Multithreading and Multiprocessors

Politecnico di Milano

Christian Pilato <christian.pilato@polimi.it>

Outline

- Multithreading
- Multiprocessors
  - Flynn taxonomy
  - SIMD architectures
  - Vector architectures
  - MIMD architectures

Decreasing returns on investment

- ’80s: expansion of superscalar processors
  - Improvement of 50% in performance
  - Transistors used to create implicit parallelism
    - Pipelined Processors (10 CPI to 1 CPI)

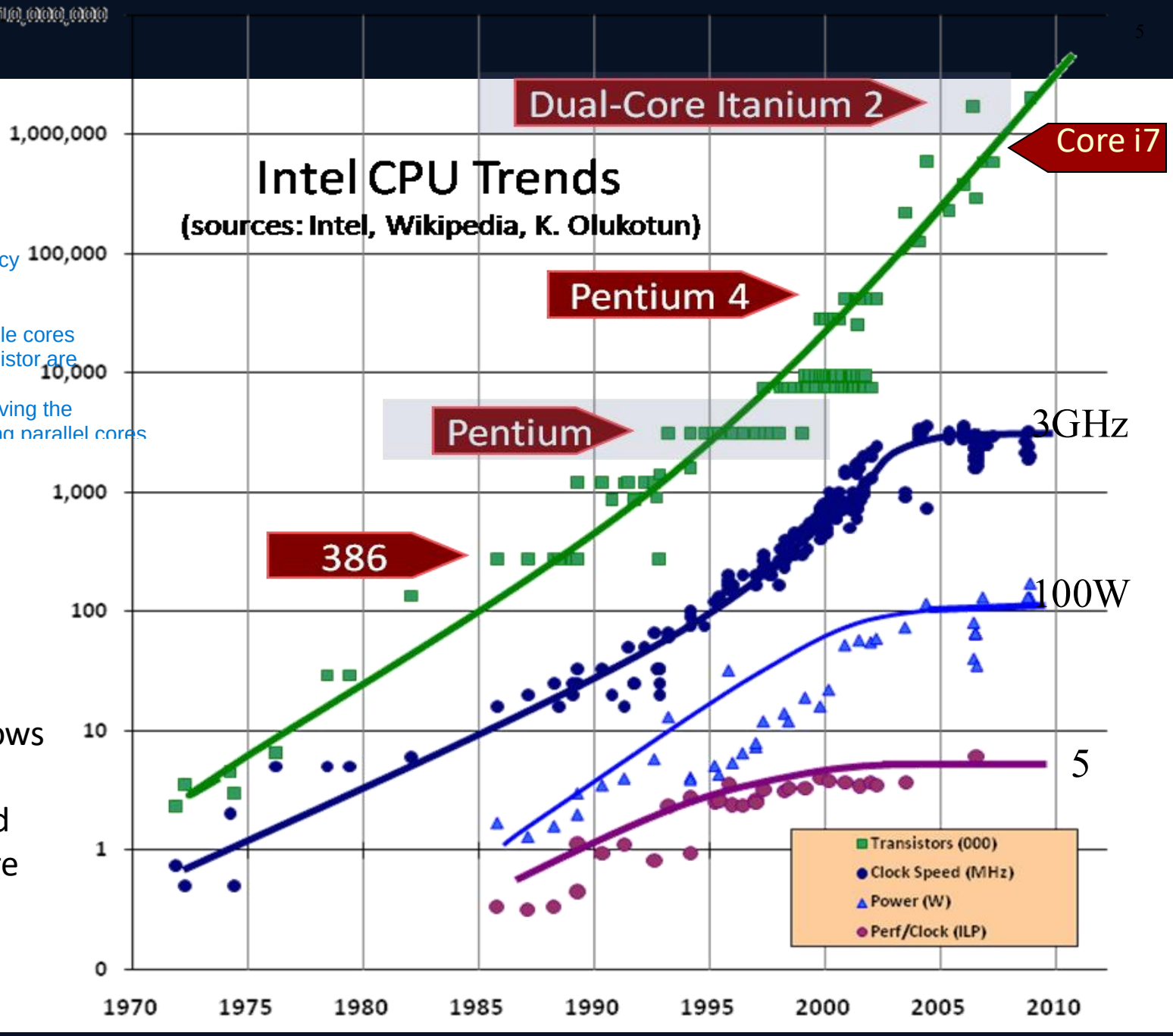
Decreasing returns on investment

- ’80s: expansion of superscalar processors
  - Improvement of 50% in performance
  - Transistors used to create implicit parallelism
    - Pipelined Processors (10 CPI to 1 CPI)
- ’90s: era of decreasing returns
  - Exploit at best the implicit parallelism
    - Issue from 2 to 6 ways, issue out-of-order, branch prediction (from 1 CPI to 0.5 CPI)
    - Performance below expectations
    - Delayed and cancelled projects

Next?

Further increase of frequency causes too much heating.

Even if performance of single cores reached their plateau, transistor are increasing, why?
Because we stopped improving the single core and started doing parallel cores



Trends:

- #transistors follows Moore
- but not freq. and performance/core

Decreasing returns on investment

- ’80s: expansion of superscalar processors
  - Improvement of 50% in performance
  - Transistors used to create implicit parallelism
    - Pipelined Processors (10 CPI to 1 CPI)
- ’90s: era of decreasing returns
  - Exploit at best the implicit parallelism
    - Issue from 2 to 6 ways, issue out-of-order, branch prediction (from 1 CPI to 0.5 CPI)
    - Performance below expectations
    - Delayed and cancelled projects
- 2000: beginning of the multicore era
  - Explicit Parallelism

Motivations for paradigm change

- Modern processors fail to utilize execution resources well
- There is no single culprit:
  - Memory conflicts, control hazards, branch misprediction, cache miss….
- Attacking the problems one at a time always has limited effectiveness
- Need for a general latency-tolerance solution which can hide all sources of latency can have a large impact on performance

Parallel programming

- Explicit parallelism implies structuring the applications into concurrent and communicating tasks
- Operating systems offer support for different types of

Parallel programming

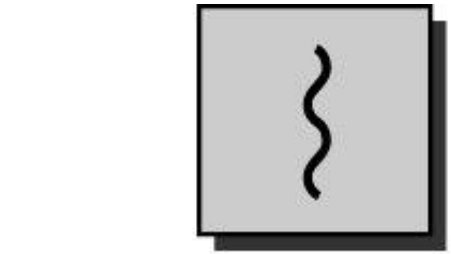
- Explicit parallelism implies structuring the applications into concurrent and communicating tasks
- Operating systems offer support for different types of tasks. The most important and frequent are:
  - processes
  - threads
- The operating systems implement multitasking

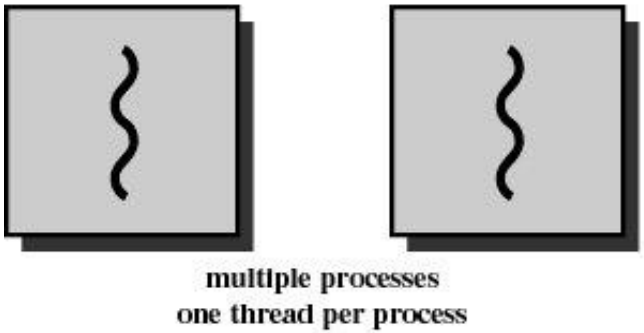
Process/Thread

one process
one thread

 = instruction trace

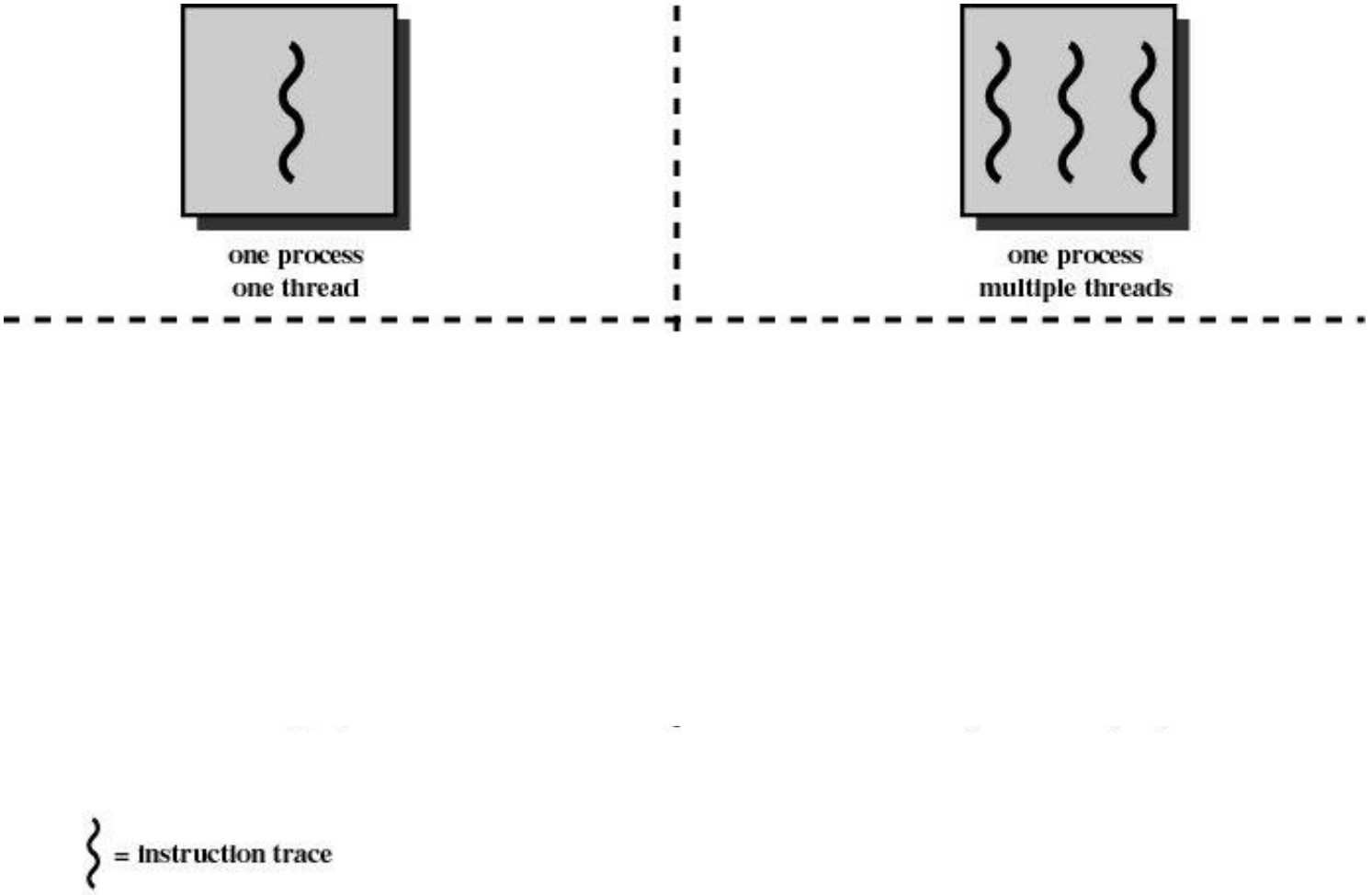
Multiplicity of processes



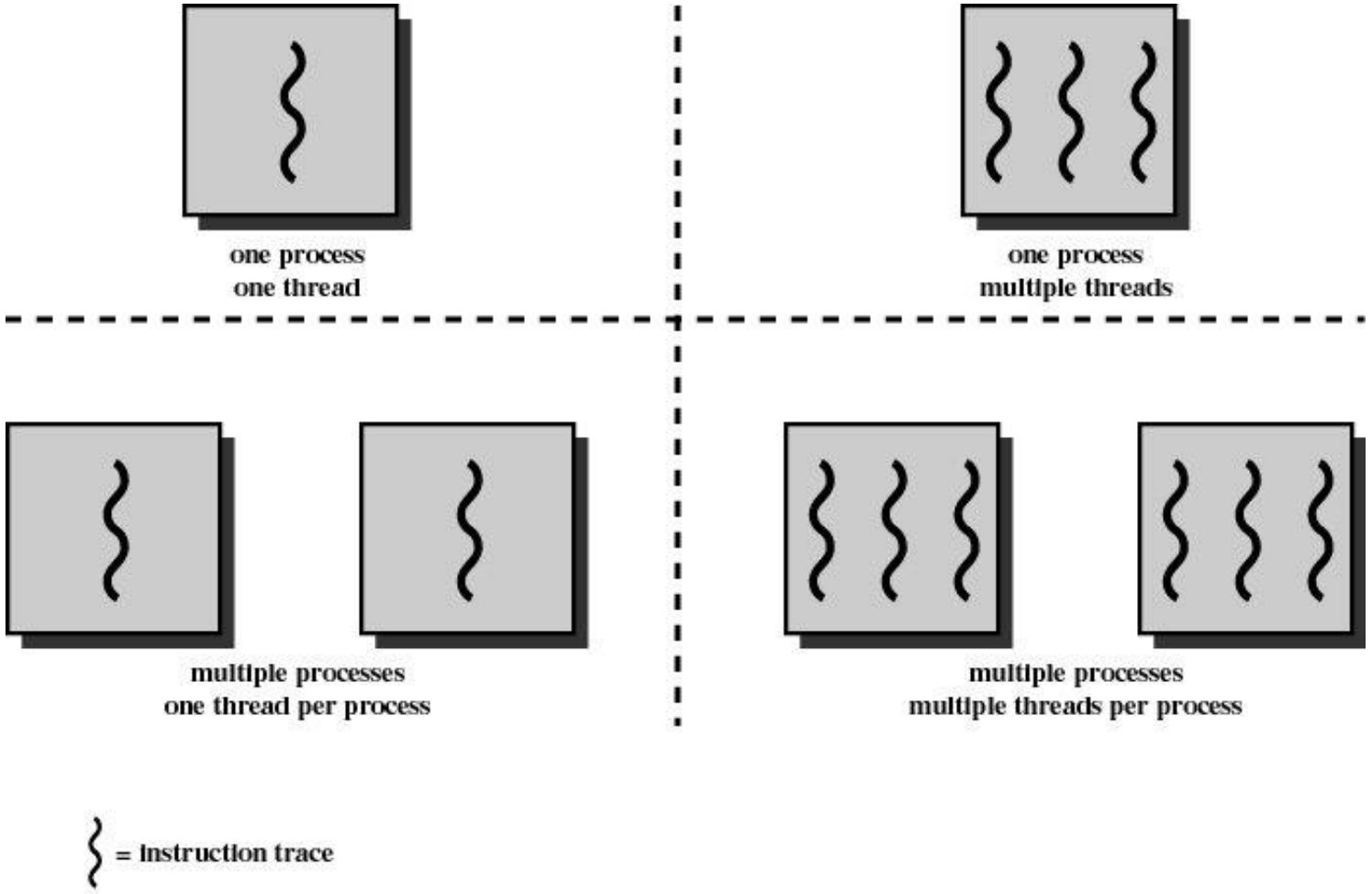


Process: execution entity
Thread: hardware entity

Multiplicity of threads



Multiplicity of processes/threads



Parallel programming

- Explicit parallelism implies structuring the applications into concurrent and communicating tasks
- Operating systems offer support for different types of tasks. The most important and frequent are:
  - processes
  - threads
- The operating systems implement multitasking differently based on the characteristics of the processor:
  - single core
  - single core with multithreading support
  - multicore

Parallel programming

- Explicit parallelism implies structuring the applications into concurrent and communicating tasks
- Operating systems offer support for different types of tasks. The most important and frequent are:
  - processes
  - threads
- The operating systems implement multitasking differently based on the characteristics of the processor:
  - single core
  - single core with multithreading support
  - multicore

??? 2nd part of today slides ;)

Multithreaded Execution

- Multithreading: multiple threads to share the functional units of 1 processor via overlapping
  - processor must duplicate independent state of each thread e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
  - memory shared through the virtual memory mechanisms, which already support multiple processes
  - HW for fast thread switch; much faster than full process switch ≈ 100s to 1000s of clocks

Multithreaded Execution

- Multithreading: multiple threads to share the functional units of 1 processor via overlapping
  - processor must duplicate independent state of each thread e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
  - memory shared through the virtual memory mechanisms, which already support multiple processes
  - HW for fast thread switch; much faster than full process switch ≈ 100s to 1000s of clocks
- When switch?

When an instruction takes long time to execute, it is possible to do a switch and execute another piece of code (thread) in the meanwhile

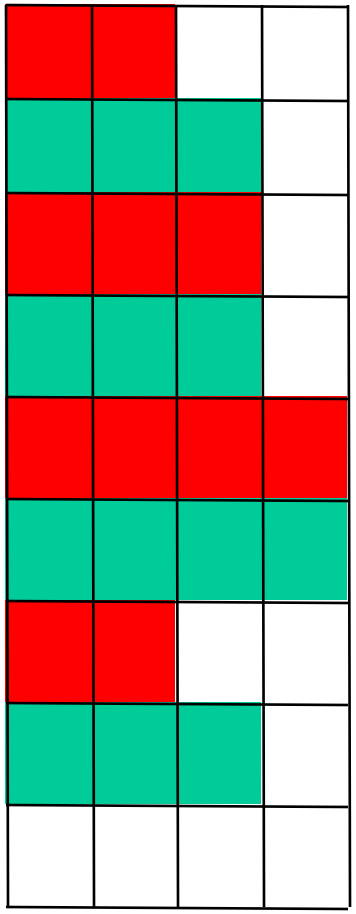
  - Alternate instruction per thread (fine grain)
  - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)

Thread-level parallelism (TLP)

- Fine grained multithreading (try to switch at instruction level)
  - Switches from one thread to the other at each instruction – the execution of more threads is interleaved (often the switching is performed taking turns, skipping one thread if there is a stall)
  - The CPU must be able to change thread at every clock cycle. It is necessary to duplicated the hardware resources.

Example of fine grained MT in superscalar processor

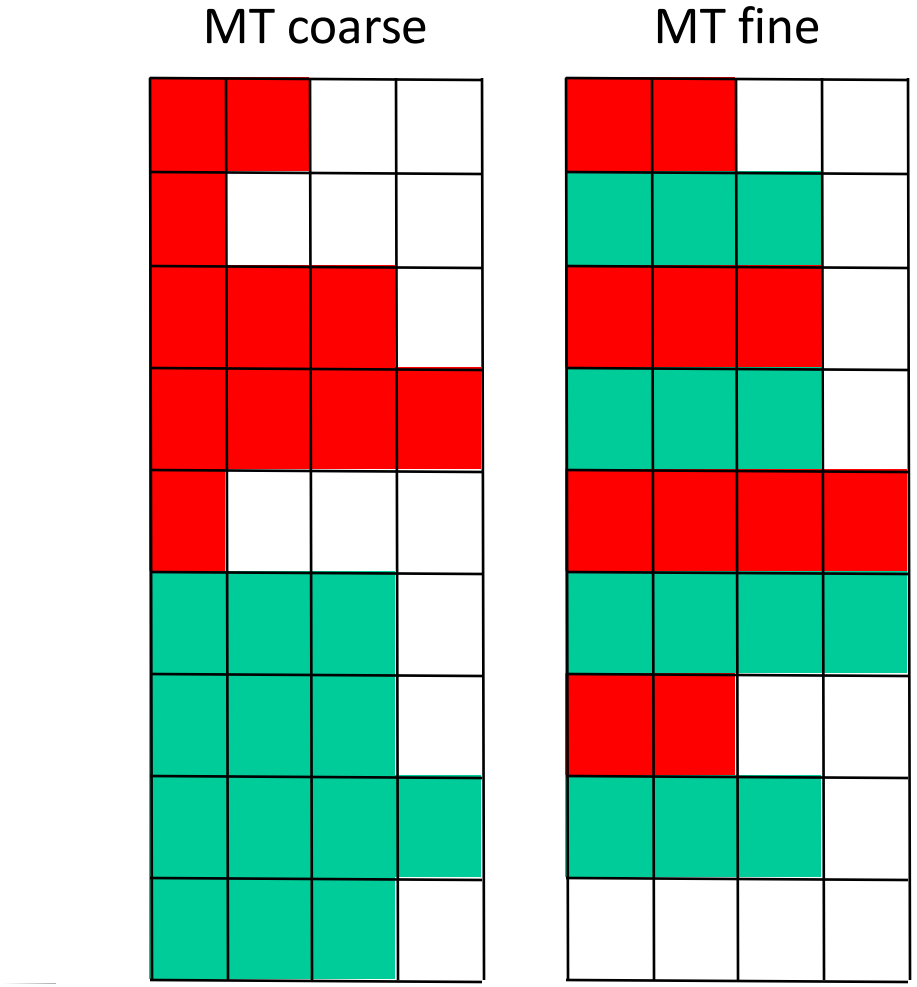
MT fine



Thread-level parallelism (TLP)

- Fine grained multithreading
  - Switches from one thread to the other at each instruction – the execution of more threads is interleaved (often the switching is performed taking turns, skipping one thread if there is a stall)
  - The CPU must be able to change thread at every clock cycle. It is necessary to duplicate the hardware resources.
- Coarse grained multithreading
  - switching from one thread to another occurs only when there are long stalls – e.g., for a miss on the second level cache.
  - Two threads share many system resources (e.g., architectural registers), the switching from one thread to the next requires different clock cycles to save the context.

Thread-level parallelism: Comparison



Coarse grained multithreading

- Advantage: in normal conditions the single thread is not slowed down
  - Relieves need to have very fast thread-switching
  - Doesn't slow down thread, since instructions from other threads issued only when the thread encounters a costly stall

Coarse grained multithreading

- Advantage: in normal conditions the single thread is not slowed down
  - Relieves need to have very fast thread-switching
  - Doesn't slow down thread, since instructions from other threads issued only when the thread encounters a costly stall

- Disadvantage: for short stalls it does not reduce the throughput loss – the CPU starts the execution of instructions that belonged to a single thread, when there is one stall it is necessary to empty the pipeline before starting the new thread

Do both ILP and TLP?

(thread level parallelism)

- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP to exploit TLP? Yes
  - functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
- Could the TLP be used as a source of independent instructions that might keep the processor busy during stalls?
- Could TLP be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?

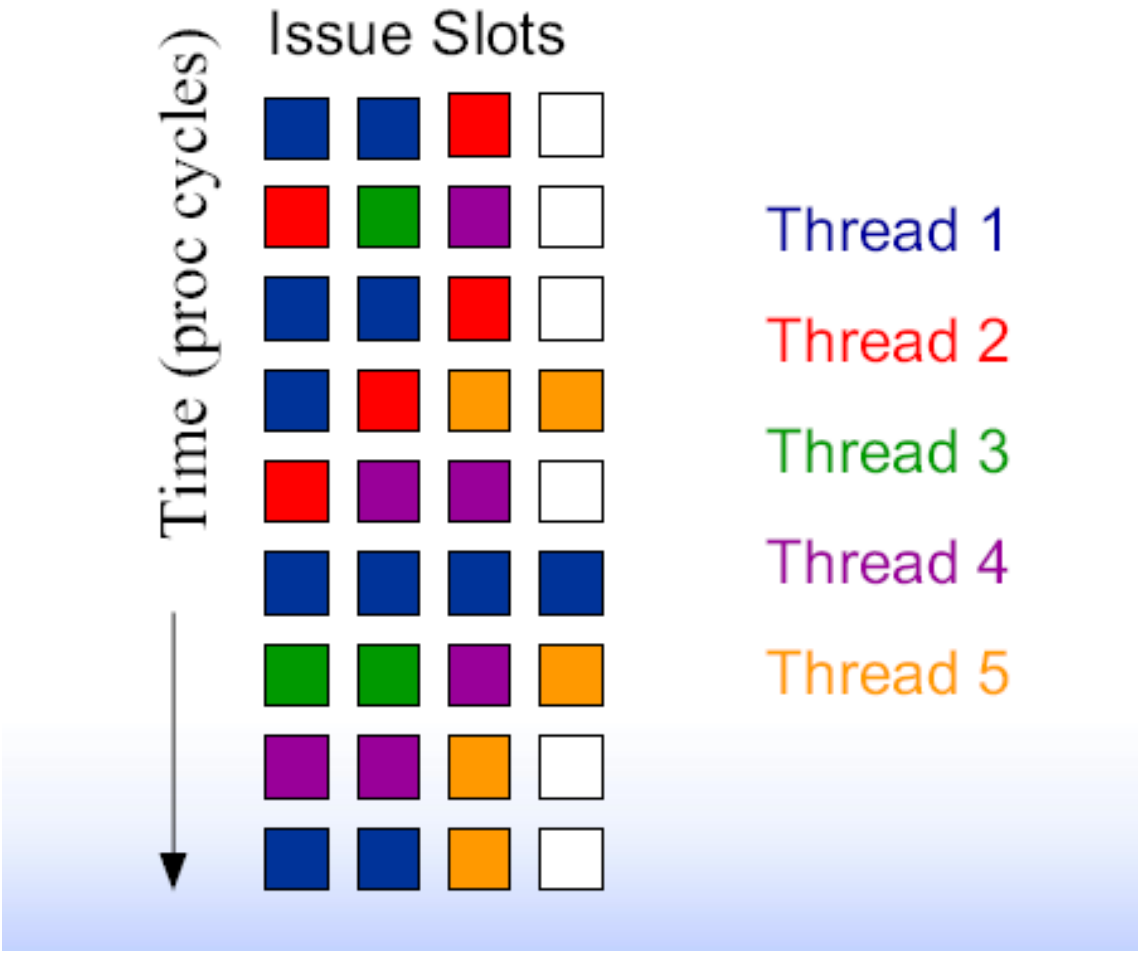
Thread Level Parallelism Simultaneous Multithreading

- Uses the resources of one superscalar processor to exploit simultaneously ILP and TLP.
- Key motivation: a CPU today has more functional resources than what one thread can in fact use
- Thanks to register renaming and dynamic scheduling, more independent instructions to different threads may be issued without worrying about dependences (that are solved by the hardware of the dynamic scheduling).
- Simultaneously schedule instructions for execution from all threads

Simultaneous Multithreading (SMT)

- Simultaneous multithreading (SMT): insight that dynamically scheduled processor already has many HW mechanisms to support multithreading
  - Large set of virtual registers that can be used to hold the register sets of independent threads
  - Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath without confusing sources and destinations across threads
  - Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW
- Just adding a per thread renaming table and keeping separate PCs
  - Independent commitment can be supported by logically keeping a separate reorder buffer for each thread

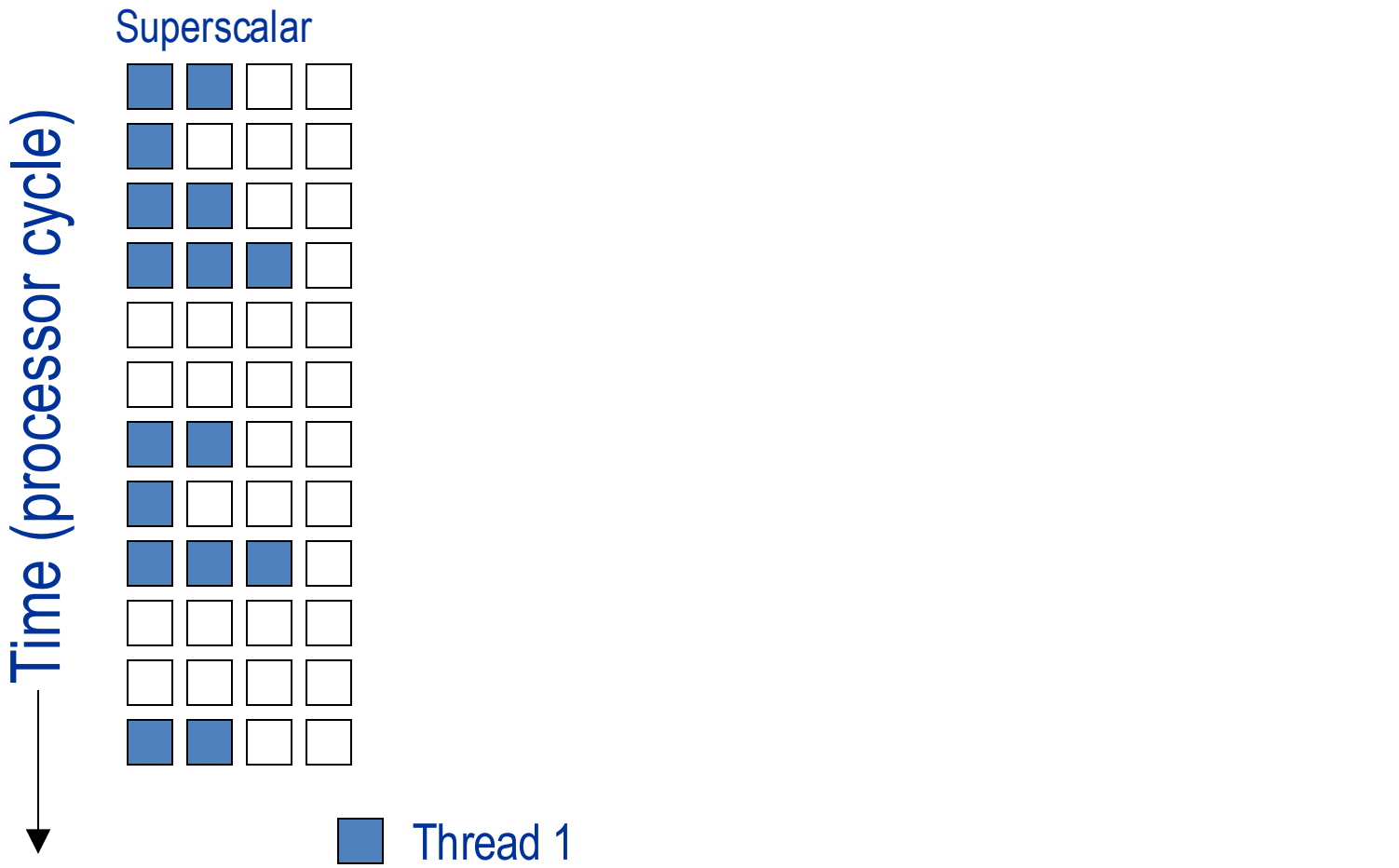
Simultaneous Multithreading (SMT)



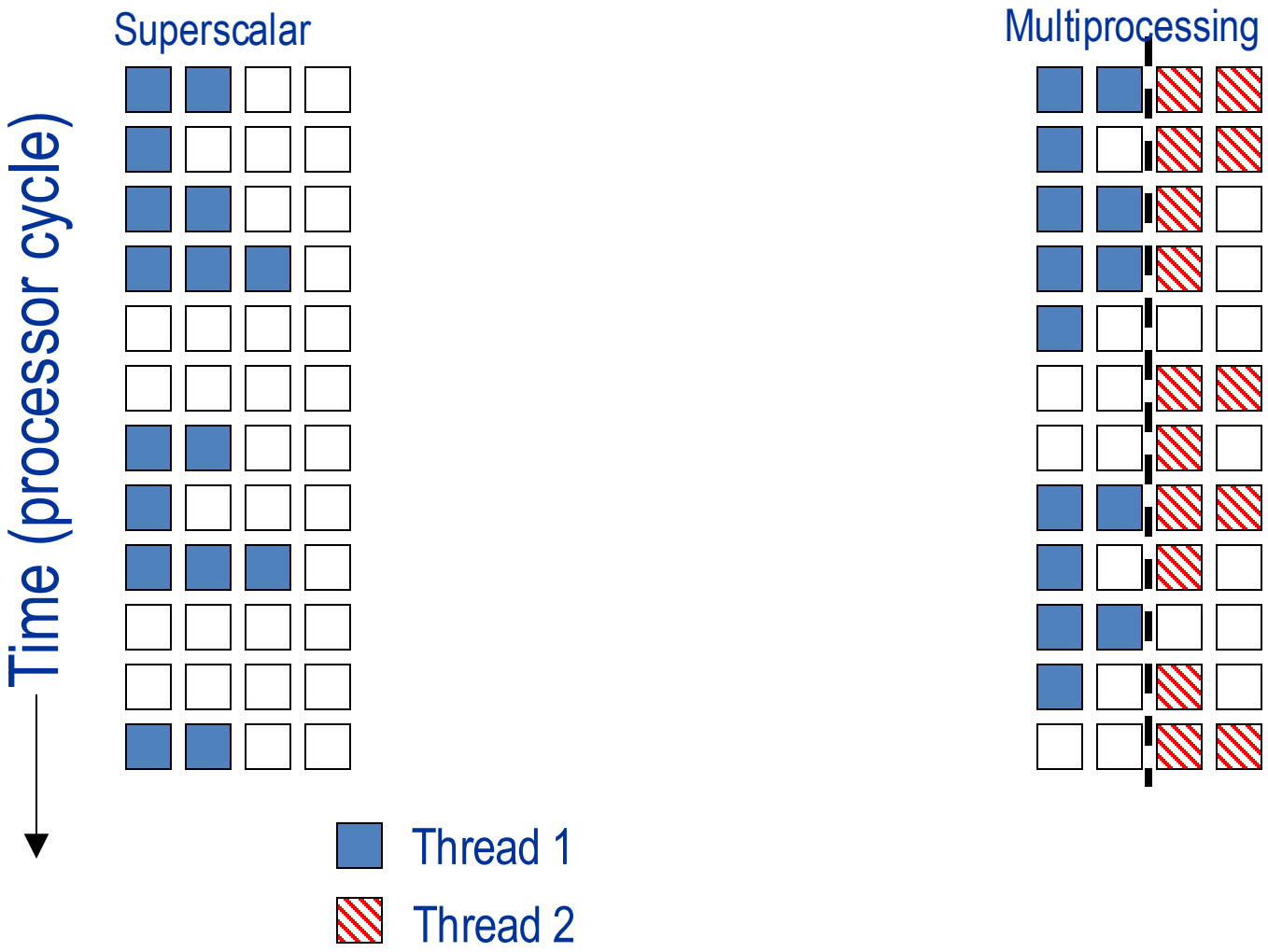
Simultaneous Multithreading (SMT)

- The system can be dynamically adapted to the environment, allowing (if possible) the execution of instructions from each thread, and allowing that the instructions of a single thread used all functional units if the other thread incurs in a long latency event.
- More threads use the issues possibilities of the CPU at each cycle; ideally, the exploitation of the issues availabilities is limited only by the unbalance between resources requests and availabilities.

Multithreaded Categories

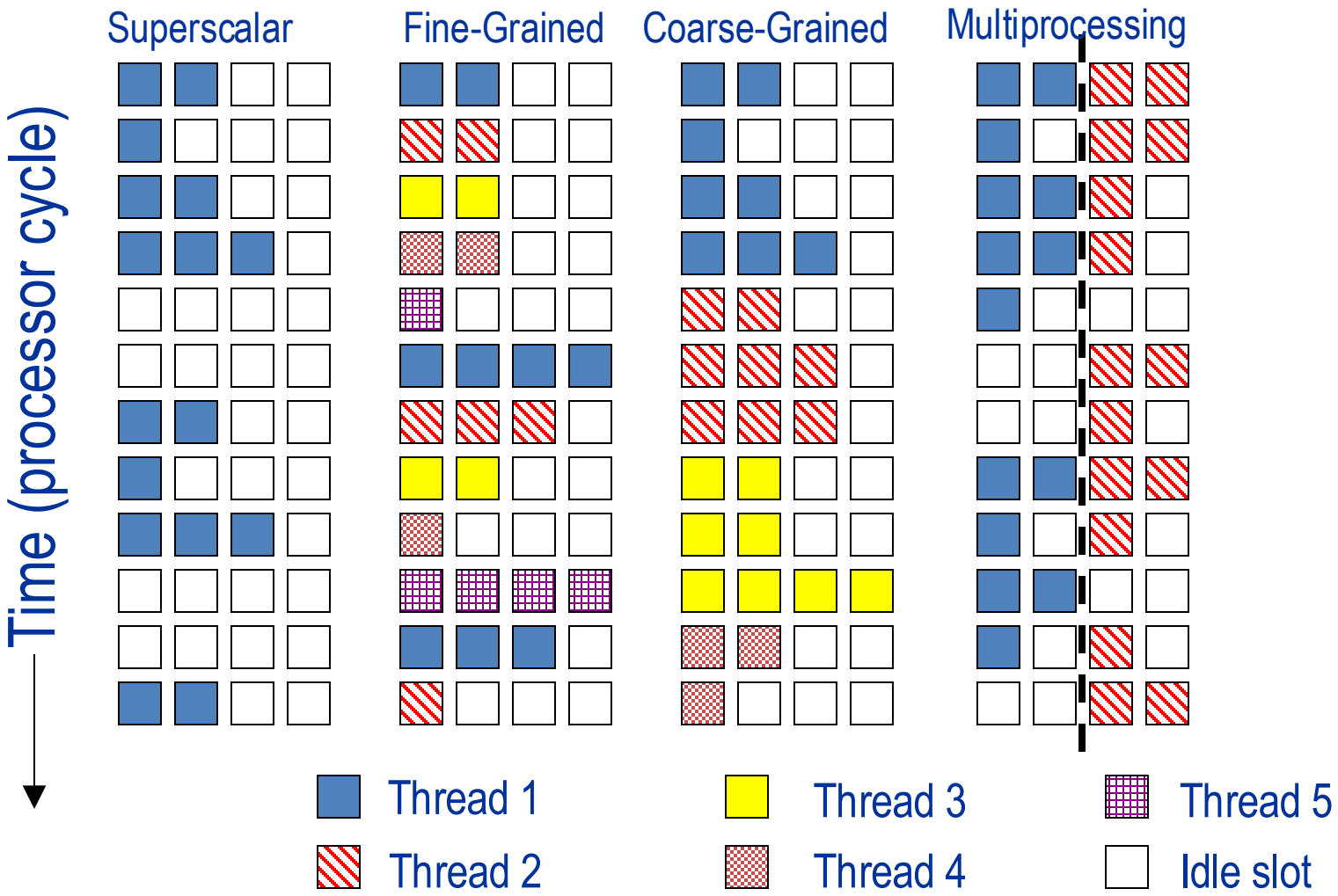


Multithreaded Categories

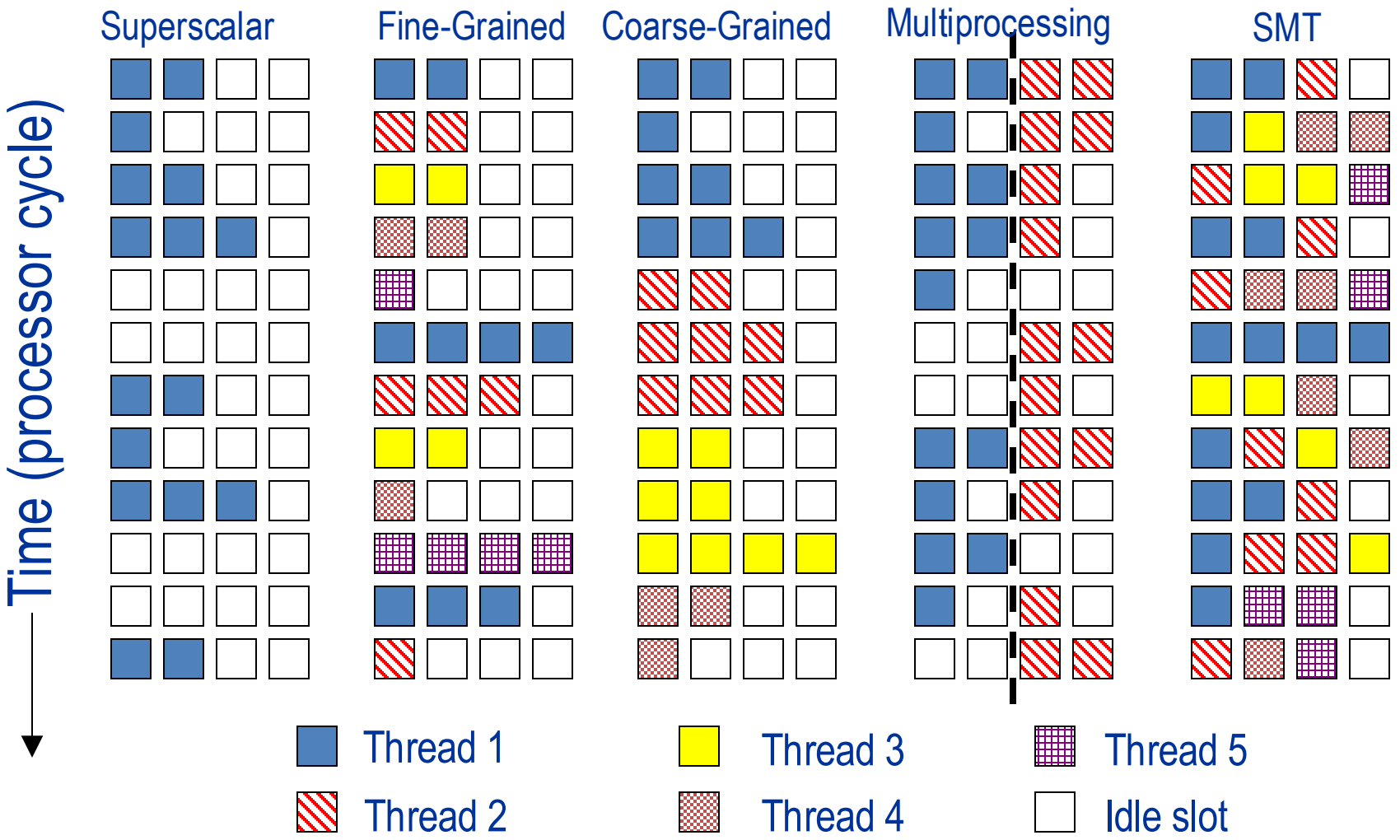


Multithreaded Categories

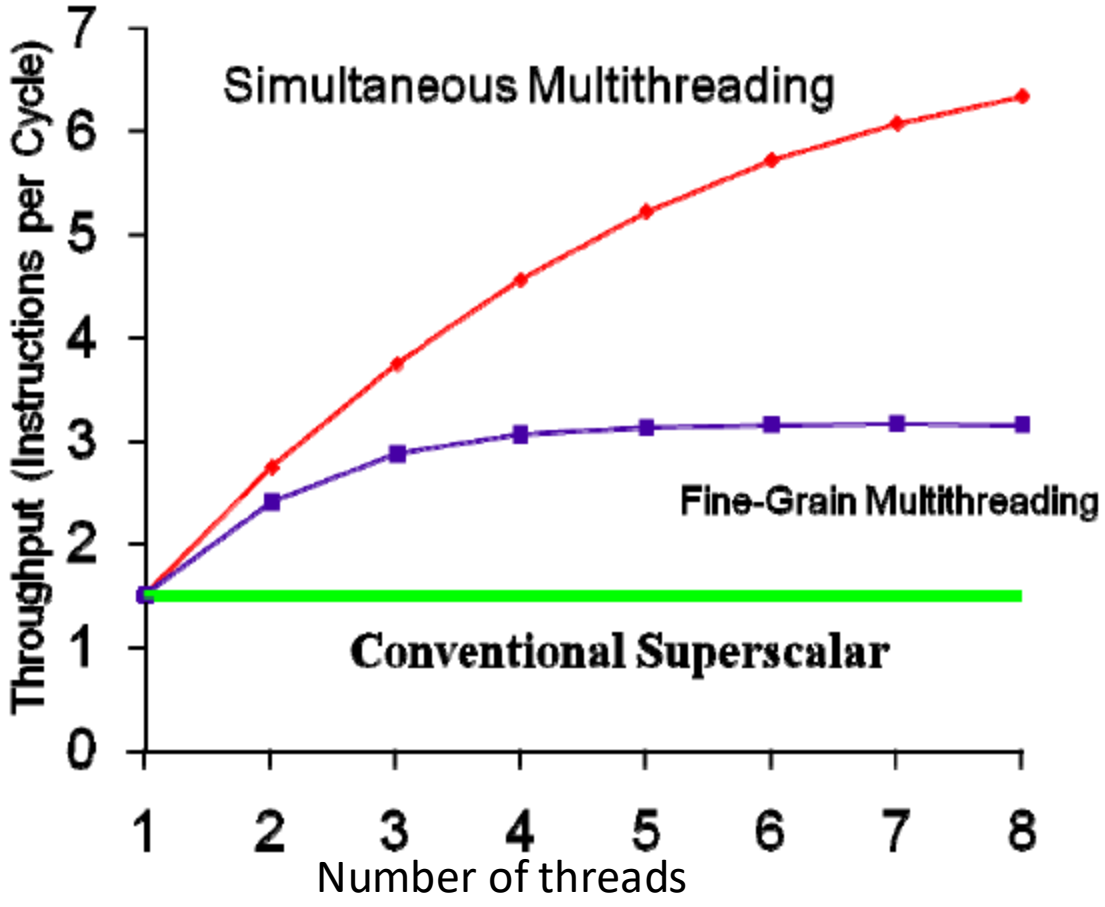
Multithreaded Categories



Multithreaded Categories



Performance comparison



What now?

What now?

- Difficult to increase performance and clock frequency of the single core
- Deep pipeline: if we increase depth of pipeline we increase troughput but introduce those problems:
  - Heat dissipation problems
  - Speed light transmission problems in wires
  - Difficulties in design and verification (more transistor->more gates->more potential issues)
  - Requirement of very large design groups

- Many new applications are multi-threaded

(mantaining simple cores)

Beyond ILP

- ILP architectures (superscalar, VLIW...):
  - ➢ Support fine-grained, instruction-level parallelism;
  - ➢ Fail to support large-scale parallel systems;
- Multiple-issue CPUs are very complex, and returns (as far as extracting greater parallelism) are diminishing ⇨ extracting parallelism at higher levels becomes more and more attractive.
- A further step: *process- and thread-level parallel*

Beyond ILP

- ILP architectures (superscalar, VLIW...):
  - Support fine-grained, instruction-level parallelism;
  - Fail to support large-scale parallel systems;
- Multiple-issue CPUs are very complex, and returns (as far as extracting greater parallelism) are diminishing ⇨ extracting parallelism at higher levels becomes more and more attractive.
- A further step: *process- and thread-level parallel architectures.*
- To achieve ever greater performance: *connect multiple microprocessors in a complex system*.

Parallel Architectures

- Definition: “A parallel computer is a collection of processing elements that cooperates and communicate to solve large problems fast”
    - Almasi and Gottlieb, Highly Parallel Computing, 1989

Parallel Architectures

- Definition: "A parallel computer is a collection of processing elements that cooperates and communicate to solve large problems fast"
  - Almasi and Gottlieb, Highly Parallel Computing, 1989
- The aim is to replicate processors to add performance vs design a faster processor.
- Parallel architecture extends traditional computer architecture with a communication architecture
  - abstractions (HW/SW interface)
  - different structures to realize abstraction efficiently

Flynn Taxonomy (1966)

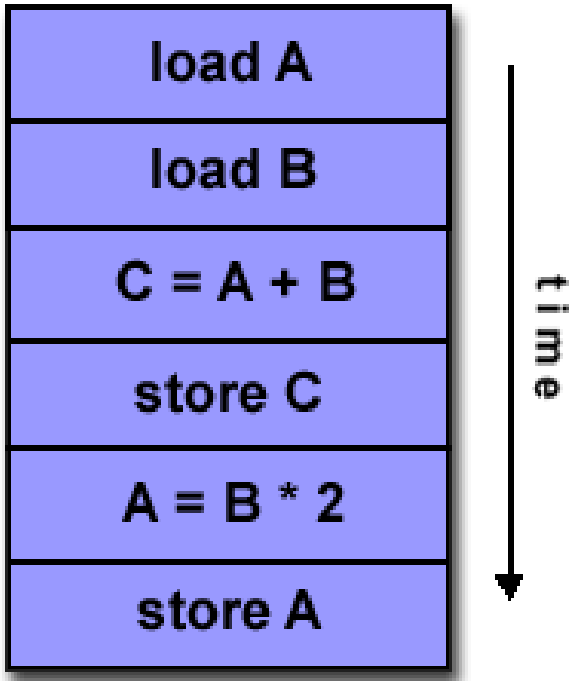
- SISD - Single Instruction Single Data
  - Uniprocessor systems
- MISD - Multiple Instruction Single Data
  - No practical configuration and no commercial systems
- SIMD - Single Instruction Multiple Data
  - Simple programming model, low overhead, flexibility, custom integrated circuits
- MIMD - Multiple Instruction Multiple Data
  - Scalable, fault tolerant, *off-the-shelf* micros

Flynn Taxonomy (1966)

- SISD - Single Instruction Single Data
  - Uniprocessor systems
- MISD - Multiple Instruction Single Data
  - No practical configuration and no commercial systems
- SIMD - Single Instruction Multiple Data
  - Simple programming model, low overhead, flexibility, custom integrated circuits
- MIMD - Multiple Instruction Multiple Data
  - Scalable, fault tolerant, *off-the-shelf* micros

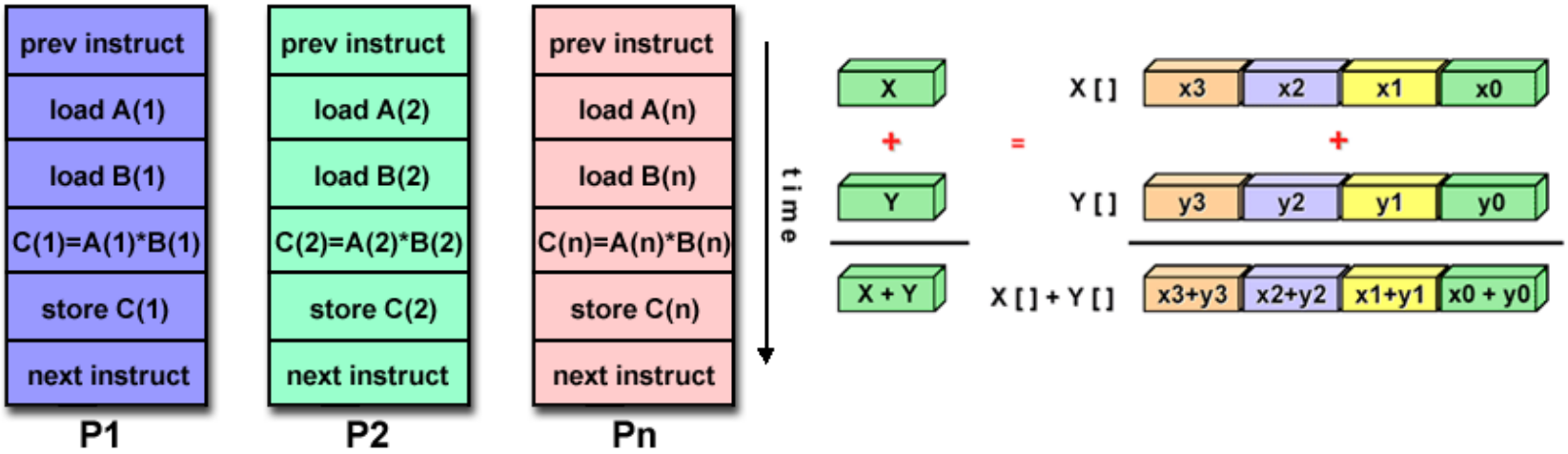
SISD

- A serial (non-parallel) computer
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and even today, the most common type of computer



SIMD

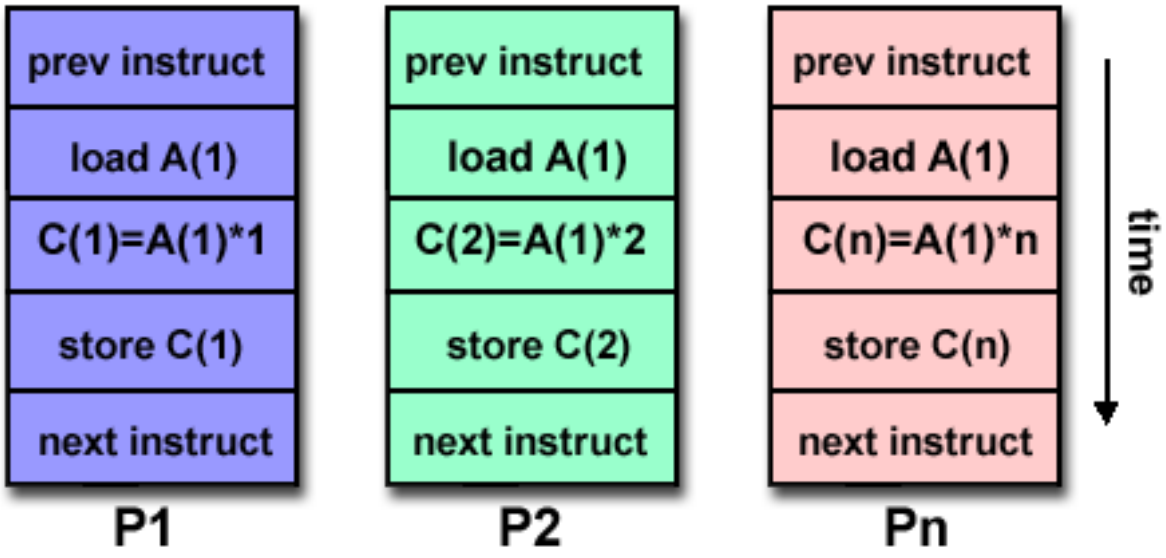
- A type of parallel computer
- Single instruction: all processing units execute the same instruction at any given clock cycle
- Multiple data: each processing unit can operate on a different data element



- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing

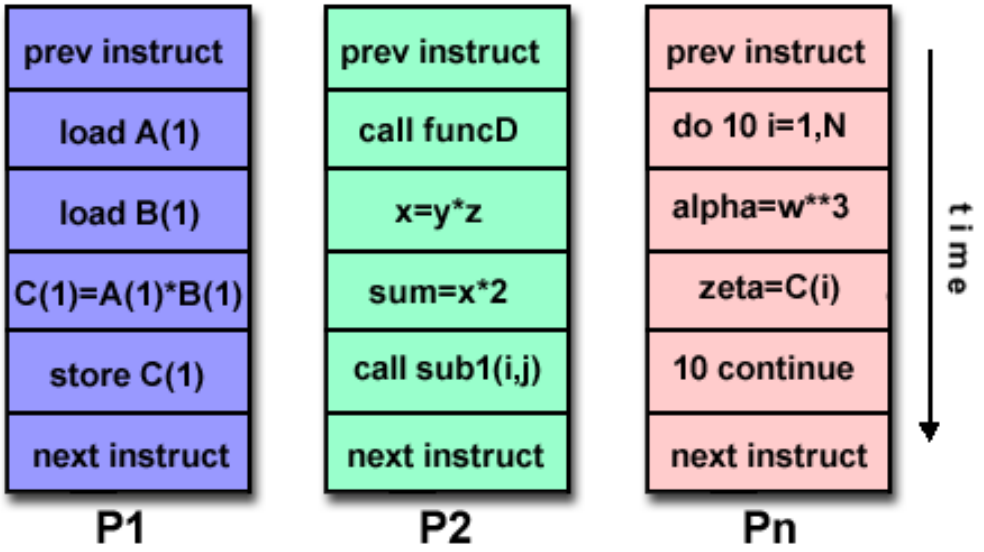
MISD

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.



MIMD

- Nowadays, the most common type of parallel computer
- Multiple Instruction: every processor may be executing a different instruction stream
- Multiple Data: every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic



Which kind of multiprocessors?

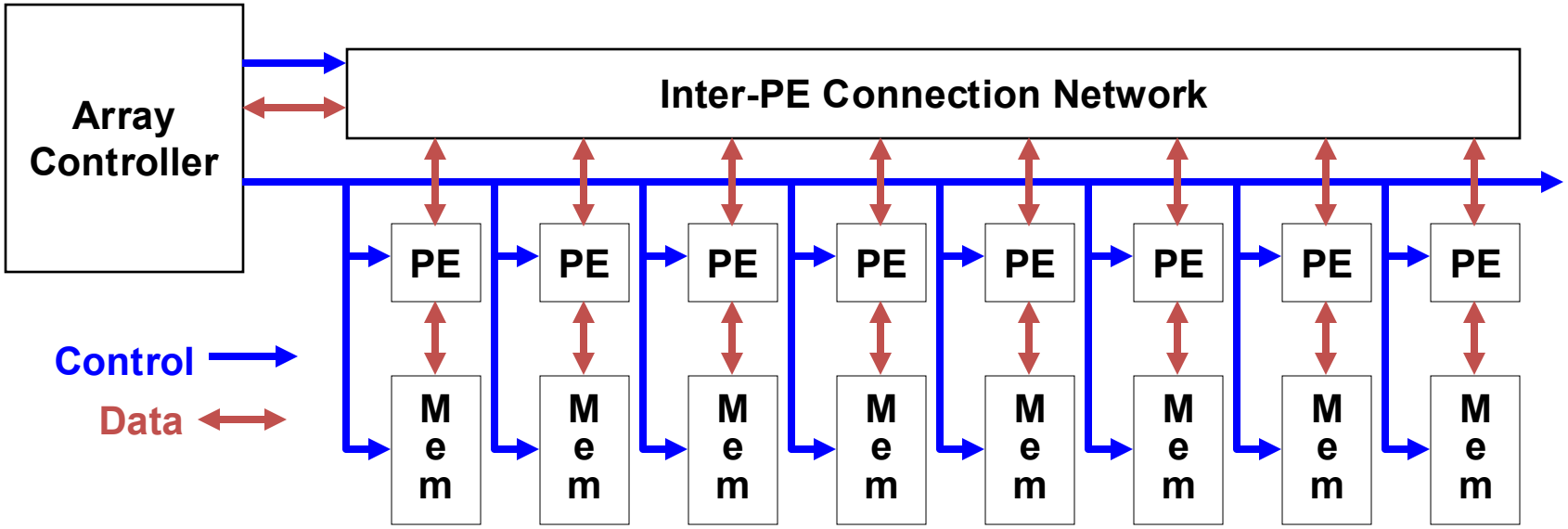
- Many of the early multiprocessors were SIMD – SIMD model received great attention in the ’80’s, today is applied only in very specific instances (vector processors, multimedia instructions);
- MIMD has emerged as architecture of choice for general-purpose multiprocessors
- Let’s see these architectures more in details..

SIMD - Single Instruction Multiple Data

- Same instruction executed by multiple processors using different data streams.
- Each processor has its own data memory.
- Single instruction memory and control processor to fetch and dispatch instructions
- Processors are typically special-purpose.
- Simple programming model.

SIMD Architecture

Central controller broadcasts instructions to multiple processing elements (PEs)



- ✓ Only requires one controller for whole array
- ✓ Only requires storage for one copy of program
- ✓ All computations fully synchronized

SIMD model

- Synchronized units: single Program Counter
- Each unit has its own addressing registers
  - Can use different data addresses
- Motivations for SIMD:
  - Cost of control unit shared by all execution units
  - Only one copy of the code in execution is necessary
- Real life:
  - SIMD have a mix of SISD instructions and SIMD
  - A host computer executes sequential operations
  - SIMD instructions sent to all the execution units, which has its own memory and registers and exploit an interconnection network to exchange data

Reality: Sony Playstation 2000

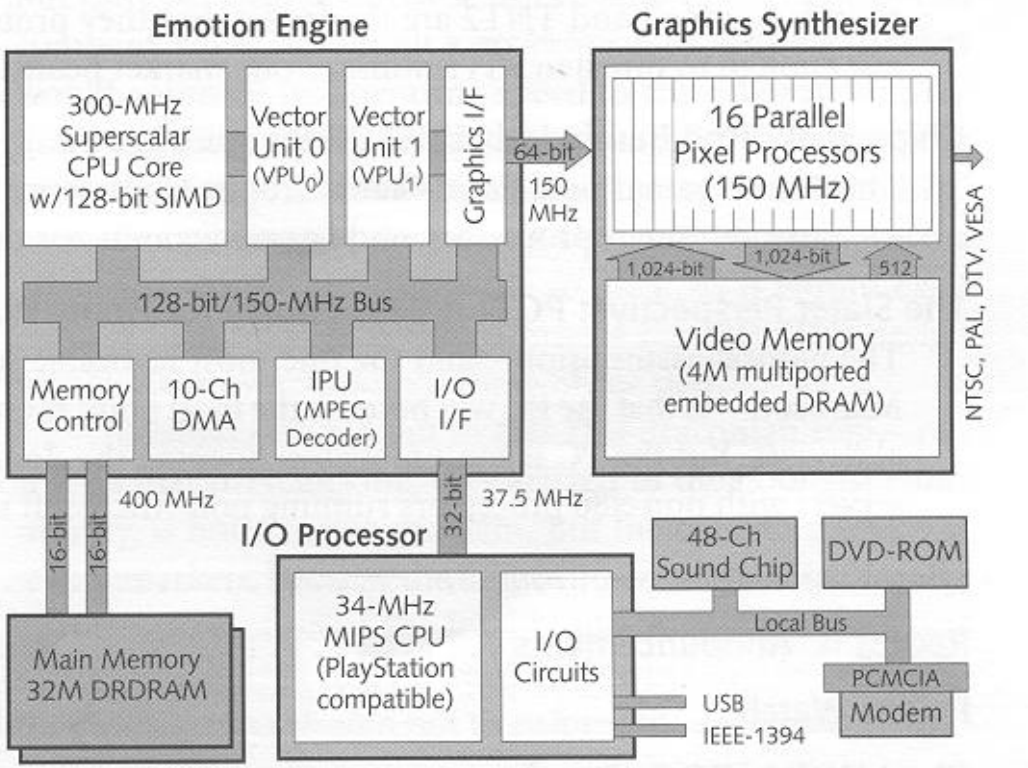
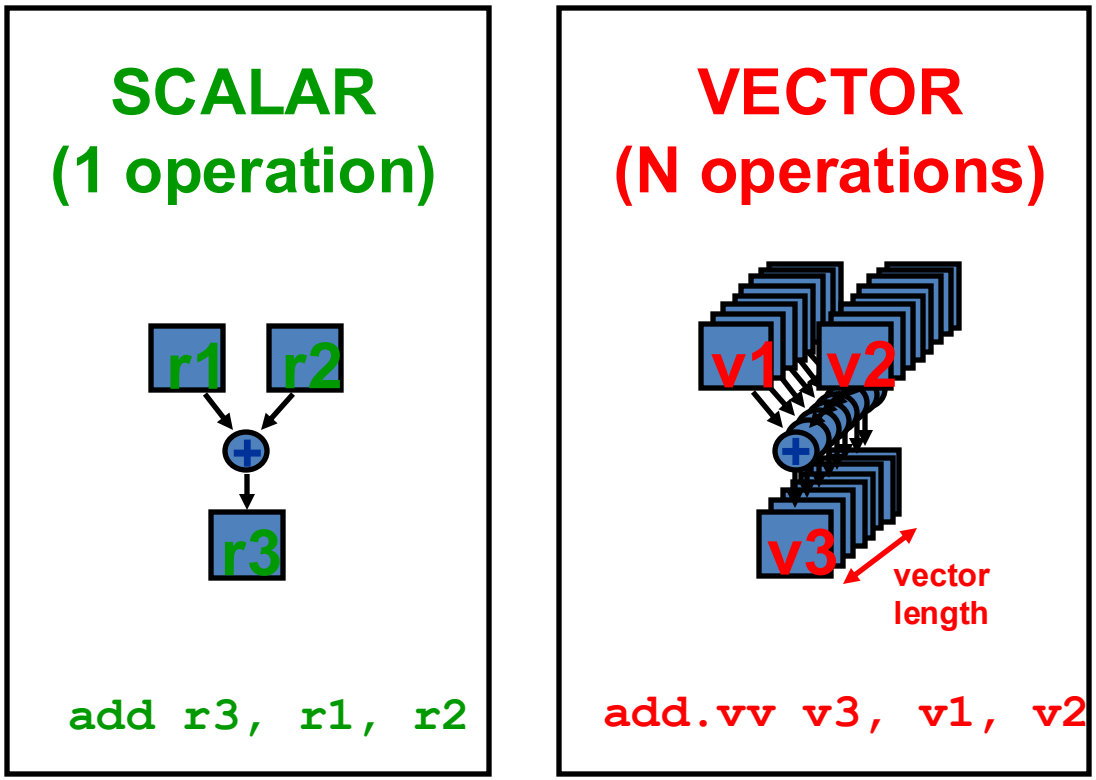


Figure 1. PlayStation 2000 employs an unprecedented level of parallelism to achieve workstation-class 3D performance.

Figure 2. PlayStation 2000 screenshot. (Source: Namco)

Alternative Model: Vector Processing

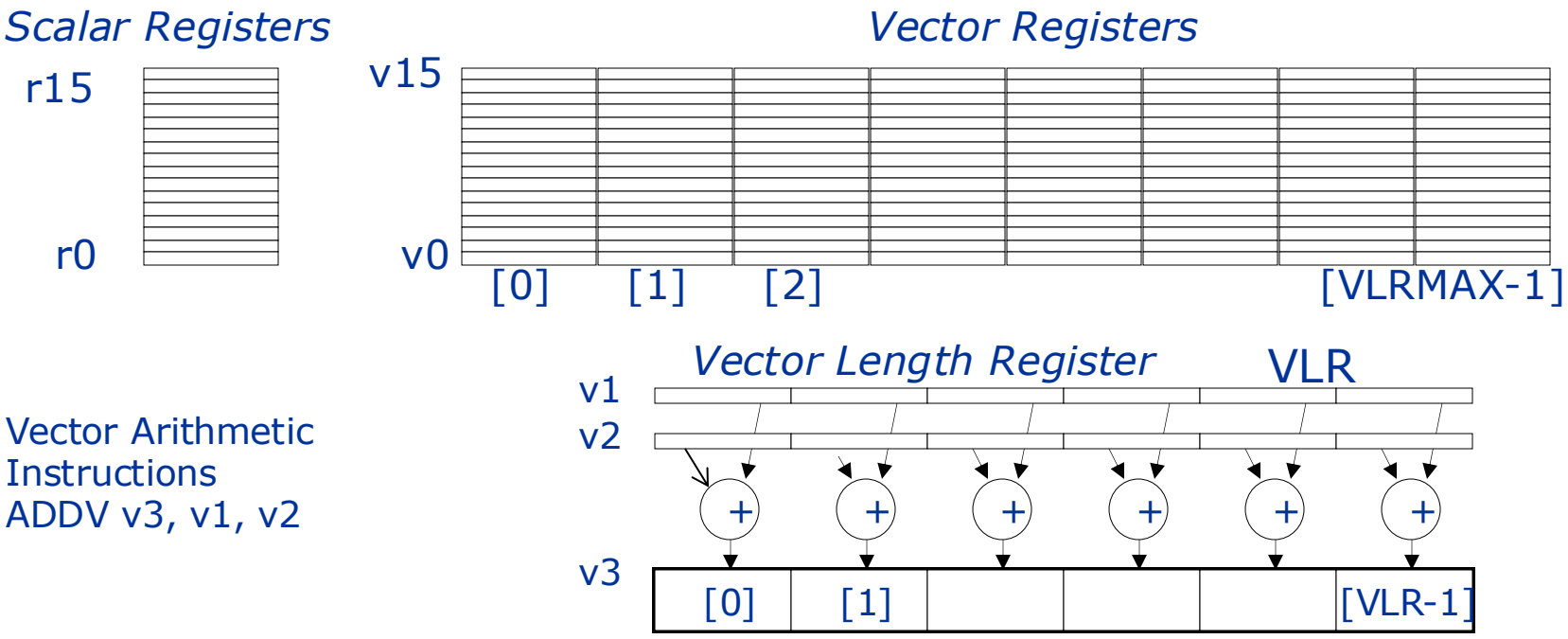
- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"



Styles of Vector Architectures

- A vector processor consists of a pipelined scalar unit (may be out-of order or VLIW) + vector unit
- *memory-memory vector processors*: all vector operations are memory to memory
- *vector-register processors*: all vector operations between vector registers (except load and store)
  - Vector equivalent of load-store architectures
  - Includes all vector machines since late 1980s: Cray, Convex, Fujitsu, Hitachi, NEC

Vector programming model



Vector Arithmetic Instructions
ADDV v3, v1, v2

Vector Code Example

```
# C code
for (i=0;i<64; i++)
  C[i] = A[i]+B[i];
```

Vector Code Example

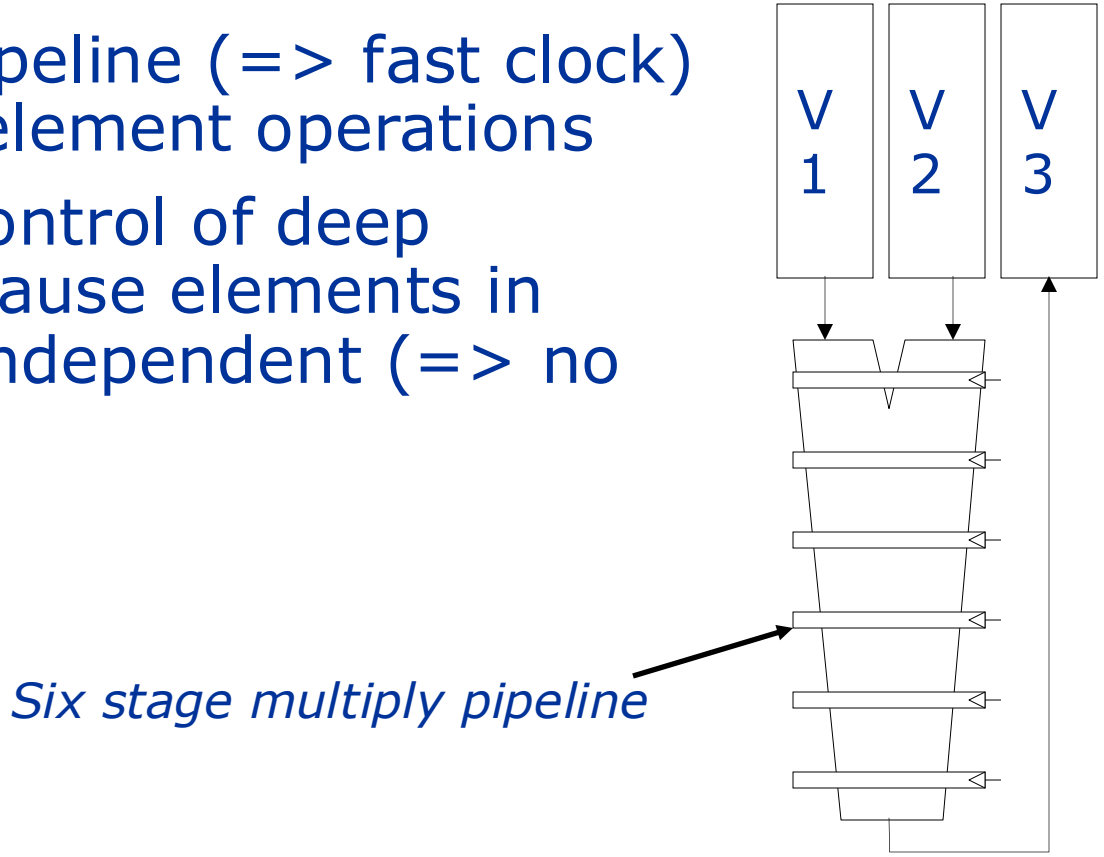
# C code	# Scalar Code	
for (i=0;i<64; i++) C[i] = A[i]+B[i];	LI R4, #64 loop: L.D F0, 0(R1) L.D F2, 0(R2) ADD.D F4, F2, F0 S.D F4, 0(R3) DADDIU R1, 8 DADDIU R2, 8 DADDIU R3, 8 DSUBIU R4, 1 BNEZ R4, loop	

Vector Code Example

# C code	# Scalar Code	# Vector Code
for (i=0;i<64; i++) C[i] = A[i]+B[i];	LI R4, #64 loop: L.D F0, 0(R1) L.D F2, 0(R2) ADD.D F4, F2, F0 S.D F4, 0(R3) DADDIU R1, 8 DADDIU R2, 8 DADDIU R3, 8 DSUBIU R4, 1 BNEZ R4, loop	LI VLR, #64 LV V1, R1 LV V2, R2 ADDV.D V3,V1,V2 SV V3, R3

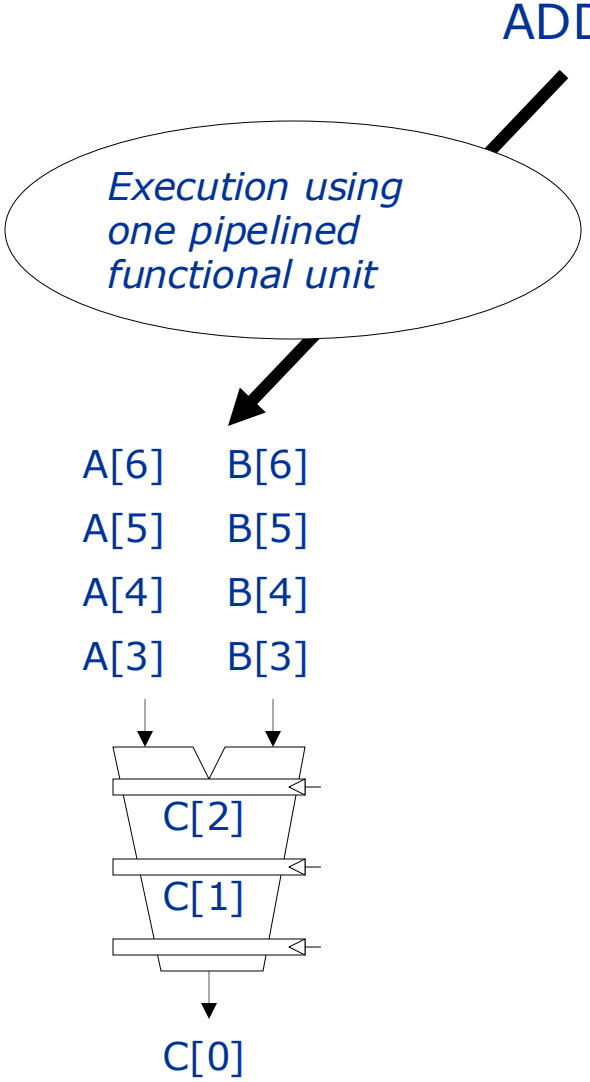
Vector Arithmetic Execution

- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)

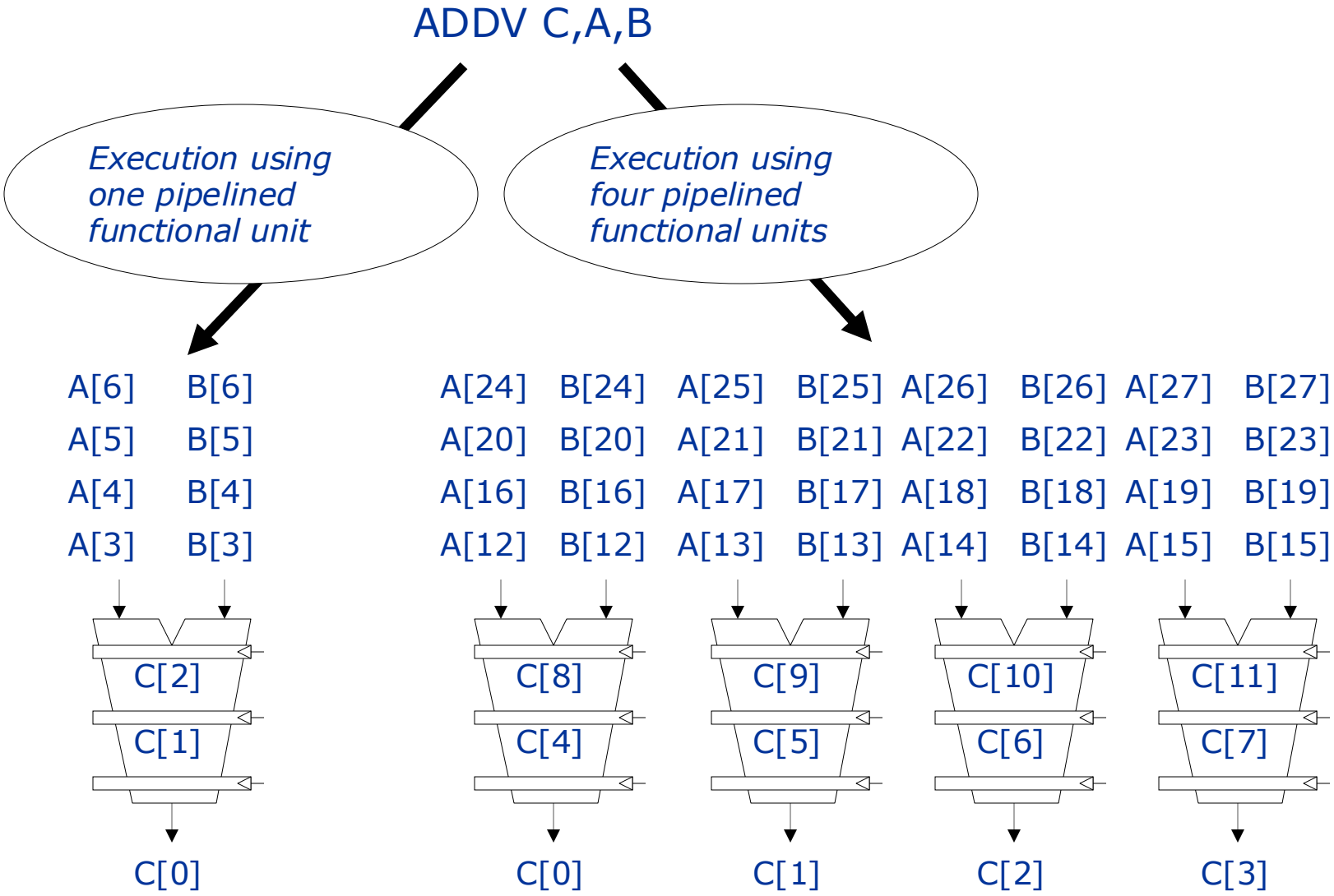


V3 ← v1 * v2

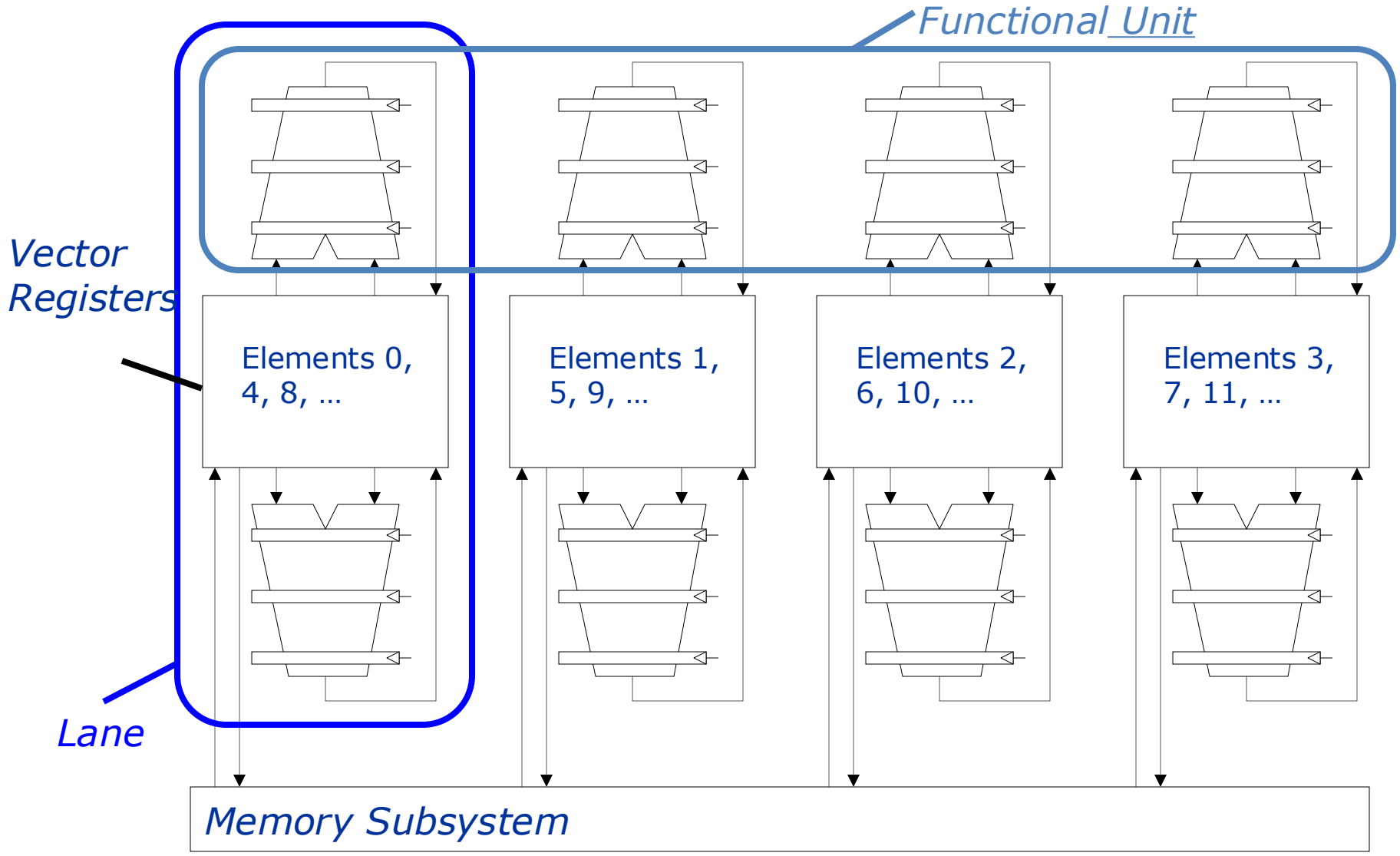
Vector Instruction Execution



Vector Instruction Execution



Vector Unit Structure



Vector Applications

Limited to scientific computing?

- Multimedia Processing (compress., graphics, audio synth, image proc.)
- Standard benchmark kernels (Matrix Multiply, FFT, Convolution, Sort)
- Lossy Compression (JPEG, MPEG video and audio)
- Lossless Compression (Zero removal, RLE, Differencing, LZW)
- Cryptography (RSA, DES/IDEA, SHA/MD5)
- Speech and handwriting recognition
- Operating systems/Networking (`memcpy`, `memset`, parity, checksum)
- Databases (hash/join, data mining, image/video serving)
- Language run-time support (stdlib, garbage collection)
- even SPECint95

Why MIMD?

Why MIMD?

- MIMDs are flexible – they can function as single-user machines for high performances on one application, as multiprogrammed multiprocessors running many tasks simultaneously, or as some combination of such functions;

Why MIMD?

- MIMDs are flexible – they can function as single-user machines for high performances on one application, as multiprogrammed multiprocessors running many tasks simultaneously, or as some combination of such functions;

- Can be built starting from standard CPUs (such is the present case nearly for all multiprocessors!).

MIMD - Multiple Instruction Multiple Data

- Each processor fetches its own instructions and operates on its own data.
- Processors are often off-the-shelf microprocessors.
- Scalable to a variable number of processor nodes.
- Flexible:
    - single-user machines focusing on high-performance for one specific application,
    - multi-programmed machines running many tasks simultaneously,
    - some combination of these functions.
- Cost/performance advantages due to the use of off-the-shelf microprocessors.
- Fault tolerance issues.

MIMD

- To exploit a MIMD with ***n*** processors
  - at least ***n*** threads or processes to execute
  - independent threads typically identified by the programmer or created by the compiler.
  - Parallelism is contained in the threads
    - *thread-level parallelism*.

MIMD

- To exploit a MIMD with ***n*** processors
  - at least ***n*** threads or processes to execute
  - independent threads typically identified by the programmer or created by the compiler.
  - Parallelism is contained in the threads
    - *thread-level parallelism*.

- Thread: from a large, independent process to parallel iterations of a loop. Important: *parallelism is identified by the software* (not by hardware as in superscalar CPUs!)... keep this in mind, we'll use it!

MIMD Machines

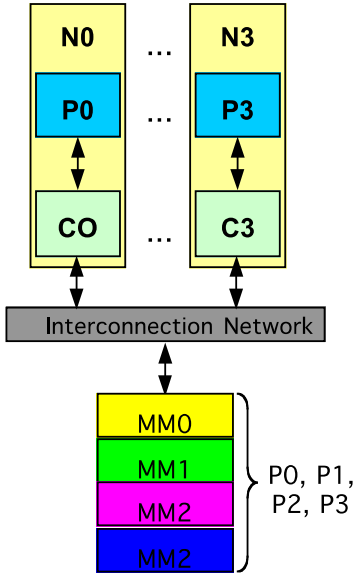
Existing MIMD machines fall into 2 classes, depending on the number of processors involved, which in turn dictates a memory organization and interconnection strategy.

MIMD Machines

Existing MIMD machines fall into 2 classes, depending on the number of processors involved, which in turn dictates a memory organization and interconnection strategy.

Centralized shared-memory architectures

- at most few dozen processor chips (< 100 cores)
- Large caches, single memory multiple banks
- Often called symmetric multiprocessors (SMP) and the style of architecture called Uniform Memory Access (UMA)

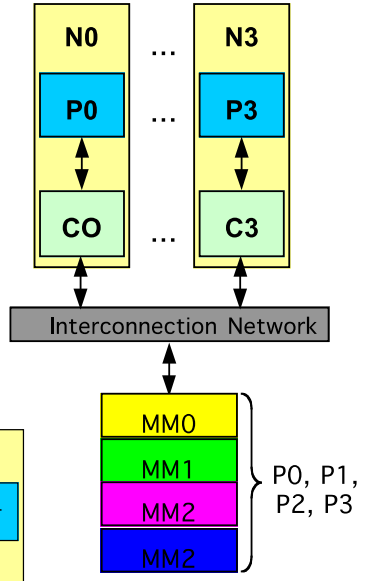


MIMD Machines

Existing MIMD machines fall into 2 classes, depending on the number of processors involved, which in turn dictates a memory organization and interconnection strategy.

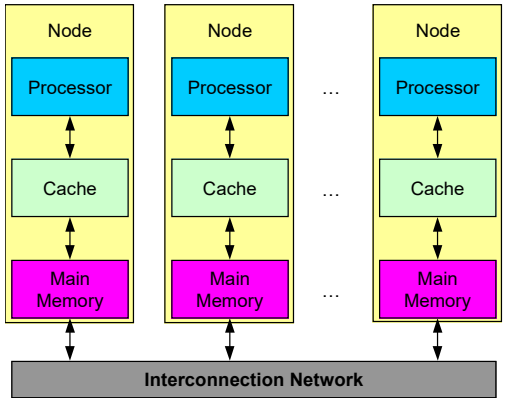
Centralized shared-memory architectures

- at most few dozen processor chips (< 100 cores)
- Large caches, single memory multiple banks
- Often called symmetric multiprocessors (SMP) and the style of architecture called Uniform Memory Access (UMA)

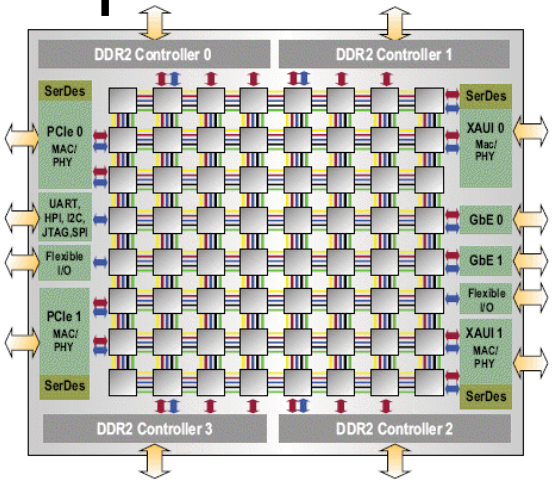


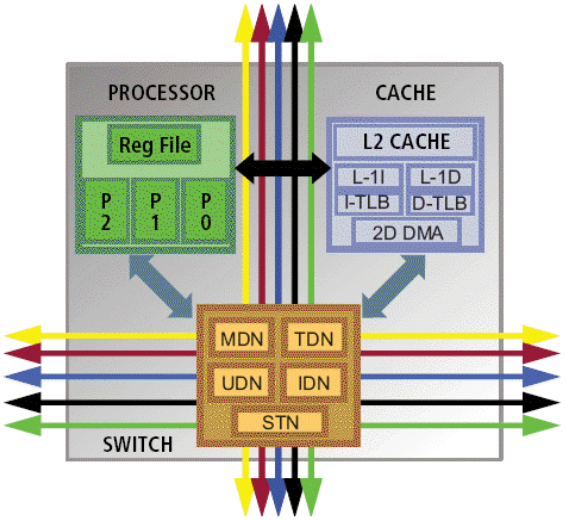
Distributed memory architectures

- To support large processor counts
- Requires high-bandwidth interconnect
- Disadvantage: data communication among processors



The Tilera Example





TILERA®

Key issues to design multiprocessors

- How many processors?
- How powerful are processors?
- How do parallel processors share data?
- Where to place the physical memory?
- How do parallel processors cooperate and coordinate?
- What type of interconnection topology?
- How to program processors?
- How to maintain cache coherency?
- How to maintain memory consistency?
- How to evaluate system performance?

Multithreading and Multiprocessors

Politecnico di Milano

Christian Pilato <christian.pilato@polimi.it>