Hardware Multithreading

Hardware multithreading

Thread

Light weight process which shares a single address space
Includes PC, register state and stack.

Process

One or more threads, address space, OS state.

Hardware Multithreading

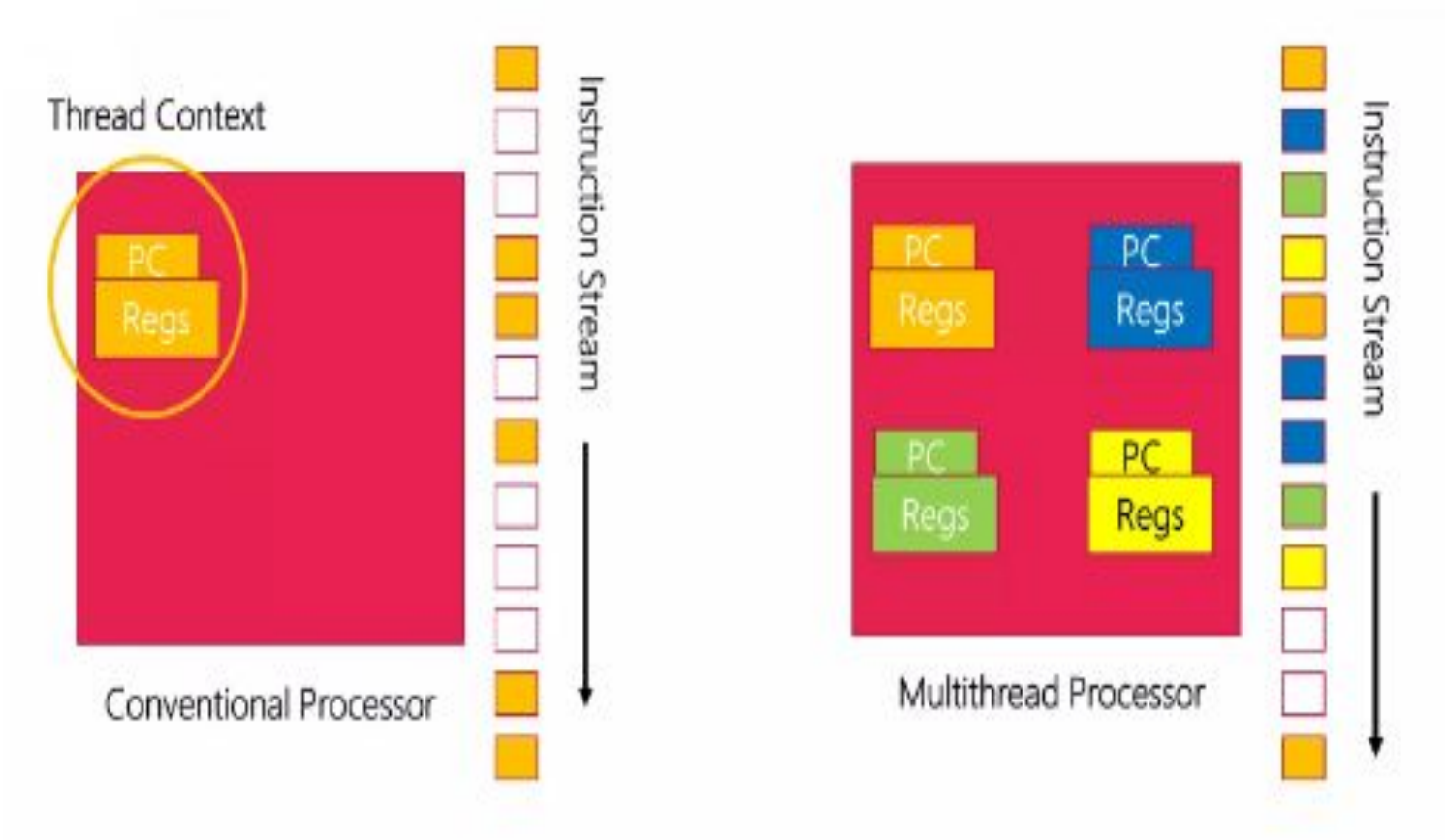
Increasing utilization of a processor by switching to another thread when one thread is stalled.

Hardware Multithreading

Hardware multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion to try to utilize the hardware resources efficiently.

1. Fine Grained Multithreading
2. Coarse Grained Multithreading
3. Simultaneous Multithreading

Hardware Multithreading



Resource sharing in multithreaded processor

Category	Resources
Replicated	PC, Architectural Registers, Register Renaming Logic
Partitioned	Re-Order Buffer (ROB), Load-Store Buffer, Queue
Shared	Caches, Physical Registers, Execution Unit

Coarse Grained Multithreading (CGMT)

We always executes instructions from a single thread continuously

Thread scheduling: Thread switching on long latency events (L2 cache misses)

Pipeline partitioning: Pipeline will not be partitioned, flushing.

Simple, Improved throughput, low cost

Not suitable for out-of-order processing.

Fine Grained Multithreading (FGMT)

Thread scheduling: Thread Switching on every clock cycle (Round Robin)

Pipeline Partitioning: Dynamic, No flush

Conceptually simple, High throughput

Very poor single thread performance

Simultaneous Multithreading (SMT)

FGMT + Superscalar processing

Thread scheduling: Thread switching on every clock cycle (round robin)

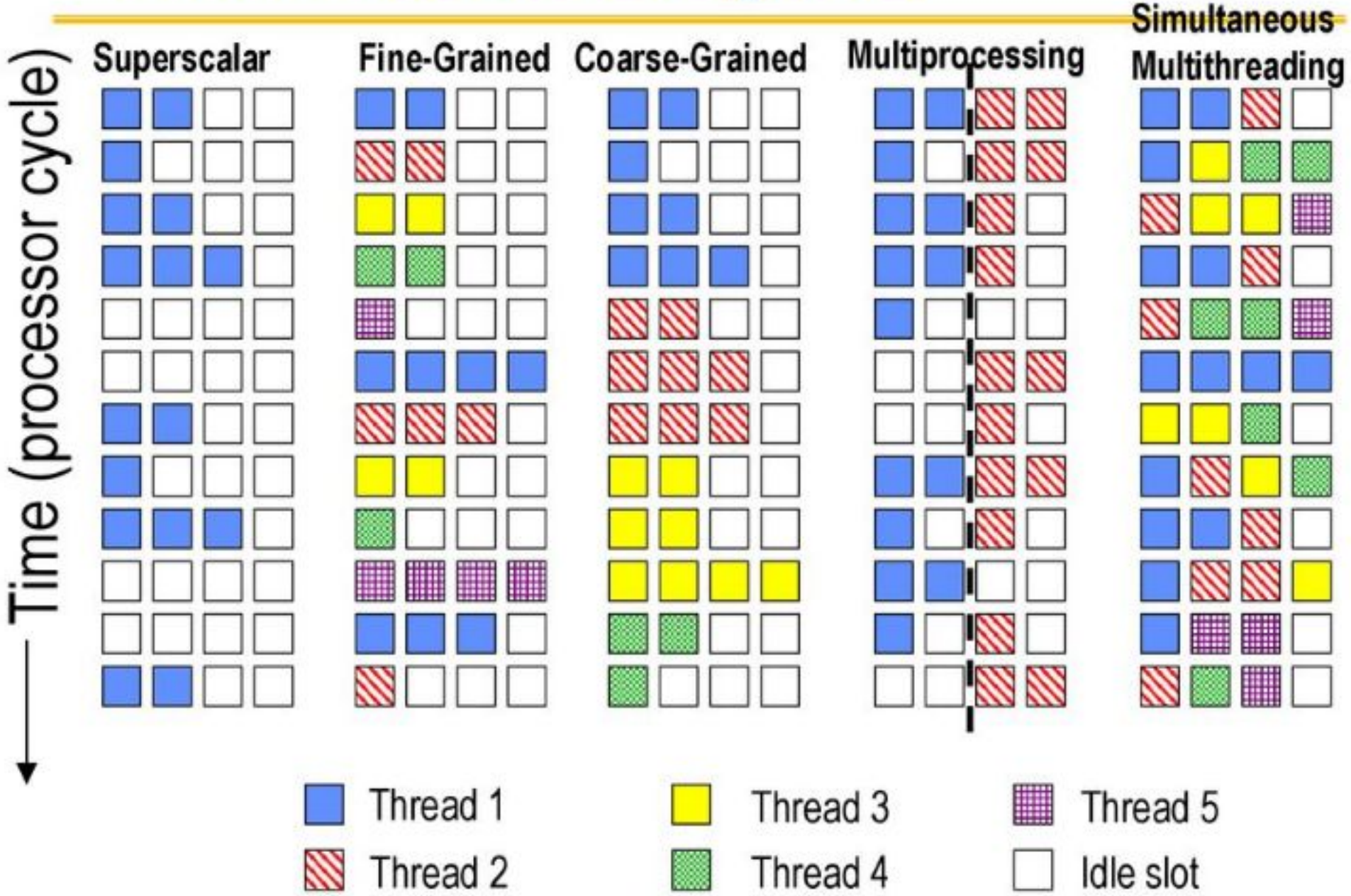
Pipeline Partitioning: Dynamic, No flush

Improved throughput, hide memory latency

Resource Partioning: Static and dynamic

Increased conflict in shared resources

Multithreaded Categories



Vector (SIMD) Processing

Data Parallelism

- Concurrency arises from performing the same operations on different pieces of data
  - Single instruction multiple data (SIMD)
  - E.g., dot product of two vectors

- Contrast with data flow
  - Concurrency arises from executing different operations in parallel (in a data driven manner)
- Contrast with thread ("control") parallelism
  - Concurrency arises from executing different threads of control in parallel

- SIMD exploits instruction-level parallelism
  - Multiple instructions concurrent: instructions happen to be the same

SIMD Processing

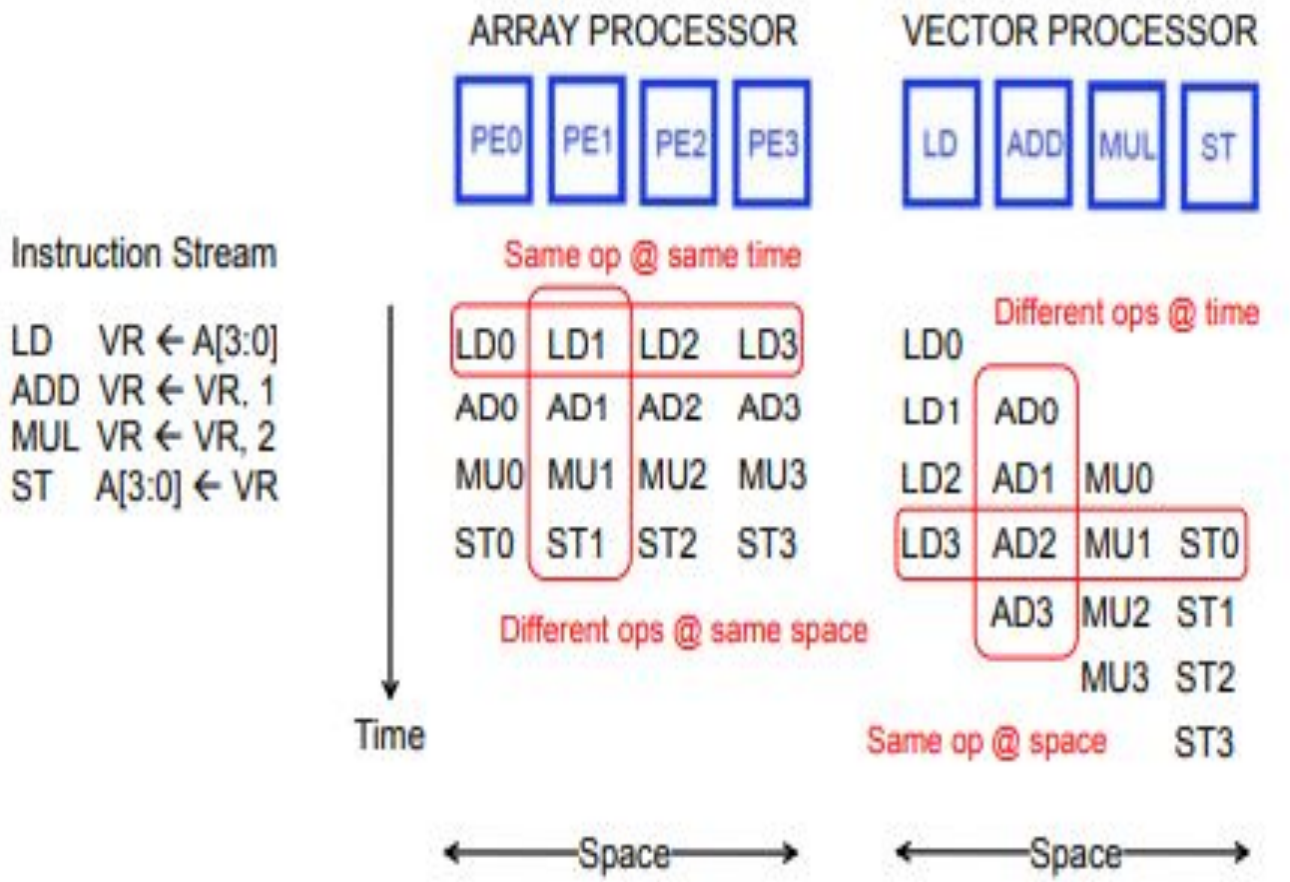
- Single instruction operates on multiple data elements
  - In time or in space

Multiple processing elements
  - Time-space duality

Array processor: Instruction operates on multiple data elements at the same time

Vector processor: Instruction operates on multiple data elements in consecutive time steps

Array vs. Vector Processors



Vector Processors

- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors

```
for (i = 0; i<=49; i++)
C[i] = (A[i] + B[i]) / 2
```

- A vector processor is one whose instructions operate on vectors rather than scalar (single data) values

- Basic requirements
  - Need to load/store vectors → vector registers (contain vectors)
  - Need to operate on vectors of different lengths → vector length register (VLEN)
  - Elements of a vector might be stored apart from each other in memory → vector stride register (VSTR)

    Stride: distance between two elements of a vector

A vector instruction performs an operation on each element in consecutive cycles

- Vector functional units are pipelined
- Each pipeline stage operates on a different data element

Vector instructions allow deeper pipelines

- No intra-vector dependencies no hardware interlocking within a vector
- No control flow within a vector
- Known stride allows prefetching of vectors into cache/memory

Vector Processor Advantages

+ No dependencies within a vector
  - Pipelining, parallelization work well
  - Can have very deep pipelines, no dependencies!

+ Each instruction generates a lot of work
  - Reduces instruction fetch bandwidth

+ Highly regular memory access pattern
  - Interleaving multiple banks for higher memory bandwidth
  - Prefetching

+ No need to explicitly code loops
  - Fewer branches in the instruction sequence

The vector instruction

VectorAdd.S Vi, Vj, Vk

computes L sums using the elements in vector registers Vj and Vk, and places the resulting sums in vector register Vi. Similar instructions are used to perform other arithmetic operations.

Special instructions are needed to transfer multiple data elements between a vector register and the memory. The instruction

VectorLoad.S Vi, X(Rj)

causes L consecutive elements beginning at memory location X + [Rj] to be loaded into vector register Vi. Similarly, the instruction

VectorStore.S Vi, X(Rj)

causes the contents of vector register Vi to be stored as L consecutive elements in the memory.

Vectorization

- In a source program written in a high-level language, loops that operate on arrays of integers or floating-point numbers are vectorizable if the operations performed in each pass are independent of the other passes.

- Using vector instructions reduces the number of instructions that need to be executed and enables the operations to be performed in parallel on multiple ALUs.

- A vectorizing compiler can recognize such loops, if they are not too complex, and generate vector instructions.

Example of loop vectorization

```
for (i = 0; i < N; i++)
    A[i] = B[i] + C[i];
```

(a) A C-language loop to add vector elements

	Move	R5, #N	R5 is the loop counter.
LOOP:	Load	R6, (R3)	R3 points to an element in array B.
	Load	R7, (R4)	R4 points to an element in array C.
	Add	R6, R6, R7	Add a pair of elements from the arrays.
	Store	R6, (R2)	R2 points to an element in array A.
	Add	R2, R2, #4	Increment the three array pointers.
	Add	R3, R3, #4	
	Add	R4, R4, #4	
	Subtract	R5, R5, #1	Decrement the loop counter.
	Branch_if_[R5]> 0	LOOP	Repeat the loop if not finished.

(b) Assembly-language instructions for the loop

Example of loop vectorization

	Move	R5, #N	R5 counts the number of elements to process.
LOOP:	VectorLoad.S	V0, (R3)	Load *L* elements from array B.
	VectorLoad.S	V1, (R4)	Load *L* elements from array C.
	VectorAdd.S	V0, V0, V1	Add *L* pairs of elements from the arrays.
	VectorStore.S	V0, (R2)	Store *L* elements to array A.
	Add	R2, R2, #4*L	Increment the array pointers by *L* words.
	Add	R3, R3, #4*L	
	Add	R4, R4, #4*L	
	Subtract	R5, R5, #L	Decrement the loop counter by *L*.
	Branch_if_[R5]>0	LOOP	Repeat the loop if not finished.

(c) Vectorized form of the loop

Graphics Processing Units (GPUs)

- The increasing demands of processing for computer graphics has led to the development of specialized chips called graphics processing units (GPUs).
- The primary purpose of GPUs is to accelerate the large number of floating-point calculations needed in high-resolution three-dimensional graphics, such as in video games.
- A large GPU chip contains hundreds of simple cores with floating-point ALUs to perform them in parallel

- The processing cores in a GPU chip have a specialized instruction set and hardware architecture, which are different from those used in a general-purpose processor.

- An example is the Compute Unified Device Architecture (CUDA) that NVIDIACorporation uses for the cores in its GPU chips.

- To facilitate writing programs that involve a general-purpose processor and a GPU, an extension to the C programming language, called CUDA C, hasbeen developed by NVIDIA

Cache Coherence

Cache Coherence

- A shared-memory multiprocessor is easy to program. Each variable in a program has a unique address location in the memory, which can be accessed by any processor.

- However, each processor has its own cache.

- When any processor writes to a shared variable in its own cache, all other caches that contain a copy of that variable will then have the old, incorrect value.

- They must be informed of the change so that they can either update their copy to the new value or invalidate it.

- This is the issue of maintaining **cache coherence**, which requires having a consistent view of shared data in multiple caches

Write-Through Protocol

A write-through protocol can be implemented in one of two ways. One version is based on updating the values in other caches, while the second relies on invalidating the copies in other caches.

Write-Back protocol

Maintaining coherence with the write-back protocol is based on the concept of ownership of a block of data in the memory. Initially, the memory is the owner of all blocks, and the memory retains ownership of any block that is read by a processor to place a copy in its cache.

Snoopy Caches

- In a single-bus system, all transactions between processors and memory modules occur via requests and responses on the bus.
- In effect, they are broadcast to all units connected to the bus.
- Suppose that each processor cache has a controller circuit that observes, or snoops, all transactions on the bus.

Directory-Based Cache Coherence

- The concept of snoopy caches is easy to implement in single-bus systems.
- Large shared memory multiprocessors use interconnection networks such as rings and meshes.
- In such systems, broadcasting every single request to the caches of all processors is inefficient.
- A scalable, but more complex, solution to this problem uses directories in each memory module to indicate which nodes may have copies of a given block in the shared state.