n1} & c_{n2} & \cdots & c_{nm} \end{vmatrix}$$

Vector computation version 1:

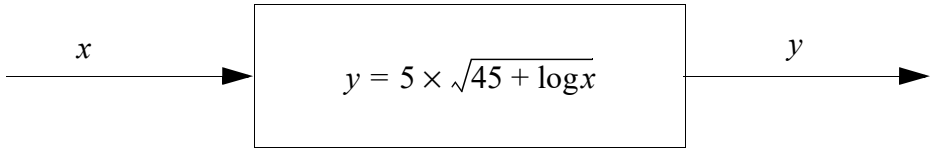
```
var   a: array [1..n, 1..m] of integer;
      b: array [1..n, 1..m] of integer;
      c: array [1..n, 1..m] of integer;
      i: integer
- - - - - - - - - - -
begin
    - - - - - - - -
    for i:=1 to n do
        c[i,1:m]:=a[i,1:m] +b [i,1:m];
    end for;
    - - - - - - - -
end;
```

Vector computation version 2:

```
var   a: array [1..n, 1..m] of integer;
      b: array [1..n, 1..m] of integer;
      c: array [1..n, 1..m] of integer;
- - - - - - - - - - -
begin
    - - - - - - - -
    c[1:n,1:m]:=a[1:n,1:m]+b[1:n,1:m];
    - - - - - - - -
end;
```

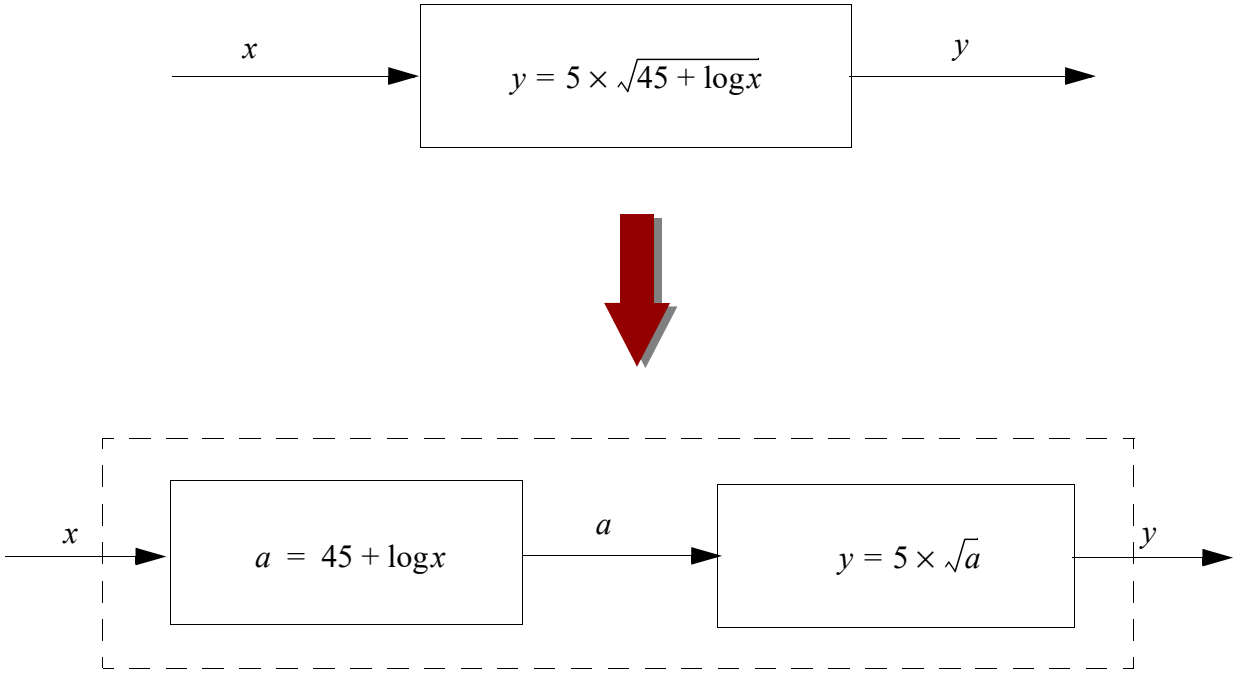
Parallel Programs

Pipeline model computation:



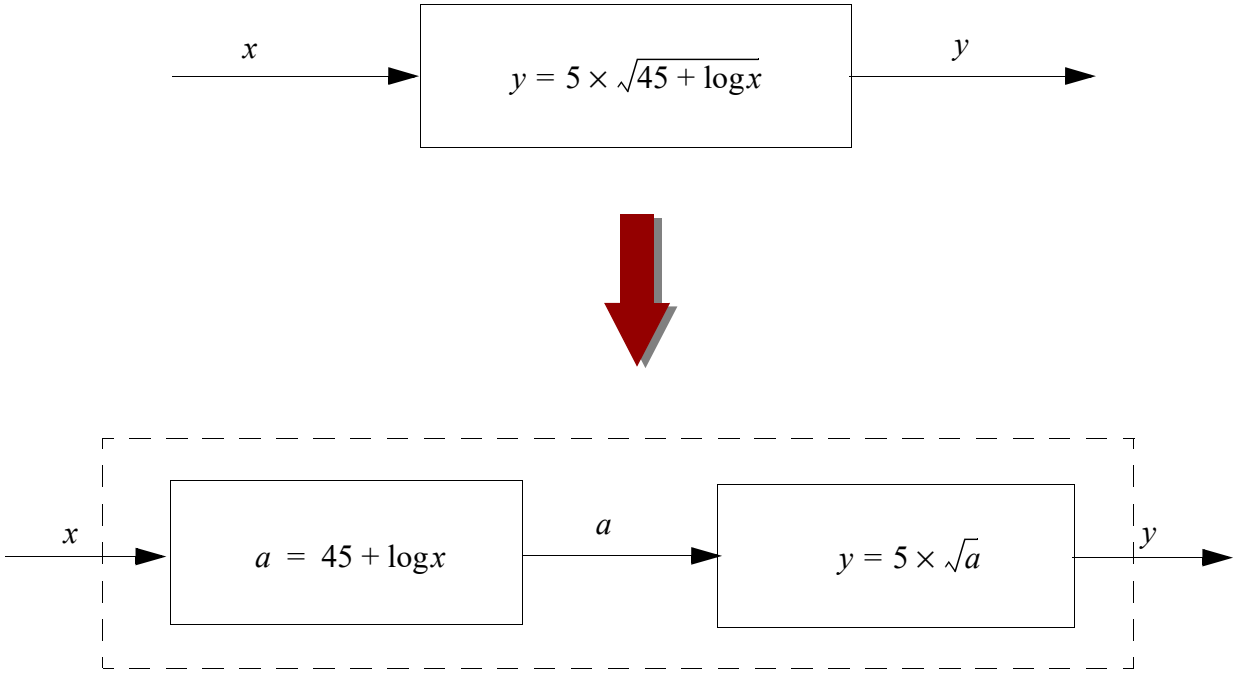
Parallel Programs

Pipeline model computation:



Parallel Programs

Pipeline model computation:



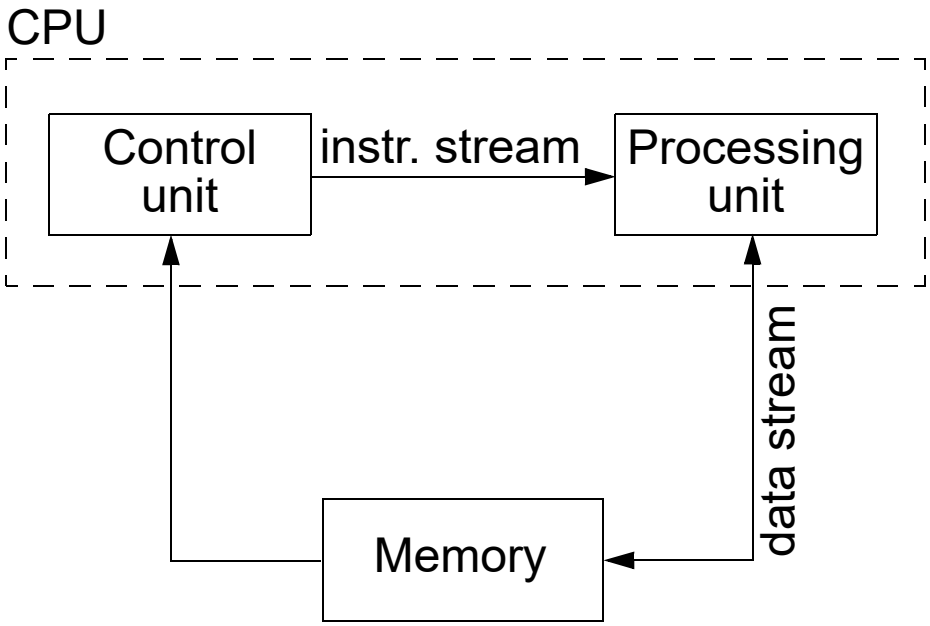

```
channel ch:real;
- - - - - - - - -
cobegin
   var x: real;
   while true do
        read(x);
        send(ch, 45+log(x));
   end while
   |
   var v: real;
   while true do
        receive(ch, v);
        write(5 * sqrt(v));
   end while
coend;
- - - - - - - - - -
```

Flynn's Classification of Computer Architectures

- Flynn's classification is based on the nature of the *instruction flow* executed by the computer and that of the *data flow* on which the instructions operate.

Flynn's Classification of Computer Architectures

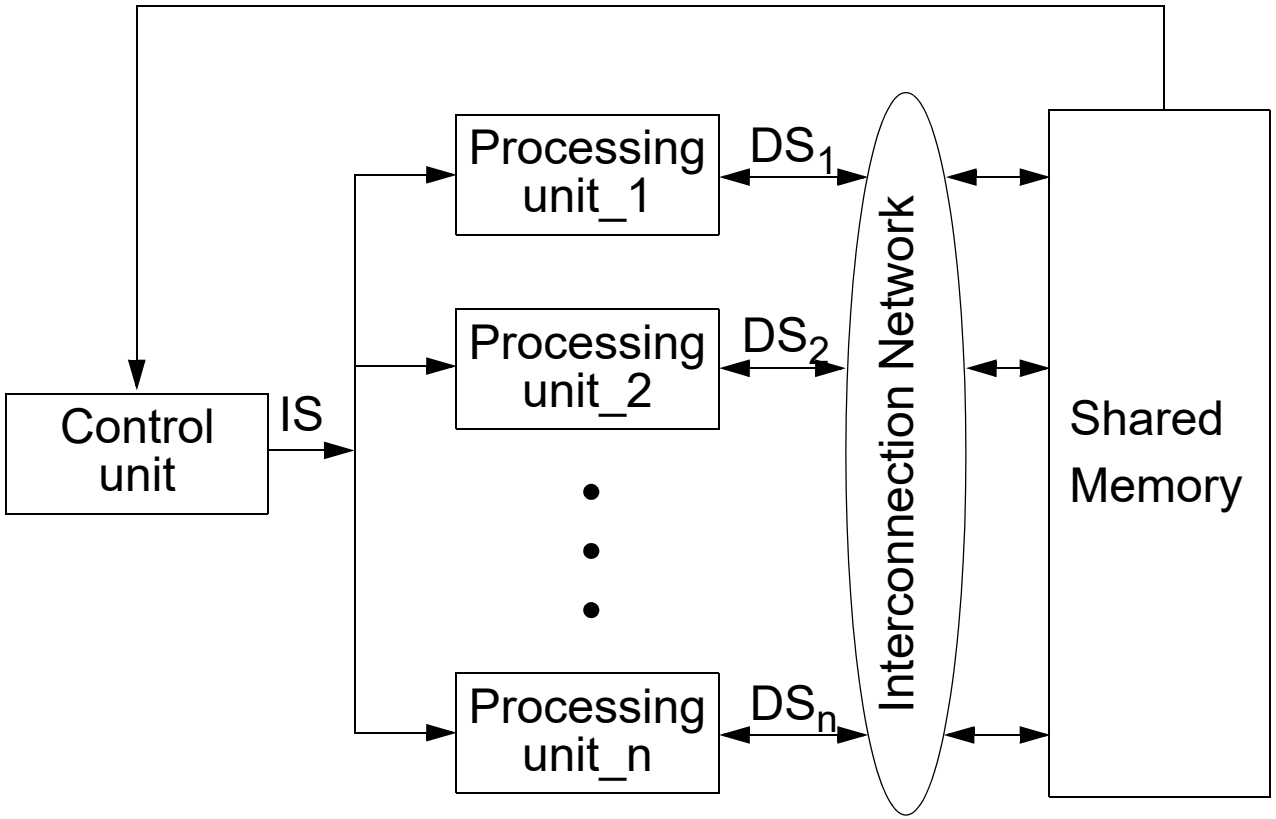
Single Instruction stream, Single Data stream (*SISD*)



Flynn's Classification of Computer Architectures

Single Instruction stream, Multiple Data stream (*SIMD*)

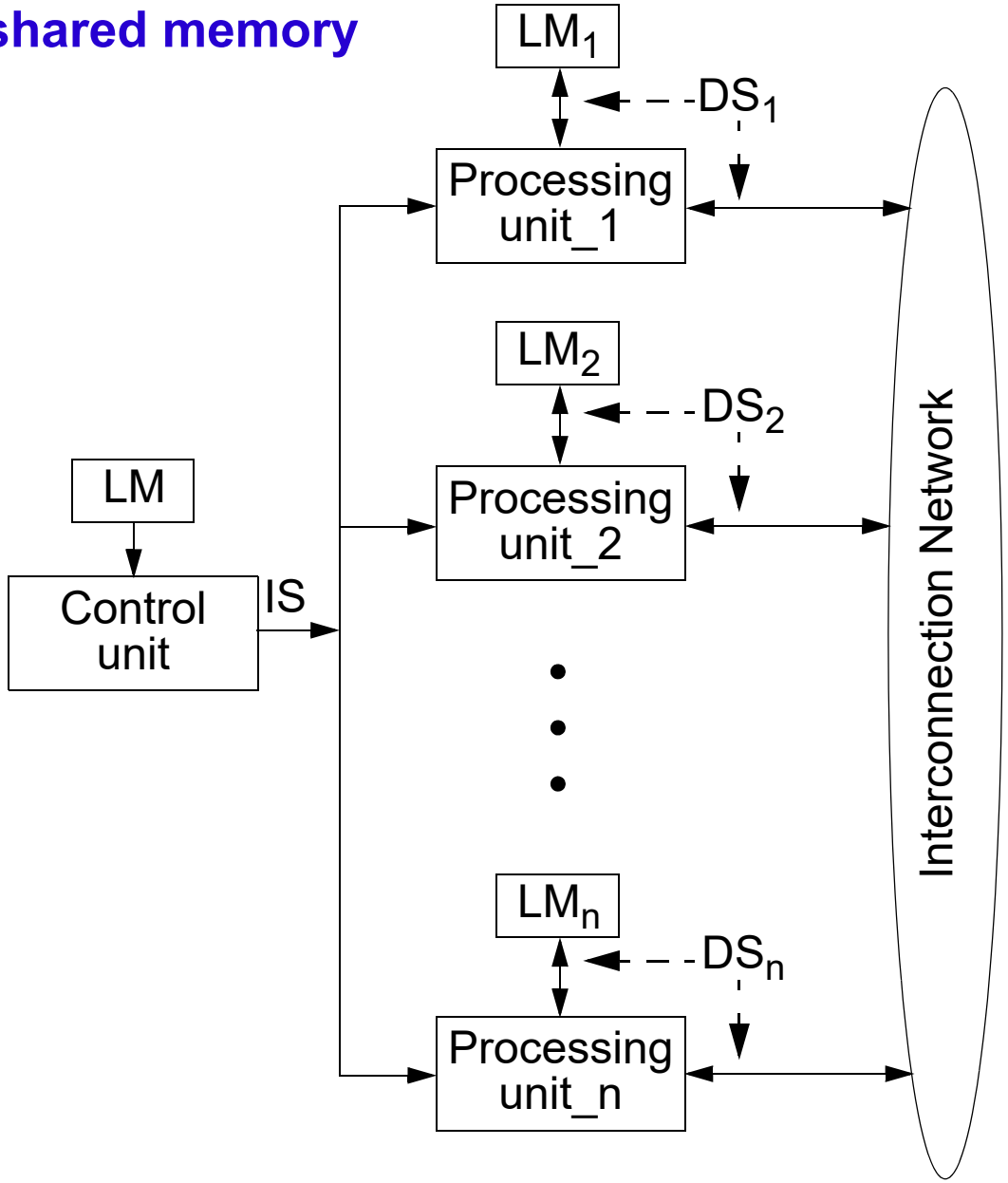
SIMD with shared memory



Flynn's Classification of Computer Architectures

Single Instruction stream, Multiple Data stream (*SIMD*)

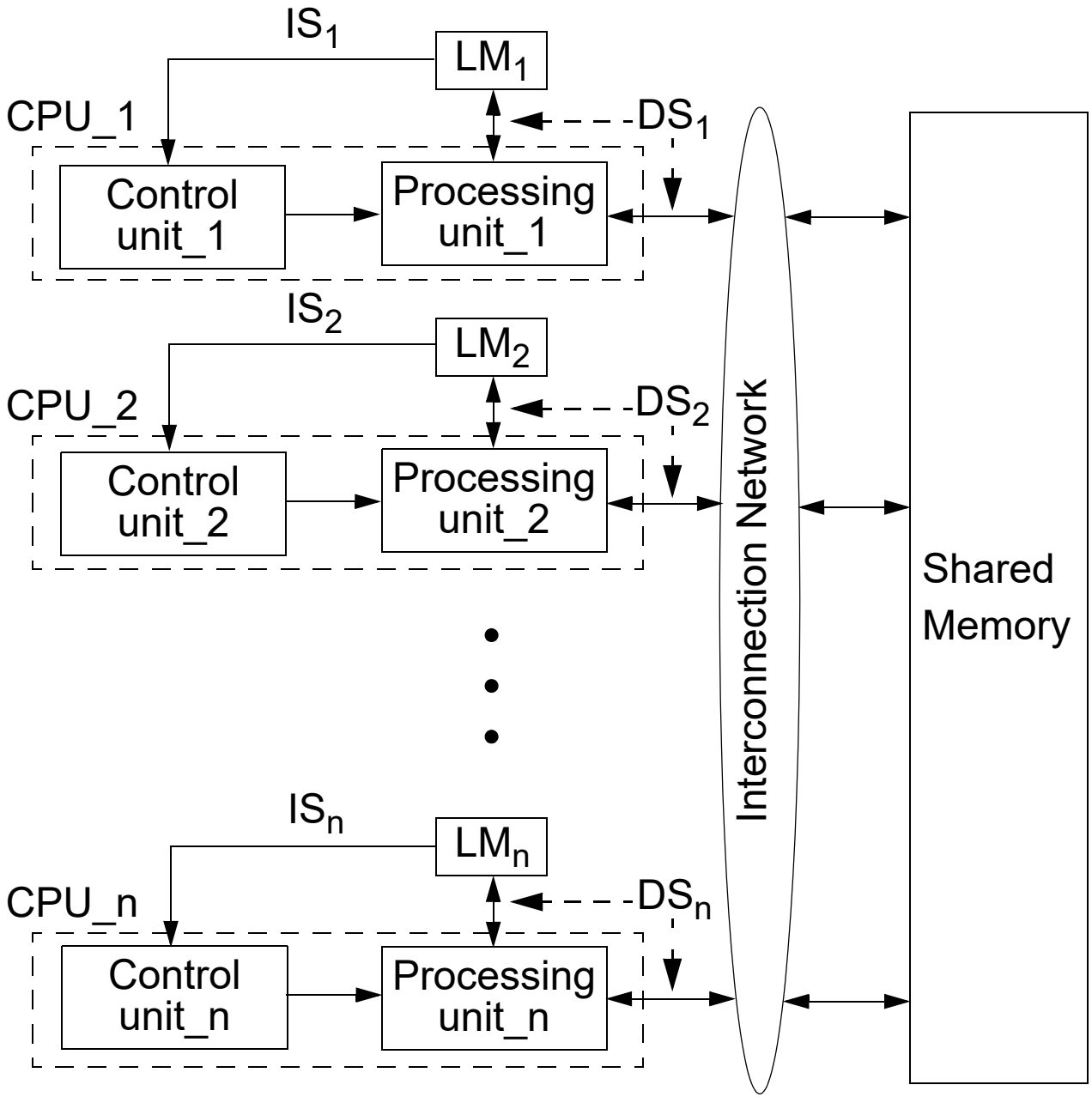
SIMD with *no* shared memory



Flynn's Classification of Computer Architectures

Multiple Instruction stream, Multiple Data stream (*MIMD*)

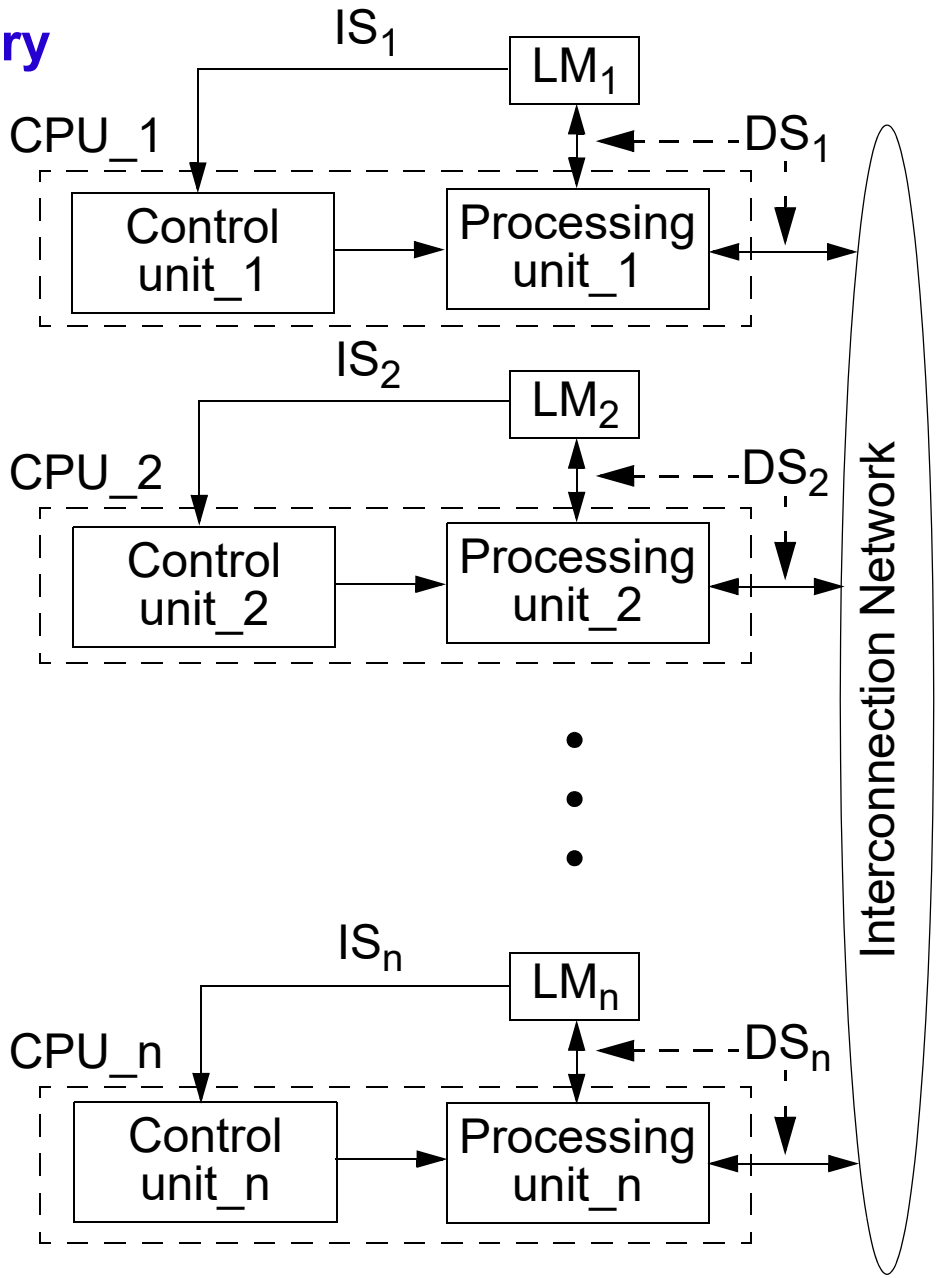
MIMD with shared memory



Flynn's Classification of Computer Architectures

Multiple Instruction stream, Multiple Data stream (*MIMD*)

MIMD with *no* shared memory



Performance of Parallel Architectures

Important questions:

- How fast runs a parallel computer at its *maximal* potential?
- How fast execution can we expect from a parallel computer for a *concrete application*?
- How do we measure the performance of a parallel computer and the performance improvement we get by using such a computer?

Performance Metrics

- Peak rate: the maximal computation rate that can be theoretically achieved when all modules are fully utilized.

  The peak rate is of no practical significance for the user. It is mostly used by vendor companies for marketing of their computers.

Performance Metrics

- **Peak rate**: the maximal computation rate that can be theoretically achieved when all modules are fully utilized.

  The peak rate is of no practical significance for the user. It is mostly used by vendor companies for marketing of their computers.

- **Speedup**: measures the gain we get by using a certain parallel computer to run a given parallel program in order to solve a specific problem.

$$S = \frac{T_S}{T_P}$$

T_S: execution time needed with the best sequential algorithm;
T_P: execution time needed with the parallel algorithm.

Performance Metrics

- **Efficiency: this metric relates the speedup to the number of processors used; by this it provides a measure of the efficiency with which the processors are used.**

$$E = \frac{S}{p}$$

S: speedup;
p: number of processors.

For the ideal situation, *in theory*:

$$S = \frac{\frac{T_S}{T_S}}{p} = p$$ **; which means *E* = 1**

***Practically* the ideal efficiency of 1 can not be achieved!**

Amdahl's Law

- **Consider *f* to be the ratio of computations that, according to the algorithm, have to be executed sequentially ($0 \le f \le 1$); *p* is the number of processors;**

$$T_P = f \times T_S + \frac{(1-f) \times T_S}{p}$$

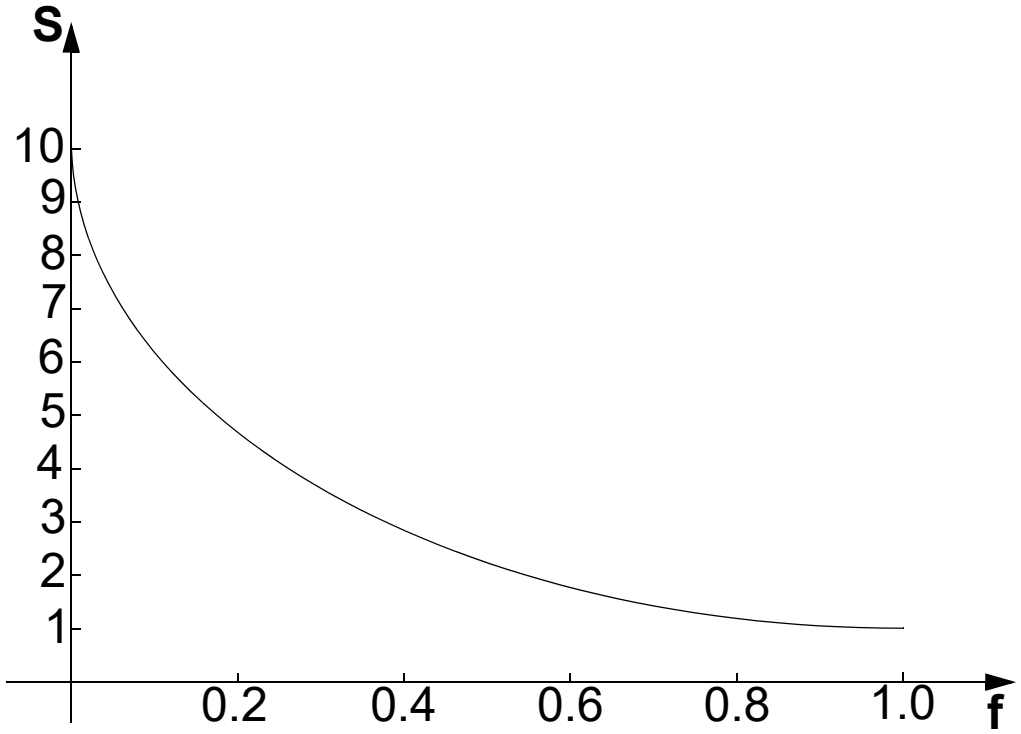
$$S = \frac{T_S}{f \times T_S + (1-f) \times \frac{T_S}{p}} = \frac{1}{f + \frac{(1-f)}{p}}$$

Amdahl's Law

- Consider *f* to be the ratio of computations that, according to the algorithm, have to be executed sequentially ($0 \le f \le 1$); *p* is the number of processors;

$$T_P = f \times T_S + \frac{(1-f) \times T_S}{p}$$

$$S = \frac{T_S}{f \times T_S + (1-f) \times \frac{T_S}{p}} = \frac{1}{f + \frac{(1-f)}{p}}$$



Amdahl's Law

‘

Amdahl's law: even a little ratio of sequential computation imposes a certain limit to speedup; a higher speedup than 1/*f* can not be achieved, regardless the number of processors.

$$E = \frac{S}{P} = \frac{1}{f \times (p-1) + 1}$$

To efficiently exploit a high number of processors, *f* must be small (the algorithm has to be highly parallel).

Other Aspects which Limit the Speedup

- Beside the intrinsic sequentiality of some parts of an algorithm there are also other factors that limit the achievable speedup:
  - communication cost
  - load balancing of processors
  - costs of creating and scheduling processes
  - I/O operations

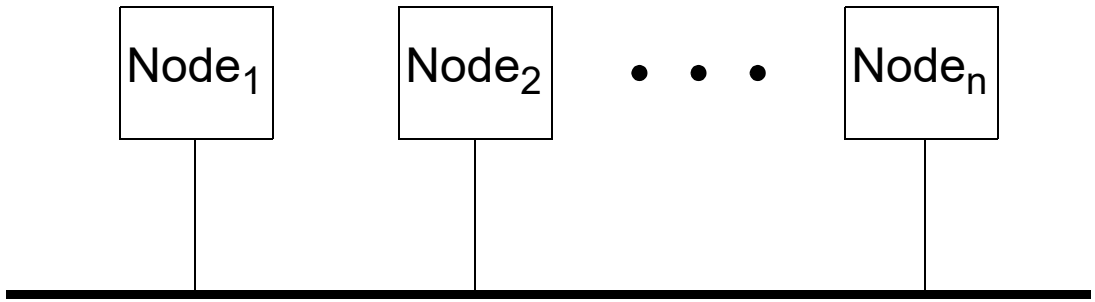
- There are many algorithms with a high degree of parallelism; for such algorithms the value of *f* is very small and can be ignored. These algorithms are suited for massively parallel systems; in such cases the other limiting factors, like the cost of communications, become critical.

The Interconnection Network

- The interconnection network (IN) is a key component of the architecture. It has a decisive influence on the overall performance and cost.
- The traffic in the IN consists of data transfer and transfer of commands and requests.
- The key parameters of the IN are
  - total bandwidth: transferred bits/second
  - cost

The Interconnection Network

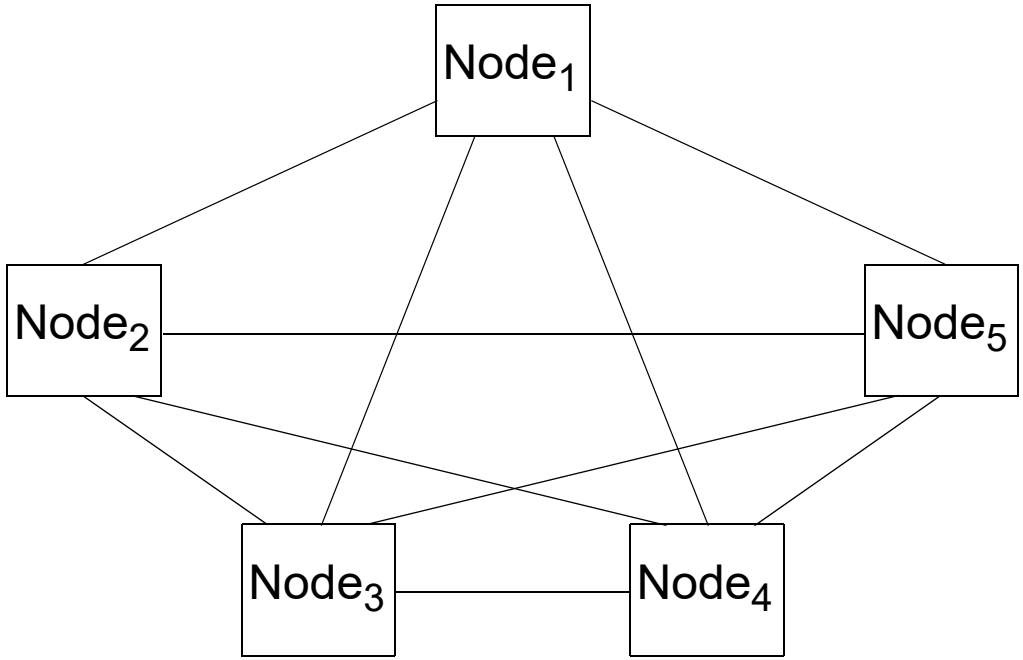
Single Bus



- Single bus networks are simple and cheap.
- One single communication allowed at a time; bandwidth shared by all nodes.
- Performance is relatively poor.
- In order to keep performance, the number of nodes is limited (16 - 20).

The Interconnection Network

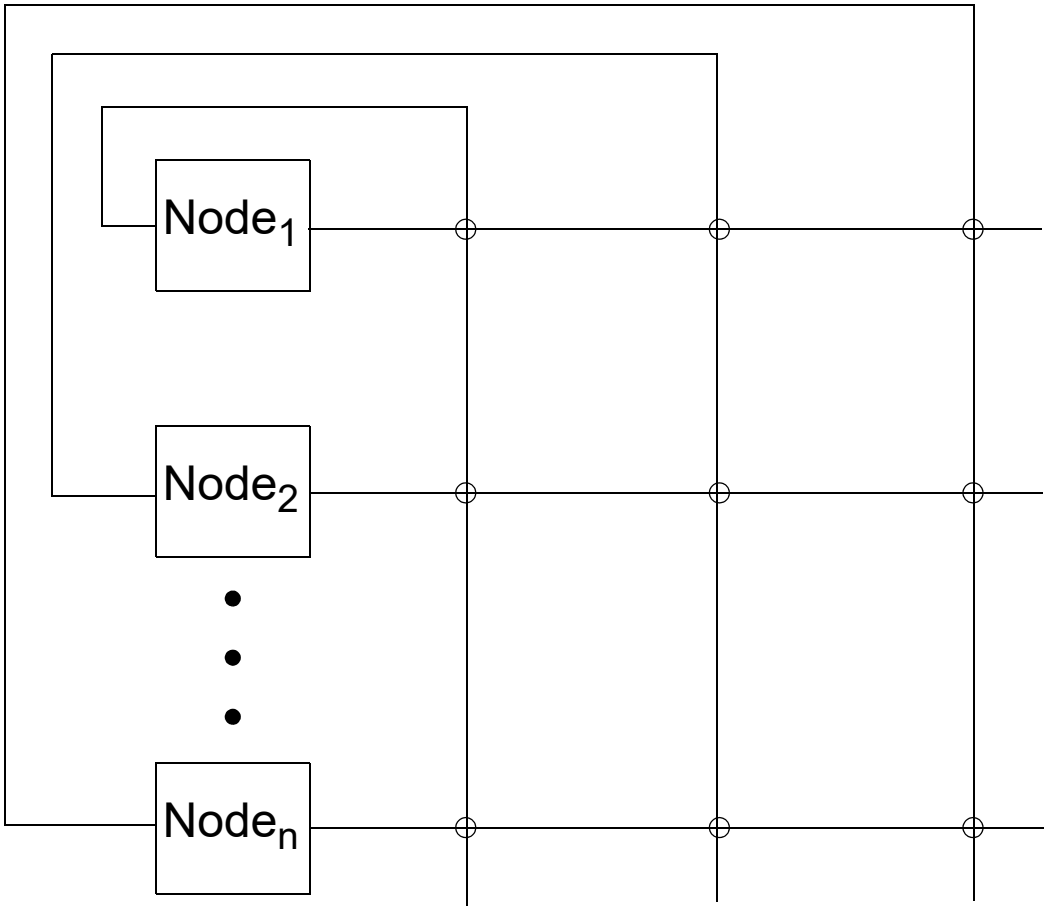
Completely connected network



- Each node is connected to every other one.
- Communications can be performed in parallel between any pair of nodes.
- Both performance and cost are high.
- Cost increases rapidly with number of nodes.

The Interconnection Network

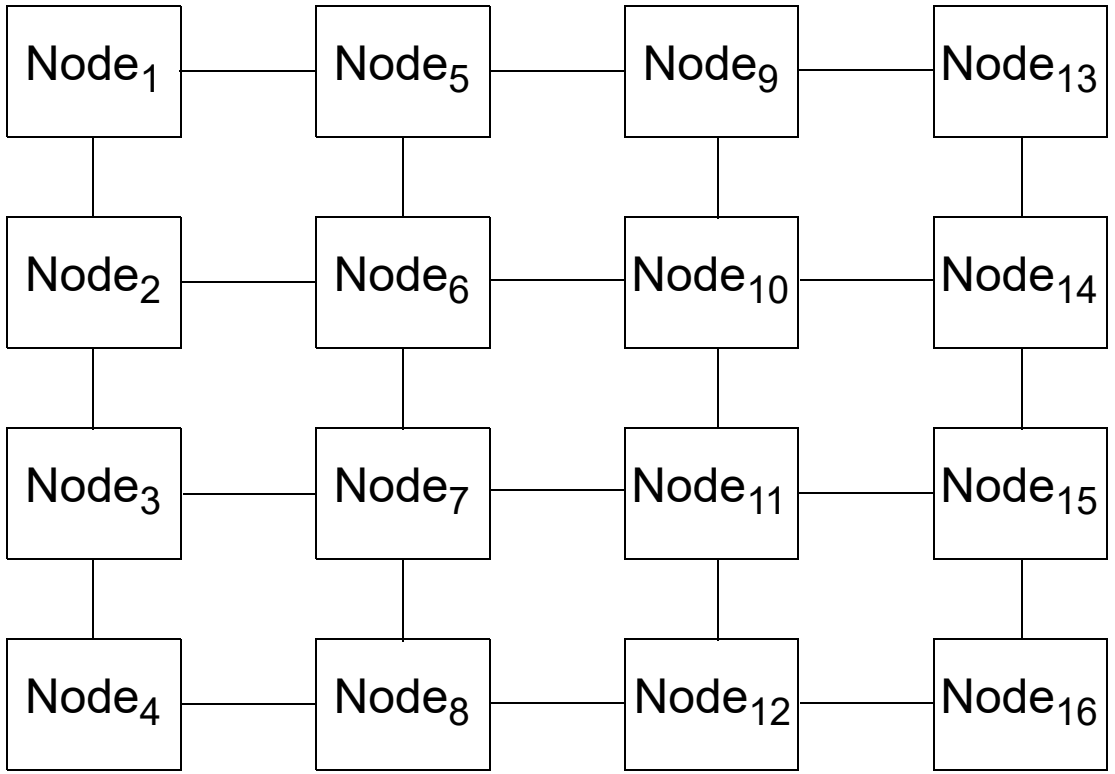
Crossbar network



- The crossbar is a *dynamic network*: the interconnection topology can be modified by positioning of switches.
- The crossbar network is completely connected: any node can be directly connected to any other.
- Fewer interconnections are needed than for the static completely connected network; however, a large number of switches is needed.
- Several communications can be performed in parallel.

The Interconnection Network

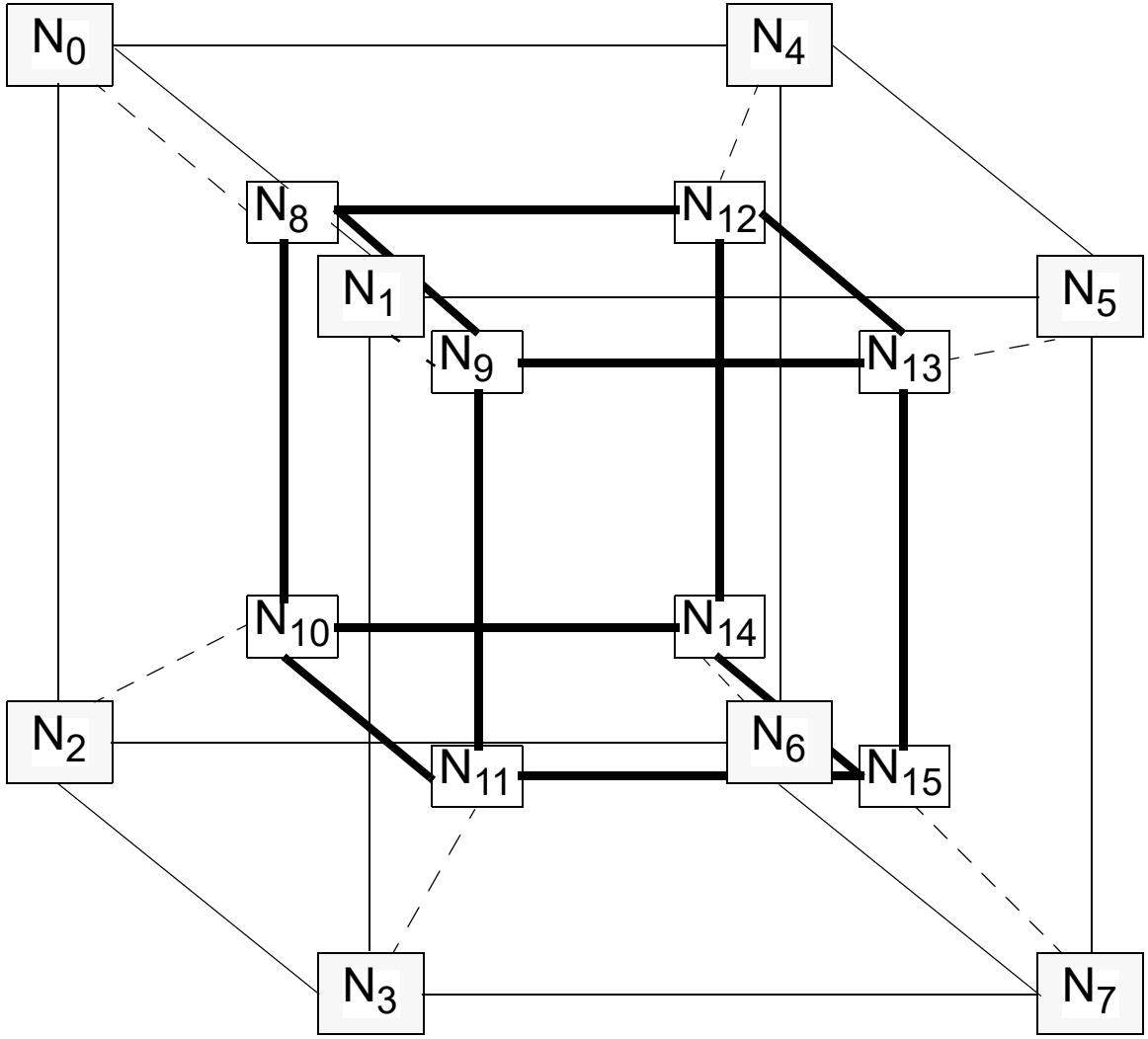
Mesh network



- Mesh networks are cheaper than completely connected ones and provide relatively good performance.
- In order to transmit an information between certain nodes, routing through intermediate nodes is needed (max. 2*(*n*-1) intermediates for an *n***n* mesh).
- It is possible to provide wraparound connections: between nodes 1 and 13, 2 and 14, etc.
- Three dimensional meshes have been also implemented.

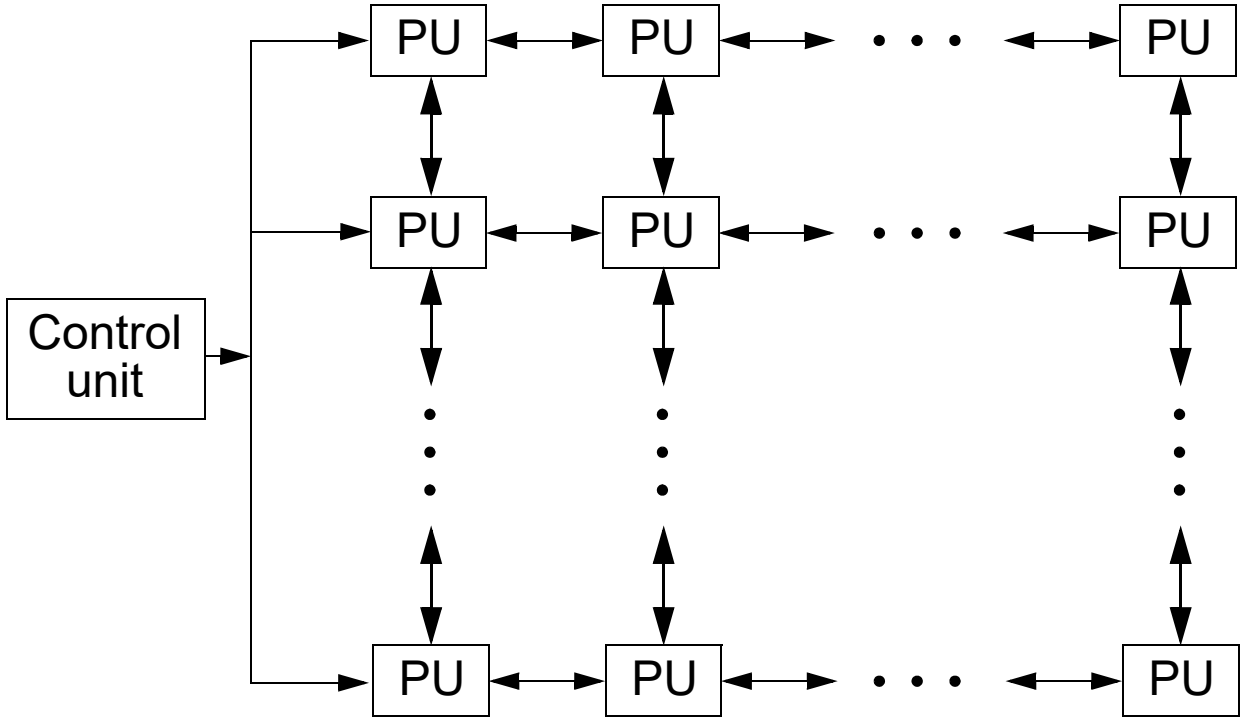
The Interconnection Network

Hypercube network



- 2^n nodes are arranged in an *n*-dimensional cube. Each node is connected to *n* neighbours.
- In order to transmit an information between certain nodes, routing through intermediate nodes is needed (maximum *n* intermediates).

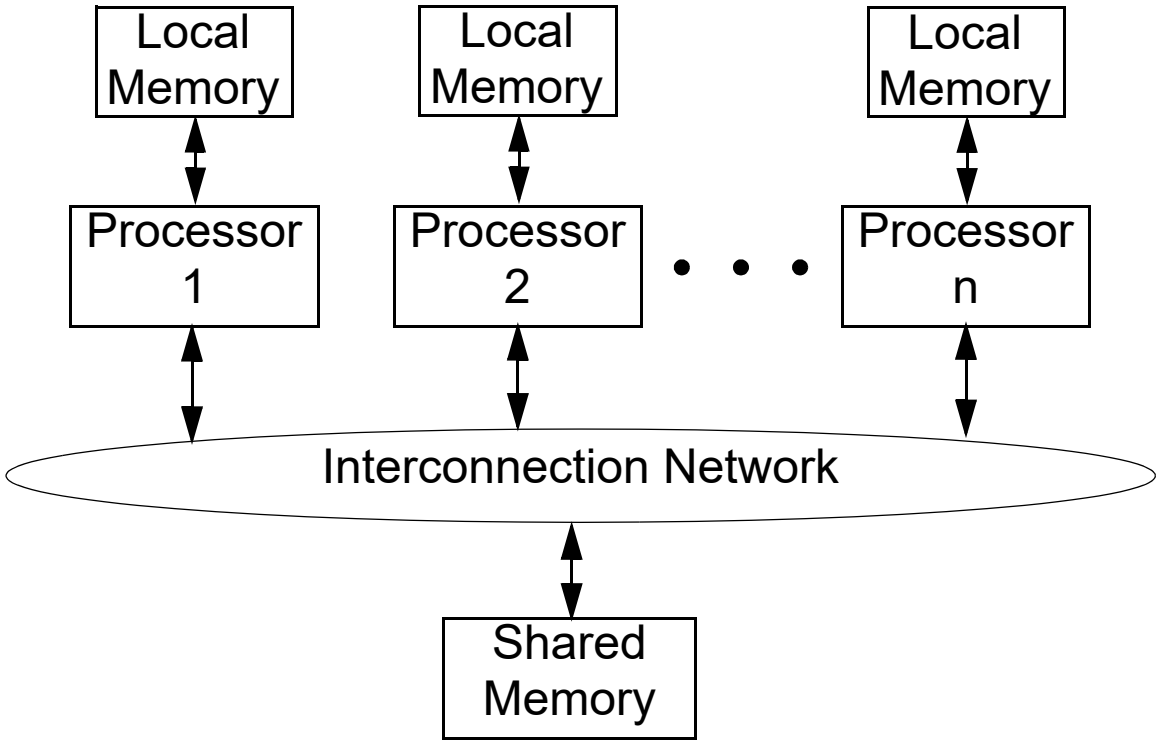
SIMD Computers



- SIMD computers are usually called ***array processors***.
- PU's are very simple: an ALU which executes the instruction broadcast by the CU, a few registers, and some local memory.
- The first SIMD computer: ILLIAC IV (1970s), 64 relatively powerful processors (mesh connection, see above).
- Newer SIMD computer: CM-2 (Connection Machine, by Thinking Machines Corporation, 65 536 very simple processors (connected as hypercube).
- Array processors are specialized for numerical problems formulated as matrix or vector calculations. Each PU computes one element of the result.

MIMD computers

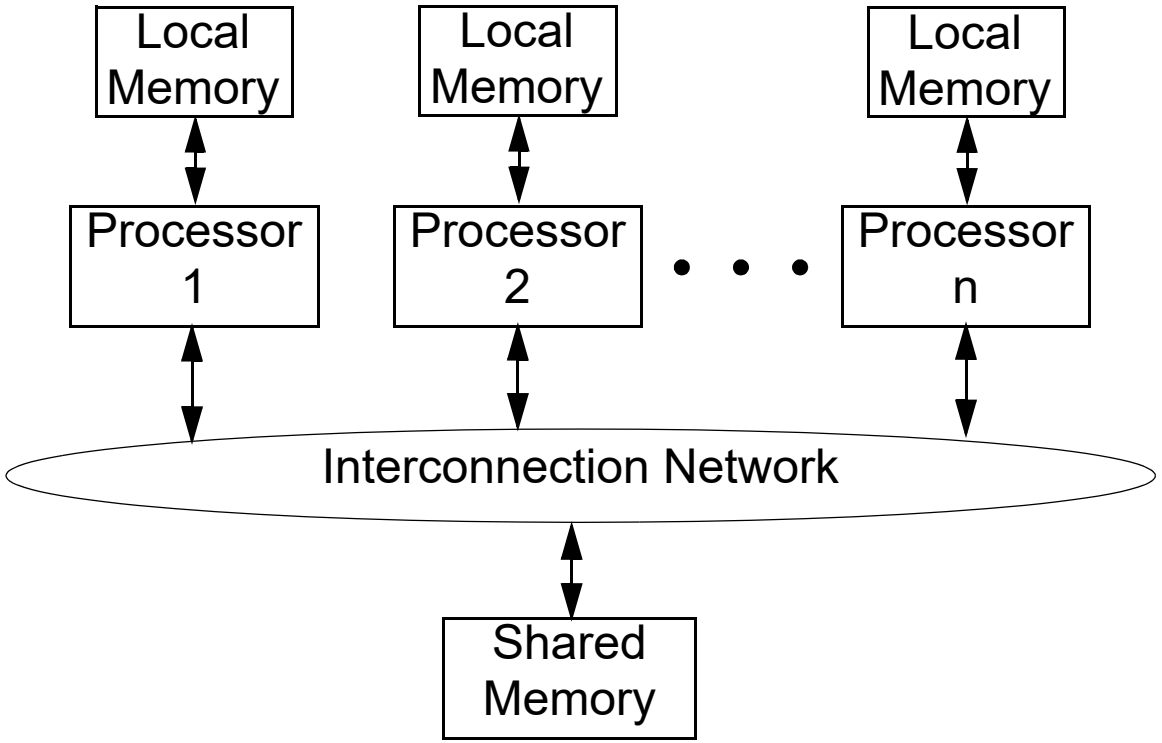
MIMD with *shared memory*



- **Communication between processors is through *shared memory*. One processor can change the value in a location and the other processors can read the new value.**
- **With many processors, memory contention seriously degrades performance $\Rightarrow$ such architectures don't support a high number of processors.**

MIMD computers

MIMD with *shared memory*

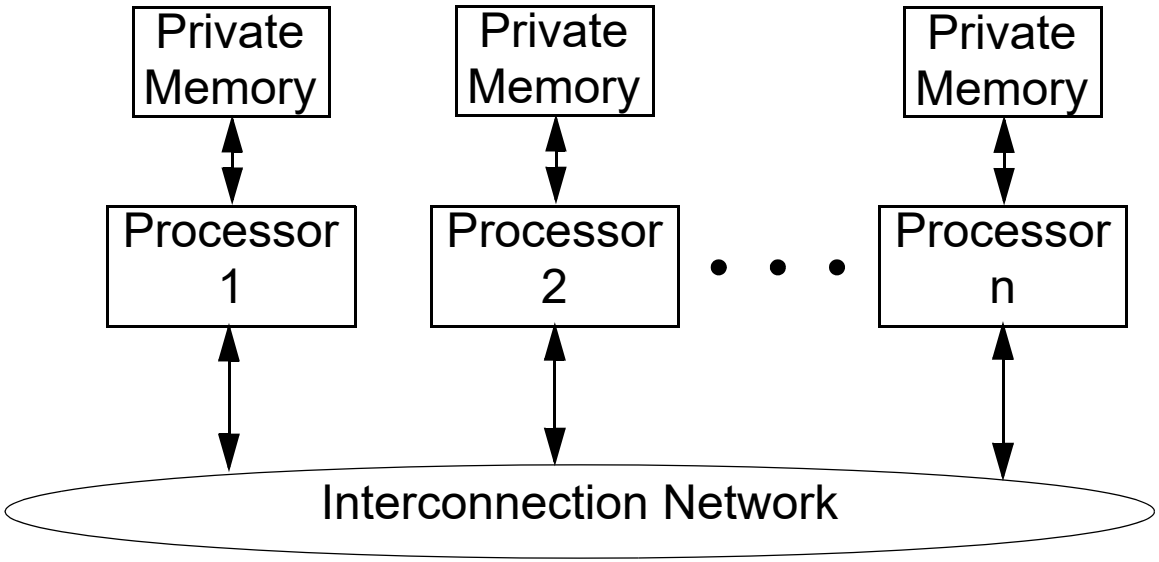


- Classical parallel mainframe computers (1970-1980-1990):
  - IBM 370/390 Series
  - CRAY X-MP, CRAY Y-MP, CRAY 3
- Modern multicore chips:
  - Intel Core Duo, i5, i7; Arm MPC

- Communication between processors is through *shared memory*. One processor can change the value in a location and the other processors can read the new value.
- With many processors, memory contention seriously degrades performance $\Rightarrow$ such architectures don't support a high number of processors.

MIMD computers

MIMD with *no shared memory*



- **Communication between processors is only by *passing messages* over the interconnection network.**
- **There is no competition of the processors for the shared memory $\Rightarrow$ the number of processors is not limited by memory contention.**
- **The speed of the interconnection network is an important parameter for the overall performance.**
- ***Modern large parallel computers do not have a system-wide shared memory*.**

Muticore Architectures
The Parallel Computer in Your Pocket

- **Multicore chips**:

  Several processors on the same chip.

  A parallel computer on a chip.

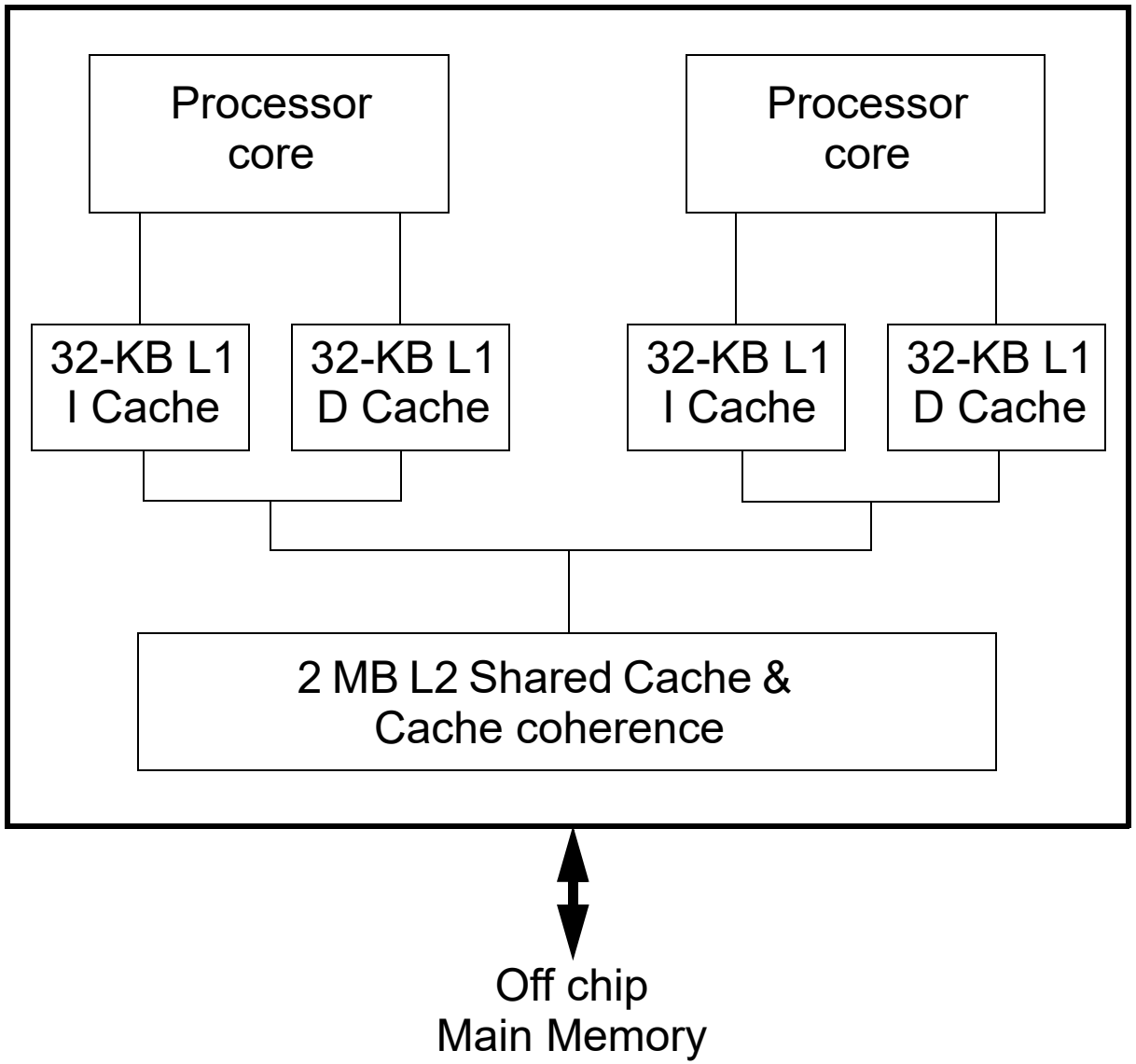
- This is the only way to increase chip performance without excessive increase in power consumption:
  - Instead of increasing processor frequency, use several processors and run each at lower frequency.

Examples:

- Intel x86 Multicore architectures
  - Intel Core Duo
  - Intel Core i7
- ARM11 MPCore

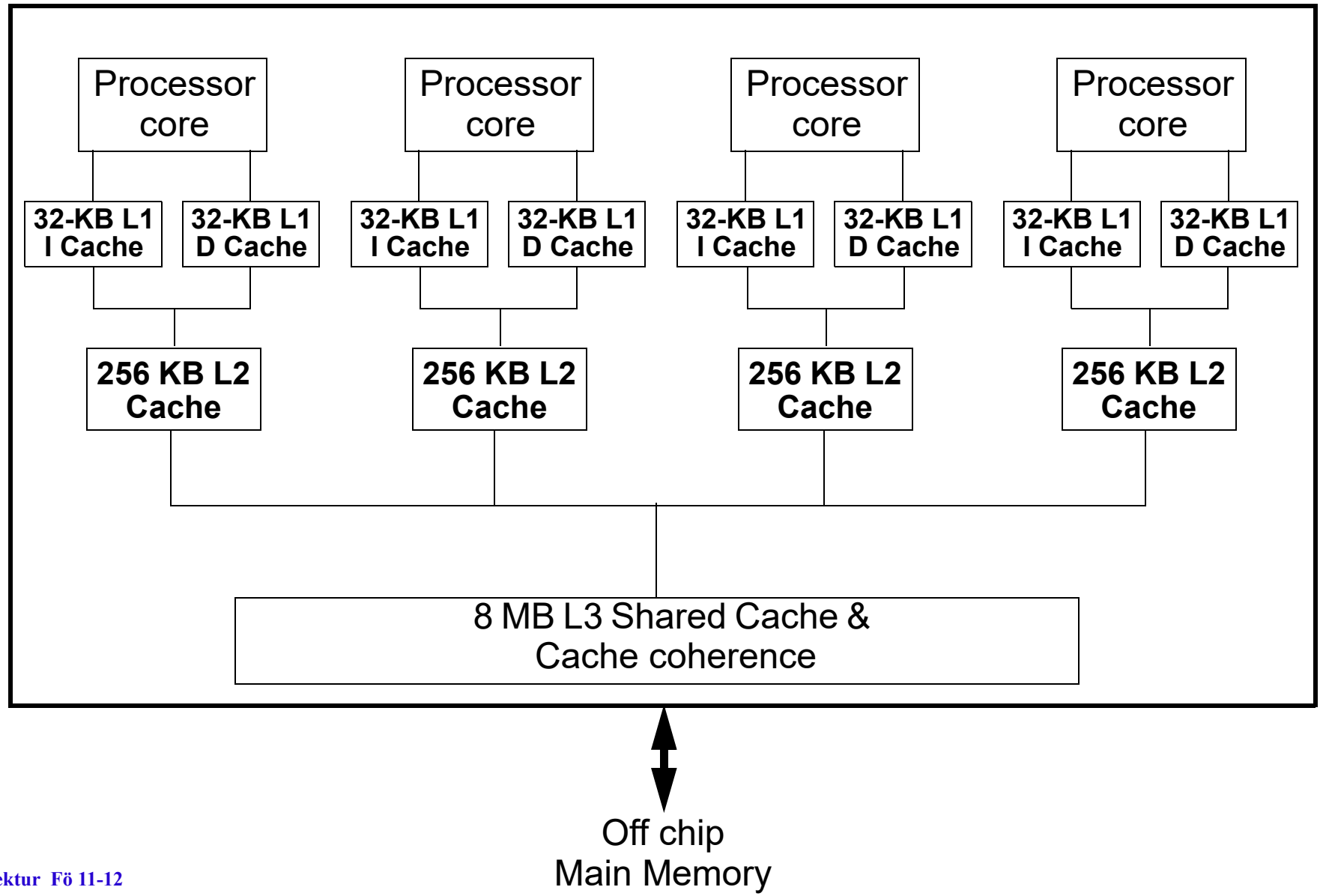
Intel Core Duo

Composed of two *Intel Core* superscalar processors

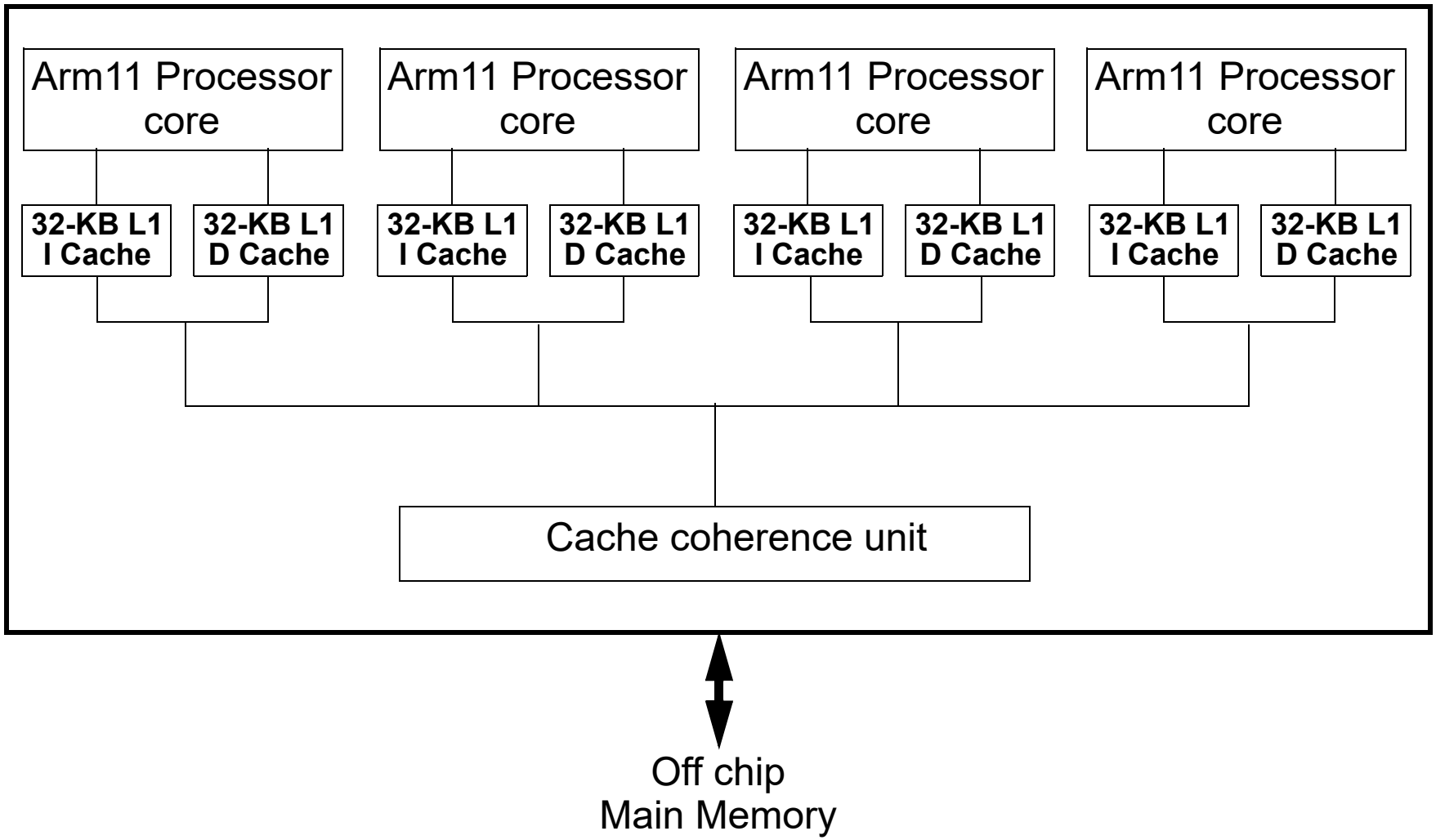


Intel Core i7

Contains four *Nehalem* processors.

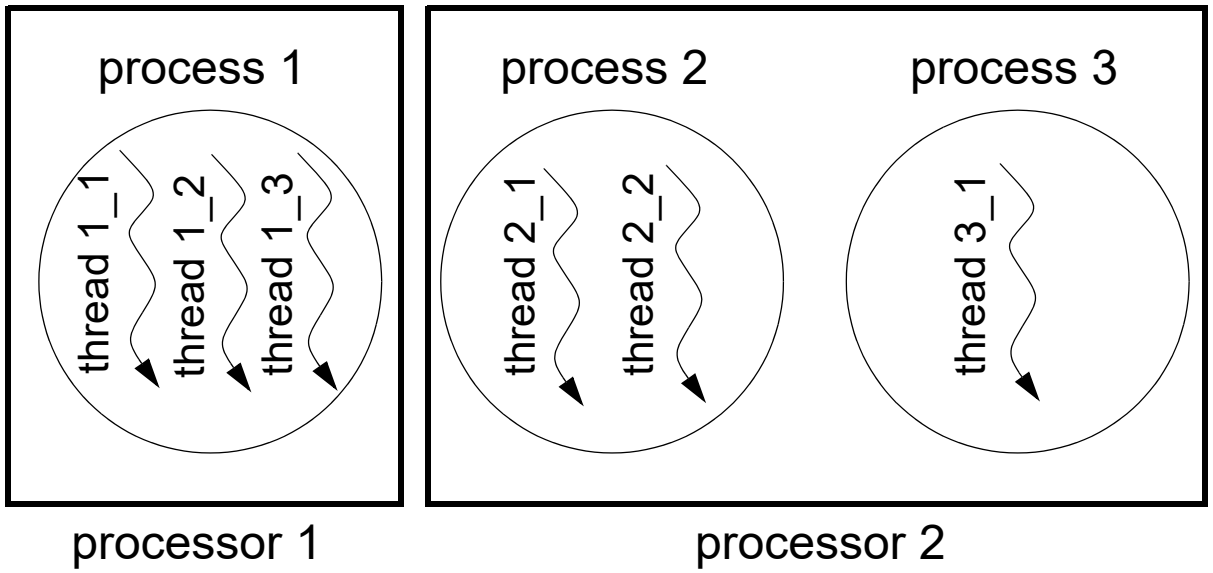


ARM11 MPCore



Multithreading

- **A running program:**
  - **one or several *processes*; each process:**
    - **one or several *threads***



  - ***thread*: a piece of sequential code executed in parallel with other threads.**

Multithreading

- Several threads can be active simultaneously on the same processor.
  - Typically, the Operating System is scheduling threads on the processor.
  - The OS is switching between threads so that one thread is active (running) on a processor at a time.

- Switching between threads implies saving/restoring the Program Counter, Registers, Status flags, etc.

Switching overhead is considerable!

Hardware Multithreading

- Multithreaded processors provide hardware support for executing multithreaded code:
  - separate program counter & register set for individual threads;
  - instruction fetching on thread basis;
  - hardware supported context switching.

- Efficient execution of multithread software.
- Efficient utilisation of processor resources.

Hardware Multithreading

- Multithreaded processors provide hardware support for executing multithreaded code:
  - separate program counter & register set for individual threads;
  - instruction fetching on thread basis;
  - hardware supported context switching.

- Efficient execution of multithread software.
- Efficient utilisation of processor resources.

- By handling several threads:
  - There is greater chance to find instructions to execute in parallel on the available resources.
  - When one thread is blocked, due to e.g. memory access or data dependencies, instructions from another thread can be executed.
- Multithreading can be implemented on both scalar and superscalar processors.

Approaches to Multithreaded Execution

- **Interleaved multithreading**:

  The processor switches from one thread to another at each clock cycle; if any thread is blocked due to dependency or memory latency, it is skipped and an instruction from a ready thread is executed.

Approaches to Multithreaded Execution

- **Interleaved multithreading**:

  The processor switches from one thread to another at each clock cycle; if any thread is blocked due to dependency or memory latency, it is skipped and an instruction from a ready thread is executed.

- **Blocked multithreading**:

  Instructions of the same thread are executed until the thread is blocked; blocking of a thread triggers a switch to another thread ready to execute.

Approaches to Multithreaded Execution

- **Interleaved multithreading**:

  The processor switches from one thread to another at each clock cycle; if any thread is blocked due to dependency or memory latency, it is skipped and an instruction from a ready thread is executed.

- **Blocked multithreading**:

  Instructions of the same thread are executed until the thread is blocked; blocking of a thread triggers a switch to another thread ready to execute.

Interleaved and blocked multithreading can be applied to *both scalar and superscalar* processors. When applied to superscalars, several instructions are issued simultaneously. *However, all instructions issued during a cycle have to be from the same thread.*

Approaches to Multithreaded Execution

- **Interleaved multithreading**:

  The processor switches from one thread to another at each clock cycle; if any thread is blocked due to dependency or memory latency, it is skipped and an instruction from a ready thread is executed.

- **Blocked multithreading**:

  Instructions of the same thread are executed until the thread is blocked; blocking of a thread triggers a switch to another thread ready to execute.

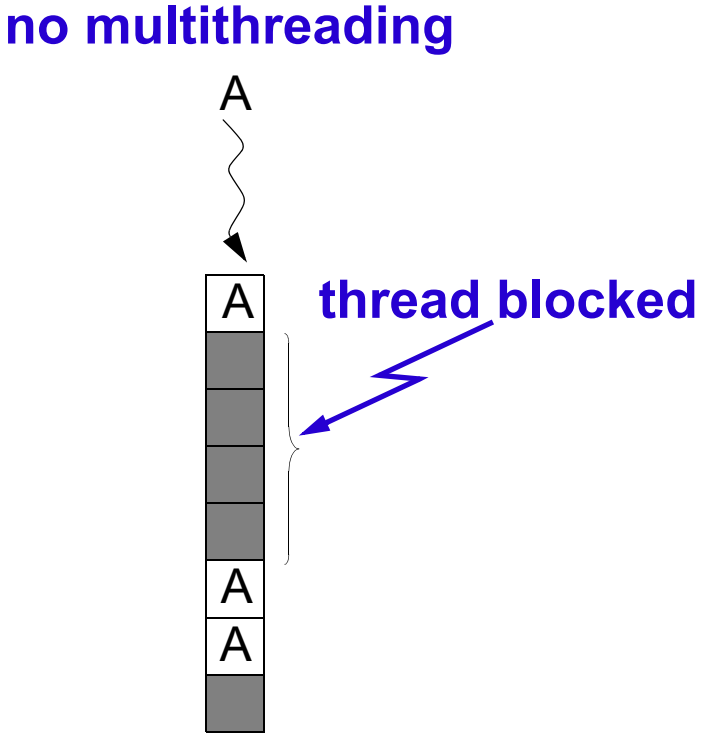
Interleaved and blocked multithreading can be applied to *both scalar and superscalar* processors. When applied to superscalars, several instructions are issued simultaneously. *However, all instructions issued during a cycle have to be from the same thread.*

- **Simultaneous multithreading (SMT)**:

  ***Applies only to superscalar processors*.**
  Several instructions are fetched during a cycle and they can belong to different threads.

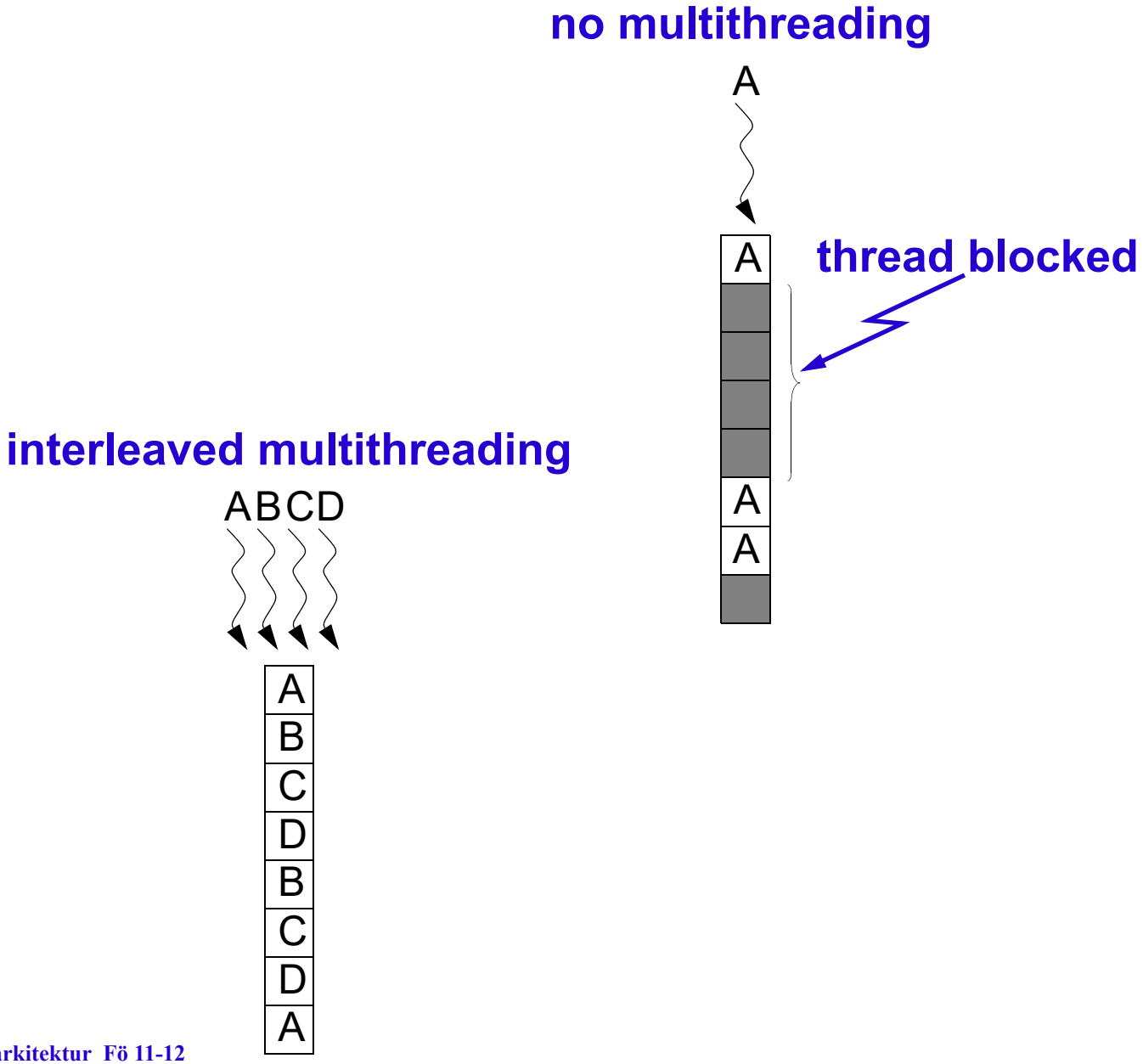
Approaches to Multithreaded

Scalar (non-superscalar) processors



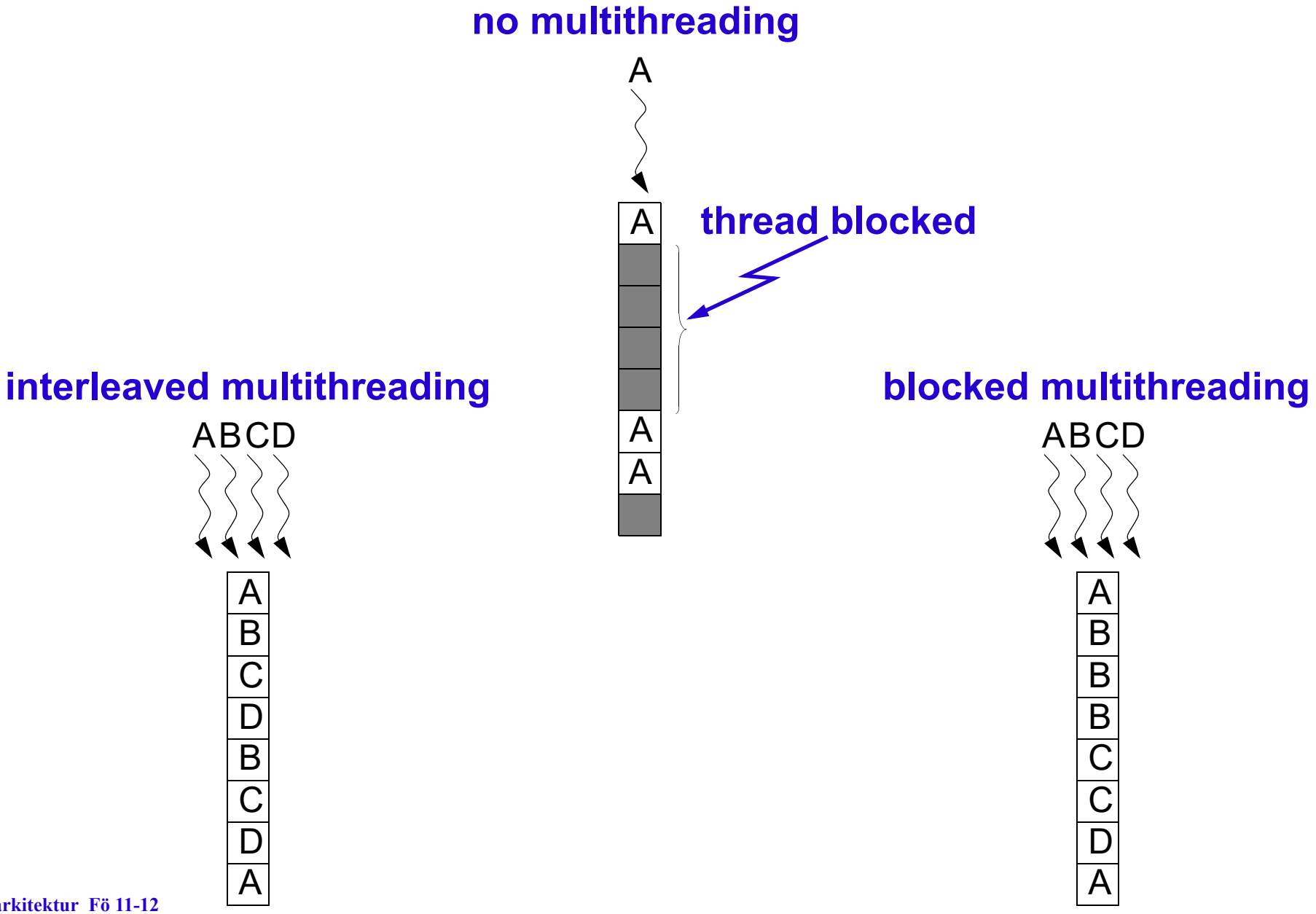
Approaches to Multithreaded

Scalar (non-superscalar) processors



Approaches to Multithreaded

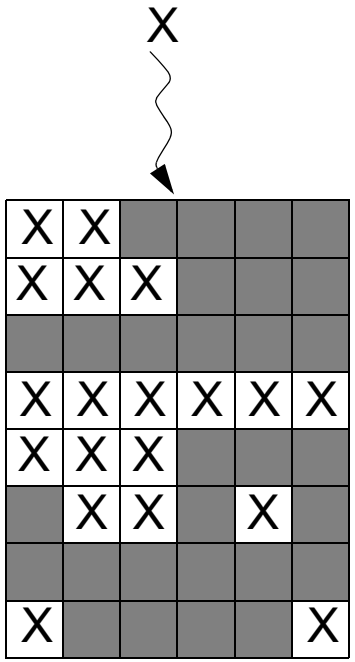
Scalar (non-superscalar) processors



Approaches to Multithreaded

Superscalar processors

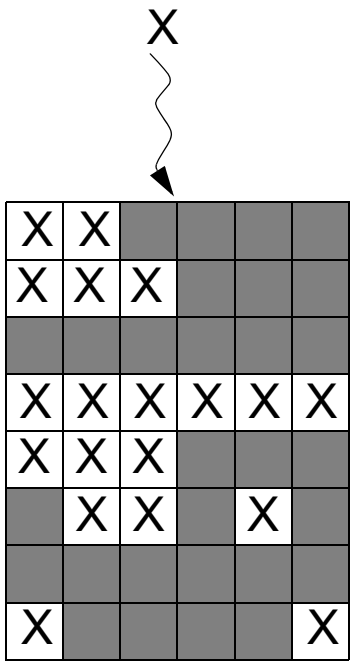
no multithreading



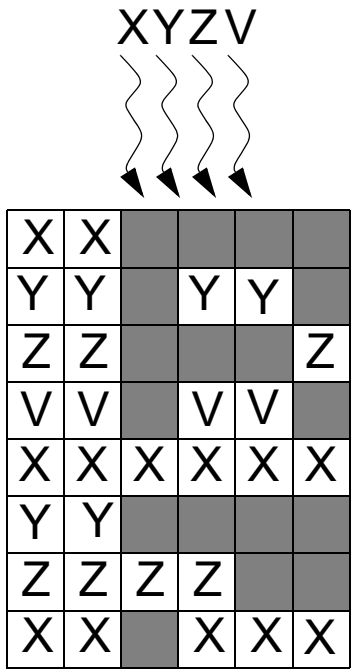
Approaches to Multithreaded

Superscalar processors

no multithreading



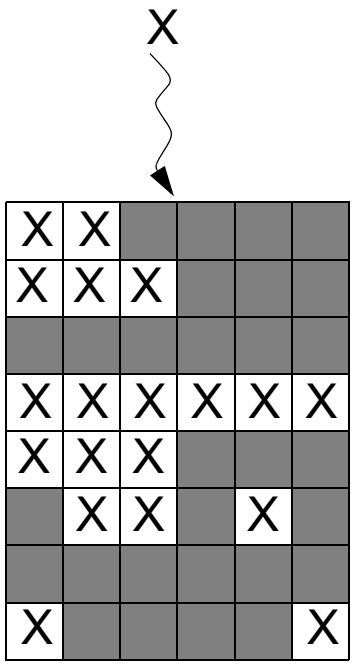
interleaved multithreading



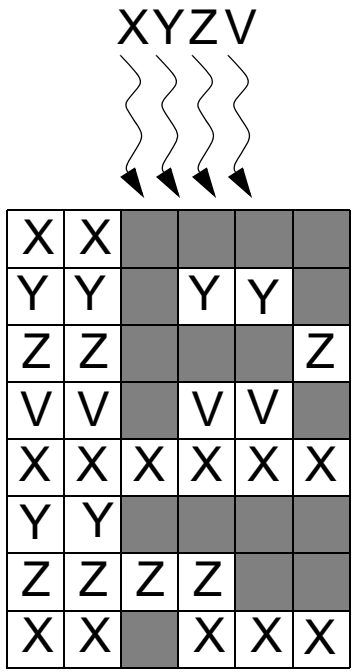
Approaches to Multithreaded

Superscalar processors

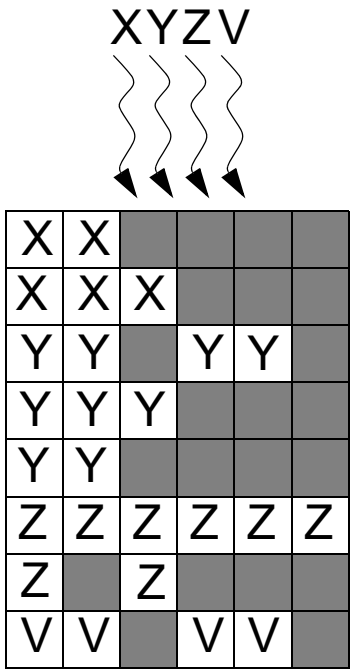
no multithreading



interleaved multithreading



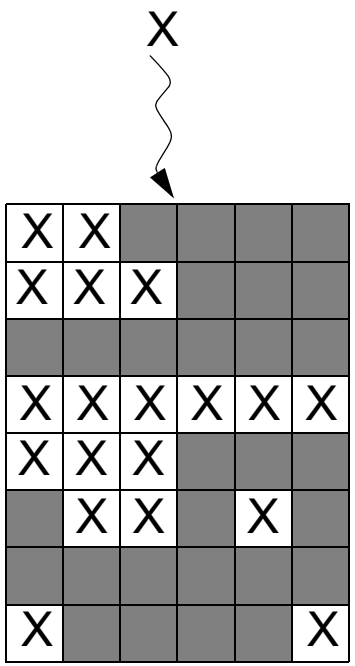
blocked multithreading



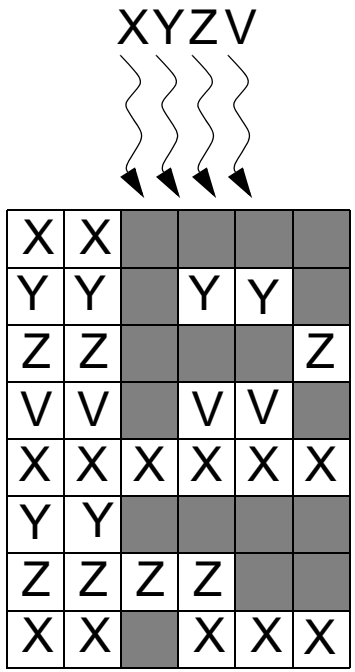
Approaches to Multithreaded

Superscalar processors

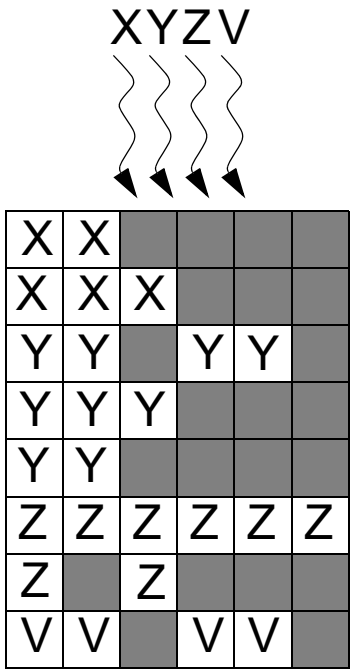
no multithreading



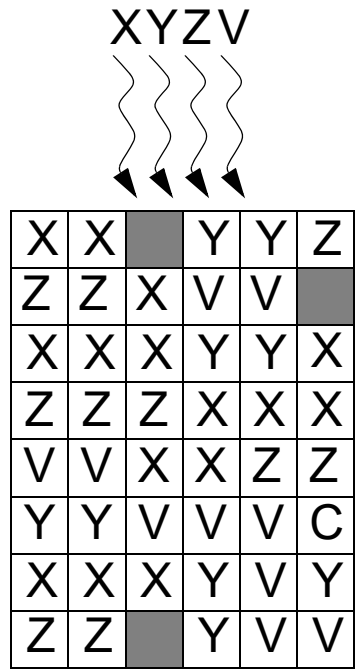
interleaved multithreading



blocked multithreading



simultaneous multithreading



Multithreaded Processors

Are multithreaded processors parallel computers?

Multithreaded Processors

Are multithreaded processors parallel computers?

- **Yes:**
  - **they execute parallel threads;**
  - **certain sections of the processor are available in several copies (e.g. program counter, instruction registers + other registers);**
  - **the processor appears to the operating system as several processors.**

Multithreaded Processors

Are multithreaded processors parallel computers?

- **Yes:**
  - **they execute parallel threads;**
  - **certain sections of the processor are available in several copies (e.g. program counter, instruction registers + other registers);**
  - **the processor appears to the operating system as several processors.**

- **No:**
  - **only certain sections of the processor are available in several copies but we do not have several processors; the execution resources (e.g. functional units) are common.**

  In fact, a single physical processor *appears* as multiple logical processors to the operating system.

Multithreaded Processors

- IBM Power5, Power6:
  - simultaneous multithreading;
  - two threads/core;
  - both power5 and power6 are dual core chips.

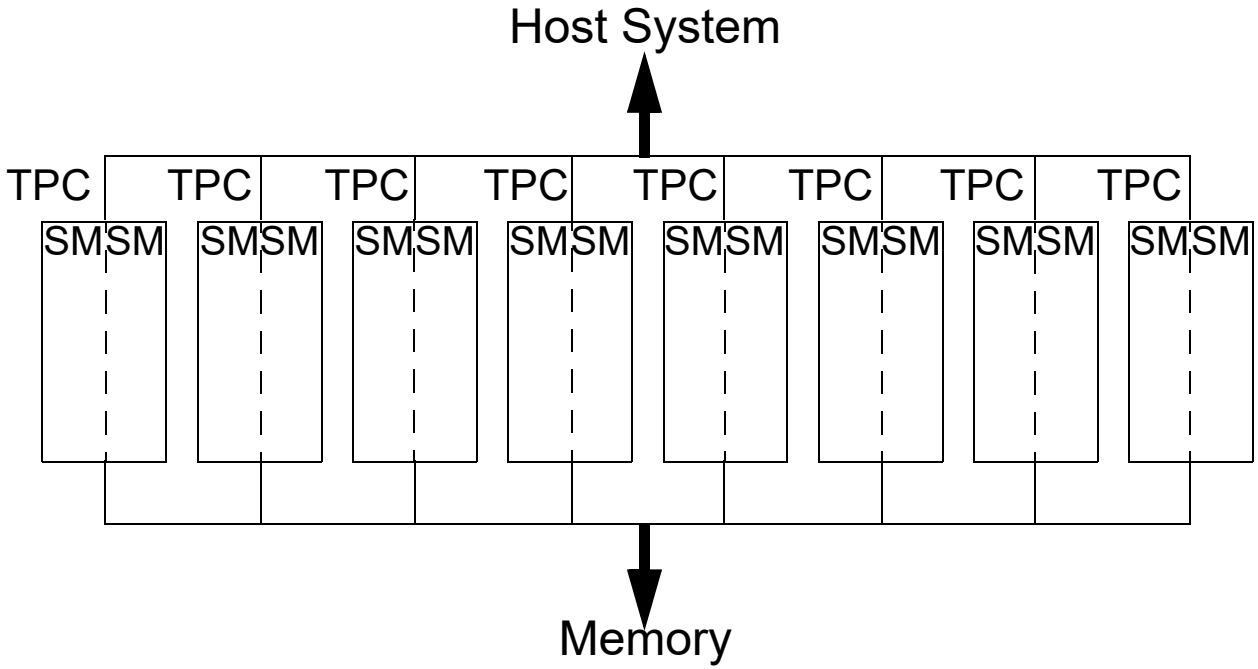
- Intel Montecito (Itanium 2 family):
  - blocked multithreading (called by Intel *temporal multithreading*);
  - two threads/core;
  - Itanium 2 processors are dual core.

- Intel Pentium 4, Nehalem
  - Pentium 4 was the first Intel processor to implement multithreading;
  - simultaneous multithreading (called by Intel *Hyperthreading*);
  - two threads/core (8 simultaneous threads per quad core);

General Purpose GPUs

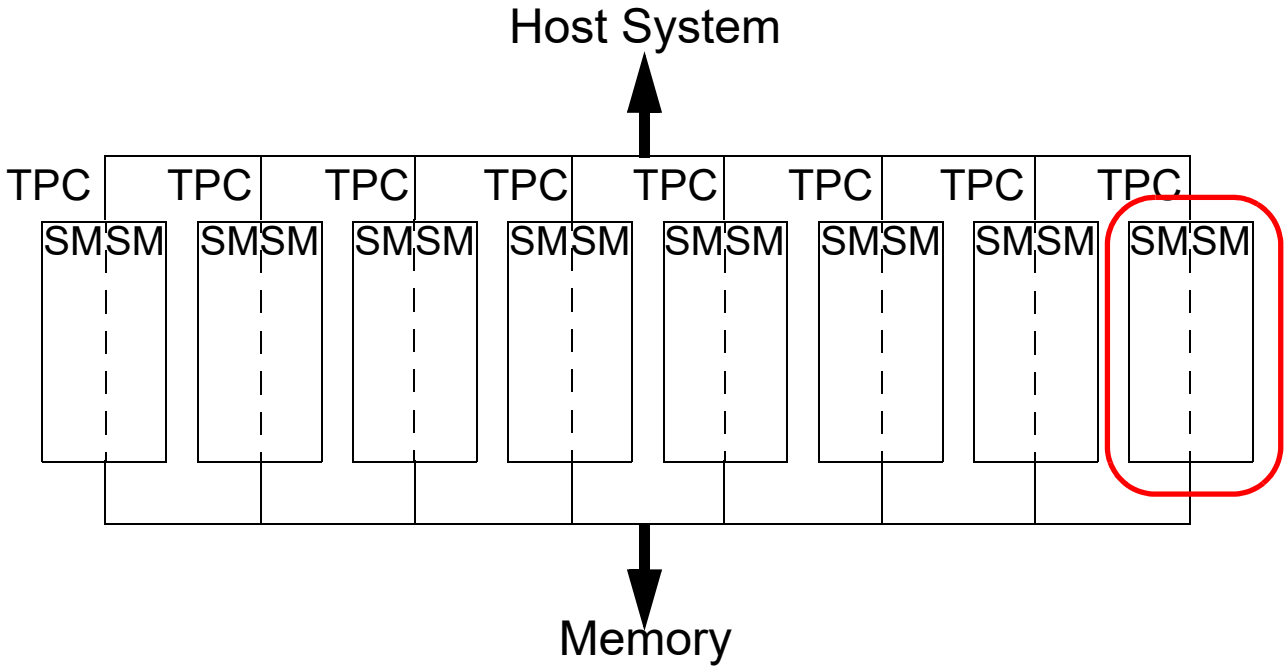
- The first GPUs (graphic processing units) were *non-programmable* 3D-graphic accelerators.
- Today's GPUs are highly *programmable* and efficient.
- NVIDIA, AMD, etc. have introduced high performance GPUs that can be used for general purpose high performance computing: g*eneral purpose graphic processing units* (GPGPUs).
- GPGPUs are multicore, multithreaded processors which also include *SIMD capabilities*.

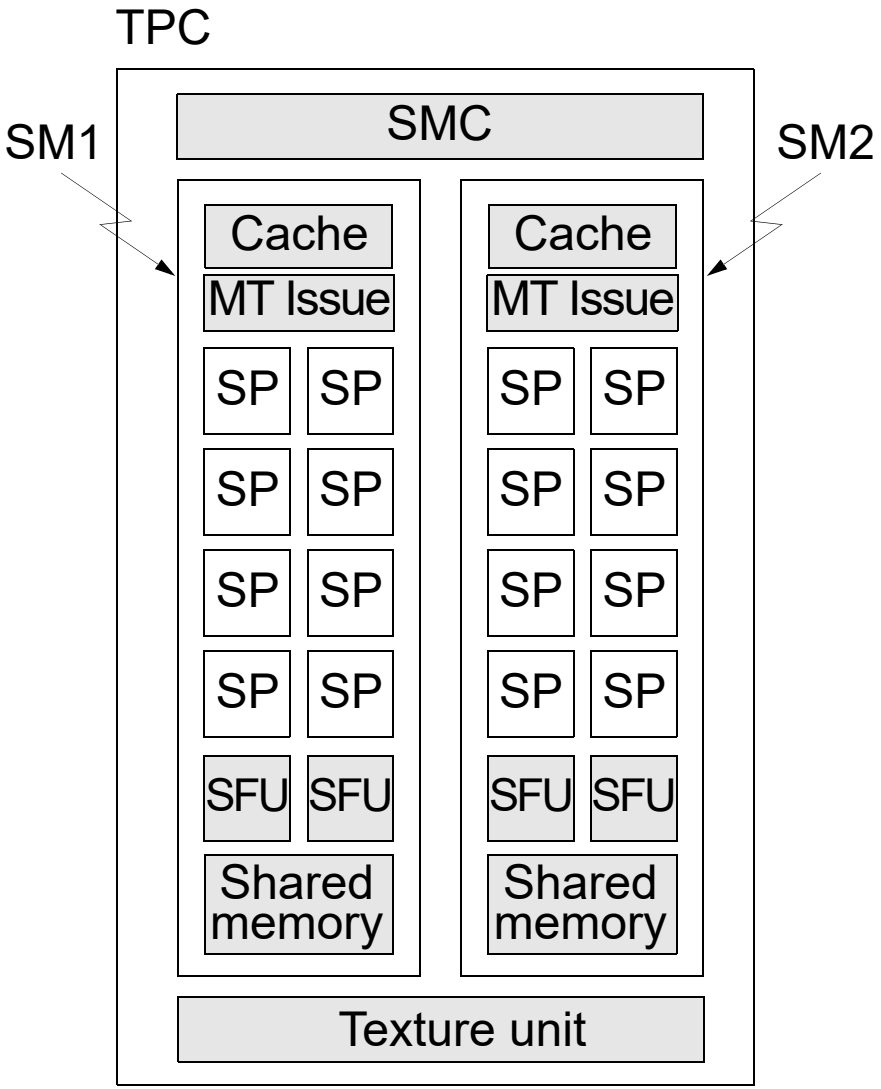
The NVIDIA Tesla GPGPU



- The NVIDIA Tesla (GeForce 8800) architecture:
  - 8 independent processing units (texture processor clusters - TPCs);
  - each TPC consists of 2 streaming multiprocessors (SMs);
  - each SM consists of 8 streaming processors (SPs).

The NVIDIA Tesla GPGPU





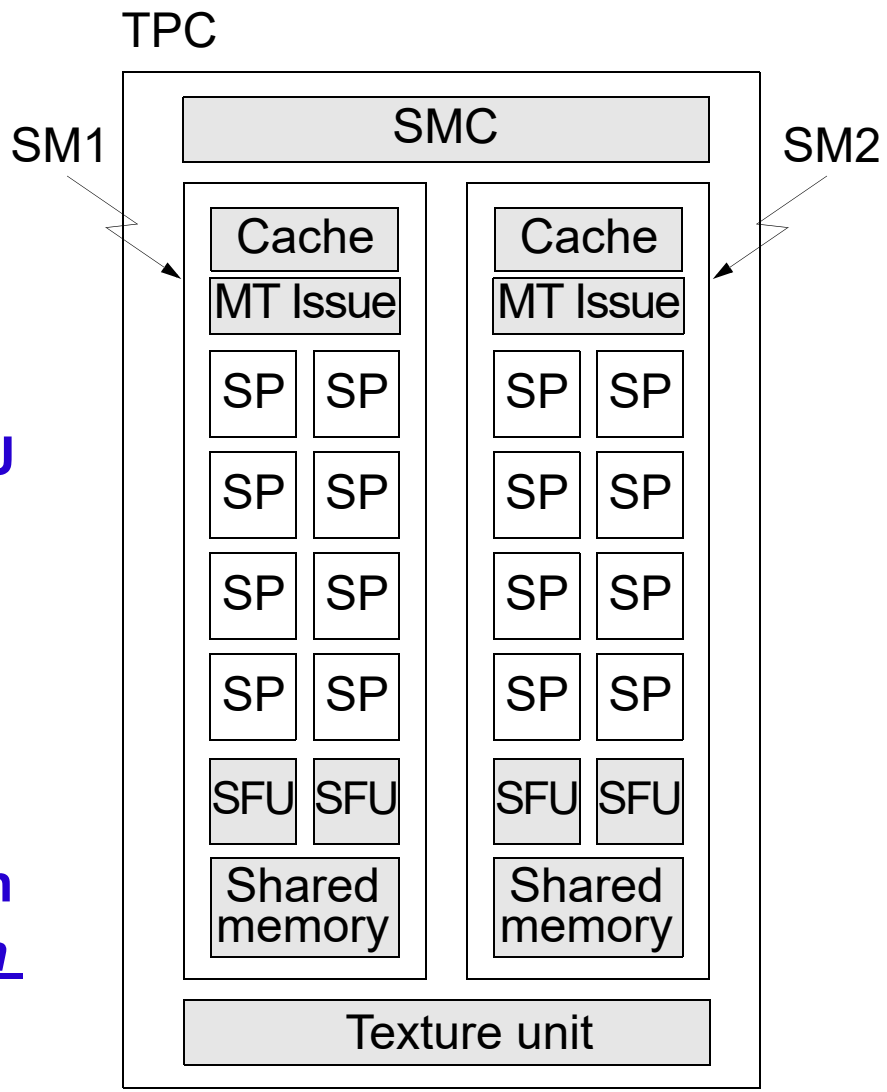
- **Each TPC consists of:**
  - **two SMs, controlled by the SM controller (SMC);**
  - **a texture unit (TU): is specialised for graphic processing; the TU can serve four threads per cycle.**

The NVIDIA Tesla GPGPU

- Each SM consists of:
  - 8 streaming processors (SPs);
  - multithreaded instruction fetch and issue unit (MT issue);
  - cache and shared memory;
  - two Special Function Units (SFUs); each SFU also includes 4 floating point multipliers.

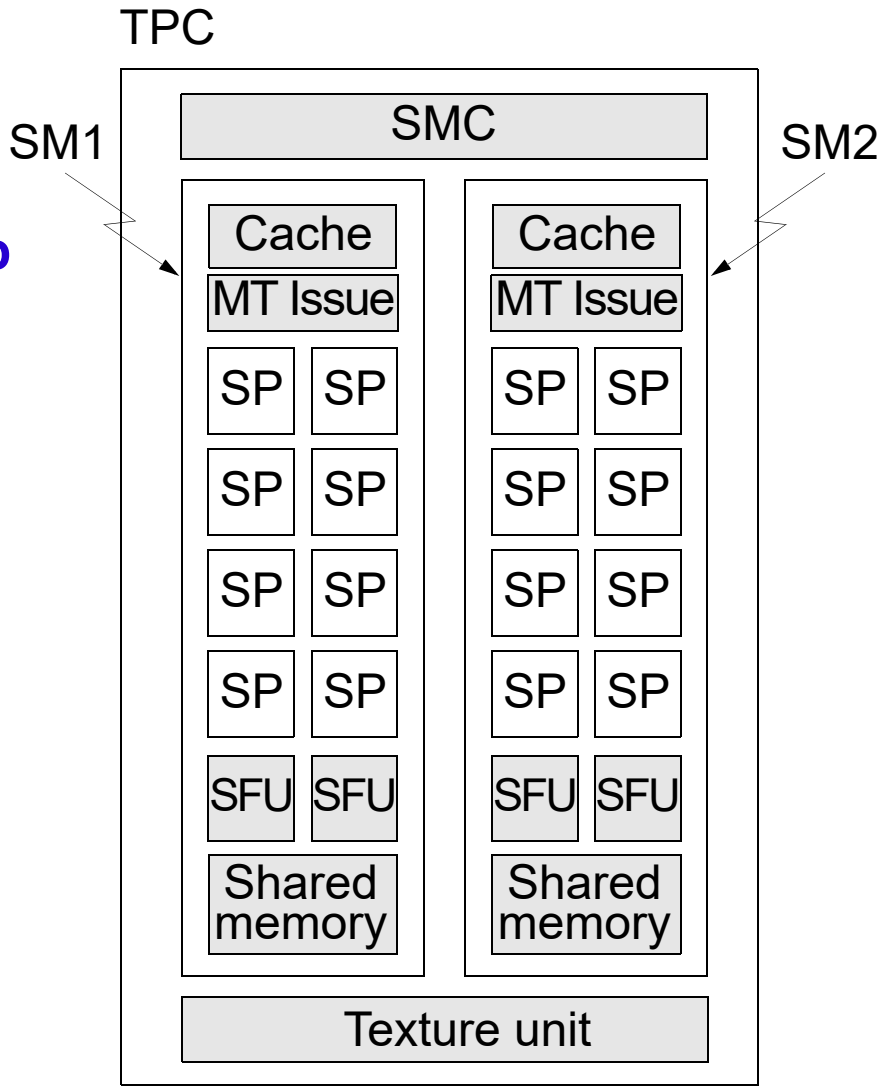
- Each SM is a multithreaded processor; it executes up to 768 concurrent threads.

- The SM architecture is based on a combination of SIMD and multithreading - ***single instruction multiple thread*** (SIMT):
  - a *warp* consist of 32 parallel threads;
  - each SM manages a group of 24 warps at a time (768 threads, in total);



The NVIDIA Tesla GPGPU

- At each instruction issue the SM selects a ready warp for execution.
- Individual threads belonging to the active warp are mapped for execution on the SP cores.
- The 32 threads of the same warp execute code starting from the same address.
- Once a warp has been selected for execution by the SM, all threads execute the same instruction at a time; some threads can stay idle (due to branching).
- The SP (streaming processor) is the primary processing unit of an SM. It performs:
  - floating point add, multiply, multiply-add;
  - integer arithmetic, comparison, conversion.



The NVIDIA Tesla GPGPU

- GPGPUS are optimised for throughput:
  - each thread may take longer time to execute but there are hundreds of threads active;
  - large amount of activity in parallel and large amount of data produced at the same time.

- Common CPU based parallel computers are primarily optimised for latency:
  - each thread runs as fast as possible, but only a limited amount of threads are active.

- Throughput oriented applications for GPGPUs:
  - extensive data parallelism: thousands of computations on independent data elements;
  - limited process-level parallelism: large groups of threads run the same program; not so many different groups that run different programs.
  - latency tolerant; the primary goal is the amount of work completed.

Vector Processors

- Vector processors include in their instruction set, beside scalar instructions, also instructions operating on vectors.

- Array processors (SIMD) computers can operate on vectors by executing simultaneously the same instruction on pairs of vector elements; *each instruction is executed by a separate processing element*.

- Several computer architectures have implemented vector operations using the parallelism provided by pipelined functional units. Such architectures are called *vector processors*.

Vector Processors

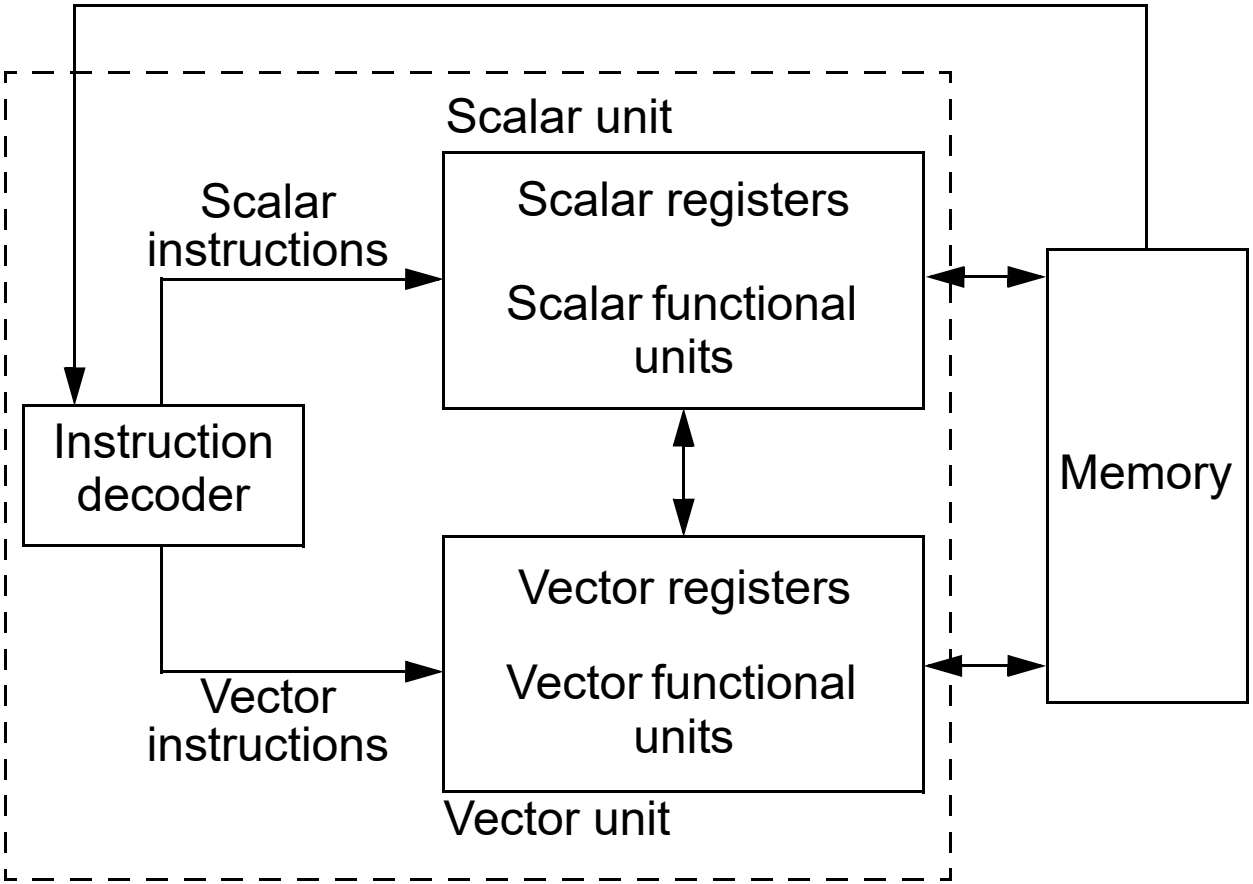
- Strictly speaking, vector processors are not parallel processors, although they behave like SIMD computers. There are not several CPUs in a vector processor, running in parallel. *They are SISD processors which have implemented vector instructions executed on pipelined functional units*.

- Vector computers usually have vector registers which can store each 64 up to 128 words.

- Vector instructions:
  - load vector from memory into vector register
  - store vector into memory
  - arithmetic and logic operations between vectors
  - operations between vectors and scalars
  - etc.

- The programmer is allowed to use operations on vectors; the compiler translates these instructions into vector instructions at machine level.

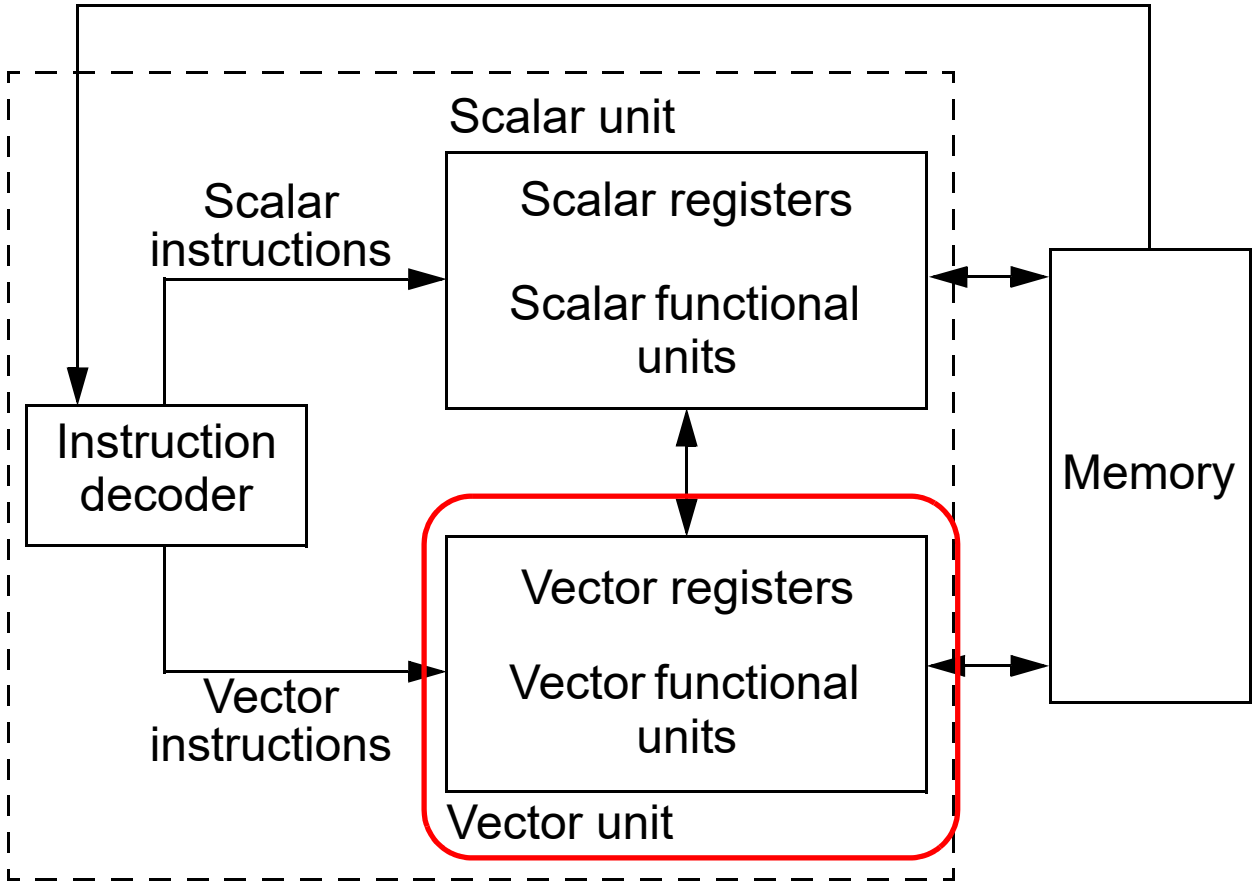
Vector Processors



Vector computers:

- CDC Cyber 205
- CRAY
- IBM 3090
- NEC SX
- Fujitsu VP
- HITACHI S8000

Vector Unit



- A vector unit consists of:
  - pipelined functional units
  - vector registers
- Vector registers:
  - *n general purpose vector registers* R_i, $0 \le i \le n-1$, each of length *s*;
  - *vector length register VL*: stores the length *l* ($0 \le l \le s$) of the currently processed vectors;
  - *mask register M*; stores a set of *l* bits, interpreted as boolean values; vector instructions can be executed in masked mode: vector register elements corresponding to a false value in *M*, are ignored.

Vector Instructions

LOAD-STORE instructions:

R ← A(x1:x2:incr)	load
A(x1:x2:incr) ← R	store
R ← MASKED(A)	masked load
A ← MASKED(R)	masked store
R ← INDIRECT(A(X))	indirect load
A(X) ← INDIRECT(R)	indirect store

Arithmetic - logic

R ← R' b_op R''
R ← S b_op R'
R ← u_op R'
M ← R rel_op R'
WHERE(M) R ← R' b_op R''

Vector Instructions

LOAD-STORE instructions:

R ← A(x1:x2:incr)	load
A(x1:x2:incr) ← R	store
R ← MASKED(A)	masked load
A ← MASKED(R)	masked store
R ← INDIRECT(A(X))	indirect load
A(X) ← INDIRECT(R)	indirect store

Arithmetic - logic

R ← R' b_op R''
R ← S b_op R'
R ← u_op R'
M ← R rel_op R'
WHERE(M) R ← R' b_op R''

Chaining

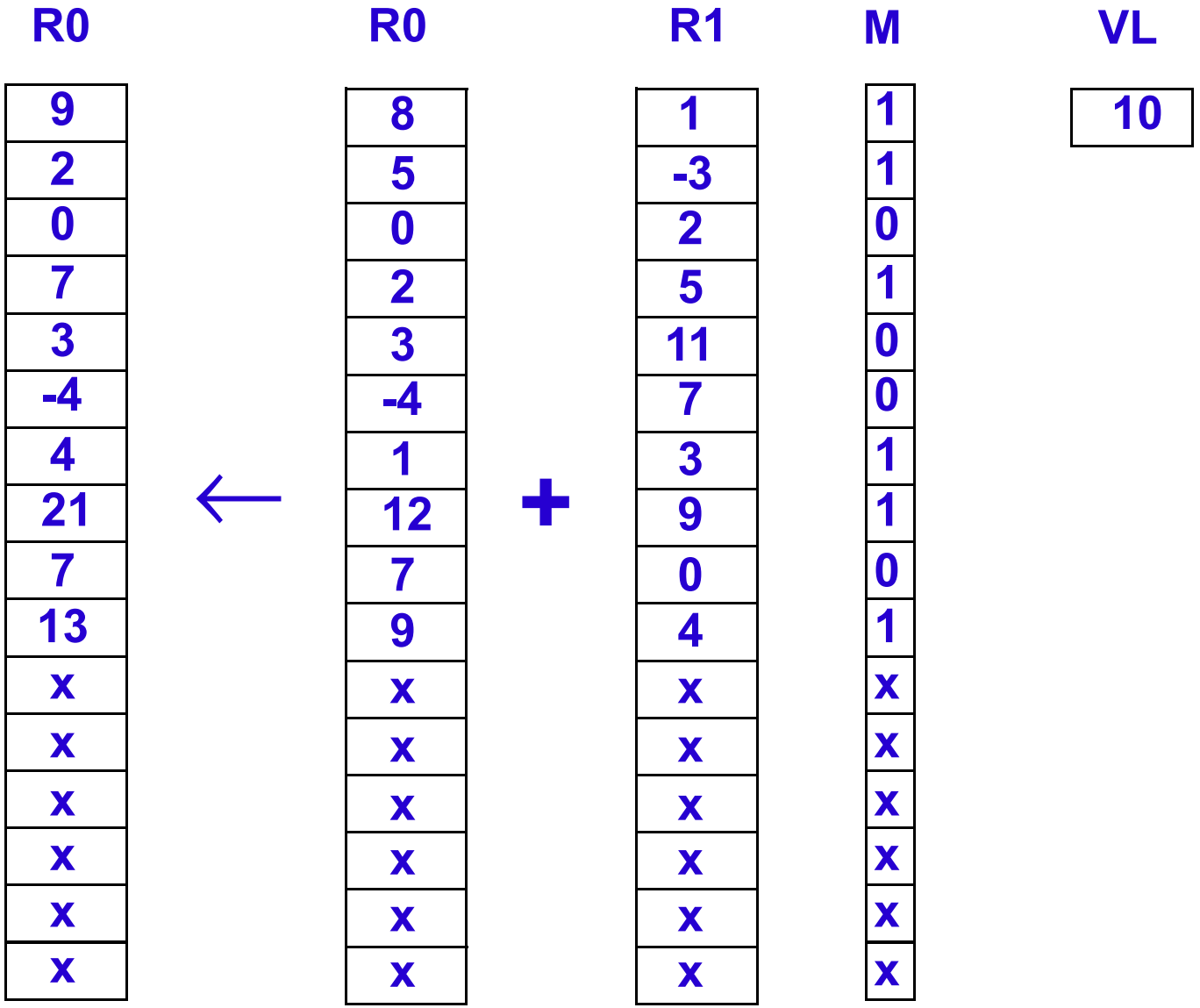
R2 ← R0 + R1
R3 ← R2 * R4

Execution of the vector multiplication has not to wait until the vector addition has terminated; as elements of the sum are generated by the addition pipeline they enter the multiplication pipeline;

addition and multiplication are performed (partially) in parallel.

Vector Instructions

Example: WHERE(M) R0 ← R0 + R1



Vector Instructions

In a language with vector computation instructions:

```
if T[1..50] > 0 then
    T[1..50] := T[1..50] + 1;
```

A compiler for a vector computer generates something like:

```
R0 ← T(0:49:1)
VL ← 50
M ← R0 > 0
WHERE(M) R0 ← R0 + 1
T(0:49:1) ← R0
```

Multimedia Extensions to General Purpose Microprocessors

- Video and audio applications very often deal with large arrays of small data types (8 or 16 bits).

- Such applications exhibit a large potential of SIMD (vector) parallelism.

General purpose microprocessors have been equipped with special instructions to exploit this potential of parallelism.

- The specialised multimedia instructions perform vector computations on bytes, half-words, or words.

Multimedia Extensions to General Purpose Microprocessors

Several vendors have extended the instruction set of their processors in order to improve performance with multimedia applications:

- MMX for Intel x86 family
- VIS for UltraSparc
- MDMX for MIPS
- MAX-2 for Hewlett-Packard PA-RISC
- NEON for ARM Cortex-A8, ARM Cortex-A9

The Pentium family provides 57 MMX instructions. They treat data in a SIMD fashion.

Multimedia Extensions to General Purpose Microprocessors

The basic idea: *subword execution*

- **Use the entire width of the data path (32 or 64 bits) when processing small data types used in signal processing (8, 12, or 16 bits).**

With word size 64 bits, the adders will be used to implement eight 8 bit additions in parallel.

This is practically a kind of SIMD parallelism, at a reduced scale.

Multimedia Extensions to General Purpose Microprocessors

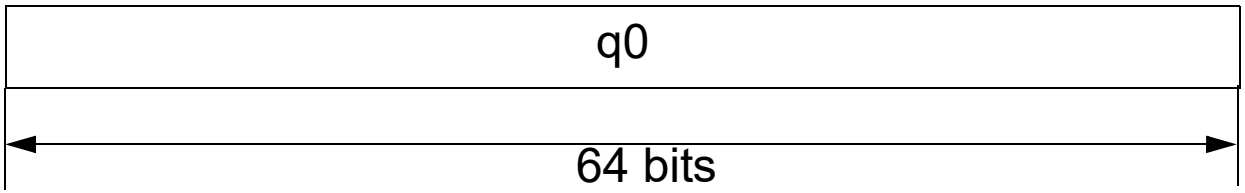
Three packed data types are defined for parallel operations: *packed byte*, *packed word*, *packed double word*.

Packed byte

Packed word

Packed double word

Quadword



Multimedia Extensions to General Purpose Microprocessors

Examples of SIMD arithmetics with the MMX instruction set:

ADD R3 ← R1,R2

a7	a6	a5	a4	a3	a2	a1	a0
+	+	+	+	+	+	+	+
b7	b6	b5	b4	b3	b2	b1	b0
=	=	=	=	=	=	=	=
a7+b7	a6+b6	a5+b5	a4+b4	a3+b3	a2+b2	a1+b1	a0+b0

Multimedia Extensions to General Purpose Microprocessors

Examples of SIMD arithmetics with the MMX instruction set:

ADD R3 ← R1,R2

a7	a6	a5	a4	a3	a2	a1	a0
+	+	+	+	+	+	+	+
b7	b6	b5	b4	b3	b2	b1	b0
=	=	=	=	=	=	=	=
a7+b7	a6+b6	a5+b5	a4+b4	a3+b3	a2+b2	a1+b1	a0+b0

MPYADD R3 ← R1,R2

a7	a6	a5	a4	a3	a2	a1	a0
x-+		x-+		x-+		x-+	
b7	b6	b5	b4	b3	b2	b1	b0
=		=		=		=	
(a6xb6)+(a7xb7)		(a4xb4)+(a5xb5)		(a2xb2)+(a3xb3)		(a0xb0)+(a1xb1)	

Multimedia Extensions to General Purpose Microprocessors

How to get the data ready for computation?

How to get the results back in the right format?

- **Packing and Unpacking:**