Unit 5

ADVANCES IN ARCHITECTURE

Introduction to Parallel Architecture

The use of the most efficient algorithms on computers capable of the highest performance to solve the most demanding problems

- High Performance Computing
- Higher speed (solve problems faster)
- Higher computational power (solve larger problems)

FLOPS, or FLOP/S: **FLoating-point Operations Per Second**

Name	Unit	Value
kiloFLOPS	kFLOPS	10^3
megaFLOPS	MFLOPS	10^6
gigaFLOPS	GFLOPS	10^9
teraFLOPS	TFLOPS	10^{12}
petaFLOPS	PFLOPS	10^{15}
exaFLOPS	EFLOPS	10^{18}
zettaFLOPS	ZFLOPS	10^{21}
yottaFLOPS	YFLOPS	10^{24}

Moore's Law:

Moore's perception that the number of transistors on a microchip **doubles every two years**, though the cost of computers is halved. The law was named after Intel cofounder Gordon E. Moore

- Moore's Law (published in an article April 19, 1965 in Electronics Magazine).
- Moores Law is a computing term, which originated around the 1970's
- Since the 1970s, the power of computers has doubled every year or and a half, yielding computers which are millions of times more powerful than their ancestors of a half century ago
- The law states that processor speeds, or overall processing power for computers will double about every 18 month's
- Deals with steady rate miniaturization of technology

Equation: Pn = Po x 2^n
- **Pn** = computer processing power in **future years**
- **Po** = computer processing power in the **beginning year**
- **n** = **number of years** to develop a new microprocessor divided by **2** (ie. every two years)

Example:
In 1988, the number of transistors in the Intel 386 SX microprocessor was 2,75,000. What were the transistors counts of the Pentium II Intel microprocessor in 1997 ?

Solution:
If Intel doubles the number of transistors every two years, the new processor would have
Pn = 275,000 x 2^n (where n = 9/2 = 4.5)
= 275,000 x 22.63
= 6.2 million transistors

- In 1997, the Pentium II had 7.5 million transistors. In other words, since 1988 up until 1997 (**9 year span**), Intel has been **doubling the number of transistors** in its microprocessors in less than every two years

Shift From Sequential to Parallel Processing

- Memory-wall challenge
- Power-wall challenge

Technique to **Improve the Performance / Speed-up**, inspired by **Amdahl's Law**

It relates the improvement of the system's performance with the parts that didn't perform well, like we need to take care of the performance of that parts of the systems.

$$OverallSpeedup = \frac{1}{(1-f) + \frac{f}{s}}$$

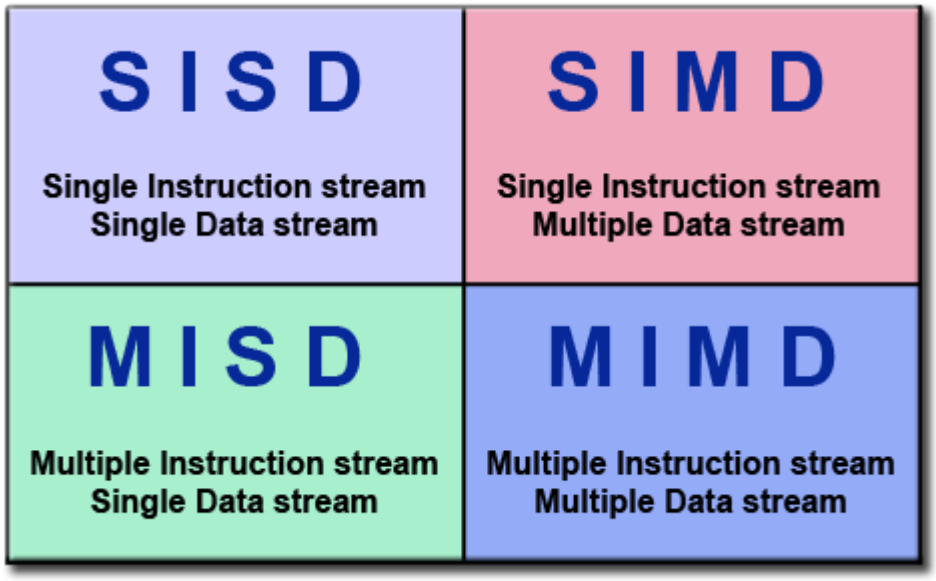
S= speed up factor
F= fraction of program which can be optimized or speed up factor can be applied.
(1-f)= fraction of program on which speed up factor cannot be applied.

Flynn's Taxonomy of Computer Architecture

Flynn′s taxonomy is a categorization of forms of parallel computer architectures. From the viewpoint of the assembly language programmer, parallel computers are classified by the concurrency in processing sequences (or streams), data, and instructions.

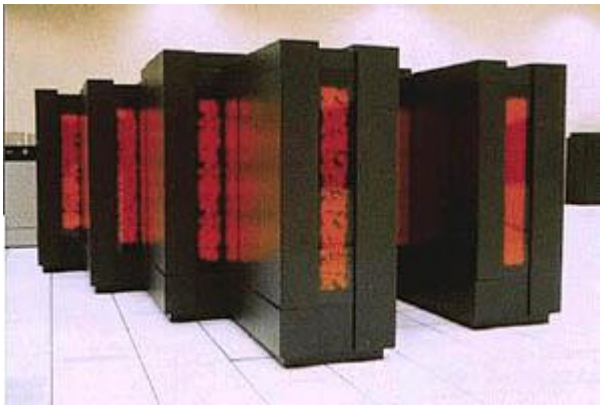
This results in four classes

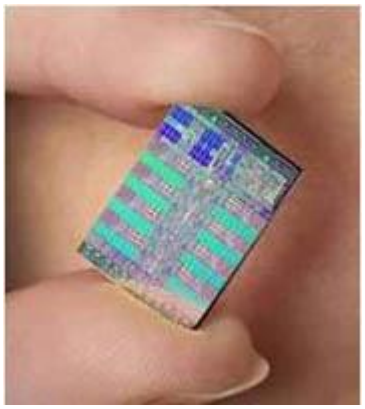
- SISD (single instruction, single data),
- SIMD (single instruction, multiple data),
- MISD (multiple instruction, single data), and
- MIMD (multiple instruction, multiple data)



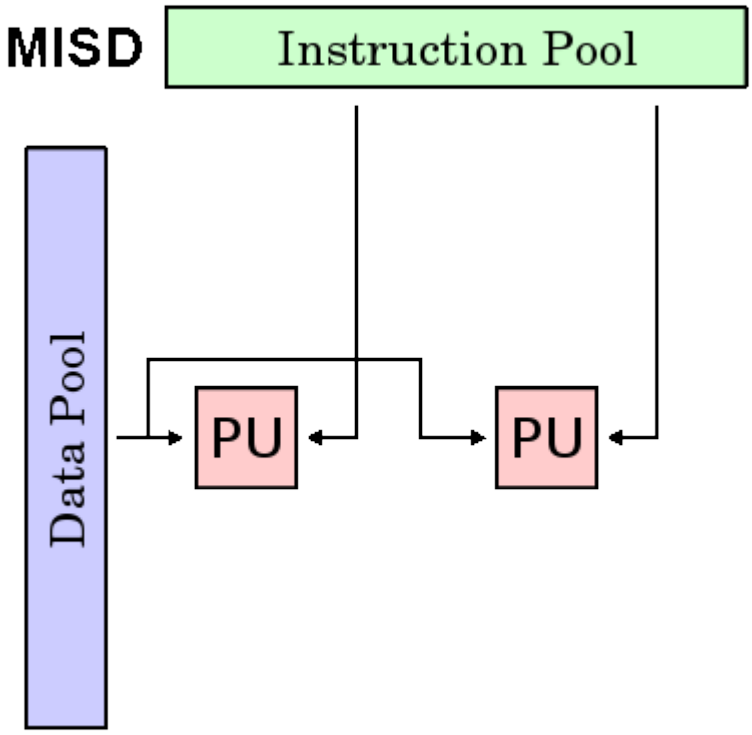
SISD − single instruction, single data stream; this is the traditional uniprocessor which includes pipelined, superscalar, and VLIW processors.

SIMD − single instruction, multiple data stream, which includes array processors and vector processors .



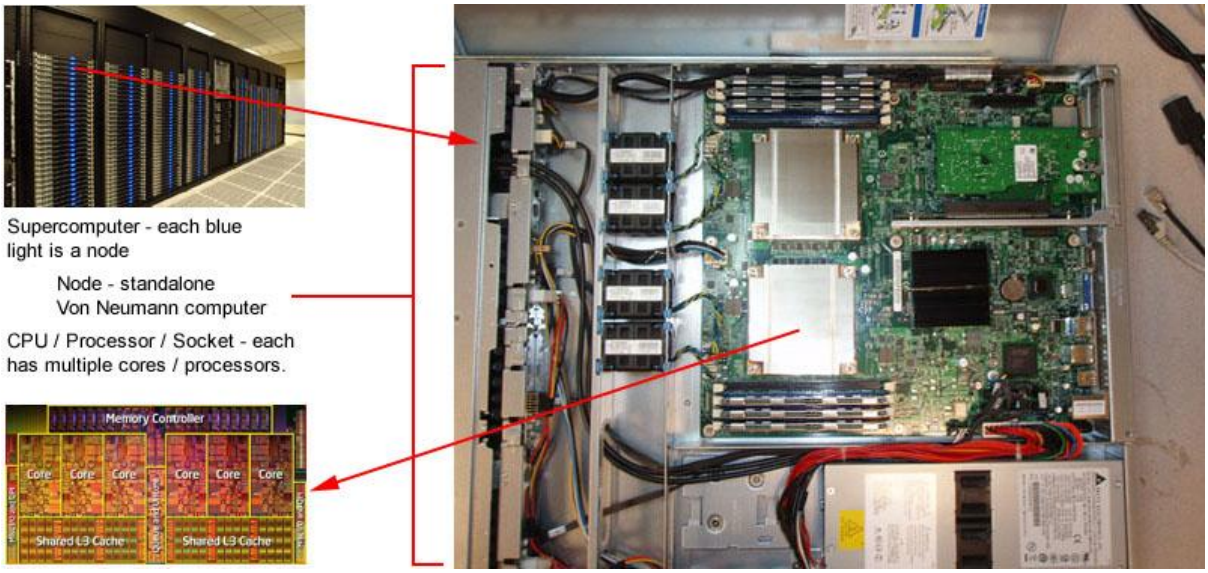


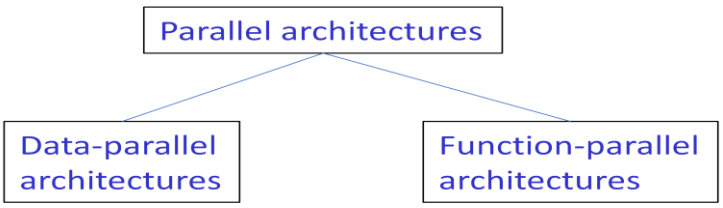
MISD − multiple instruction, single data stream, which includes systolic arrays, GPUs, and dataflow machines



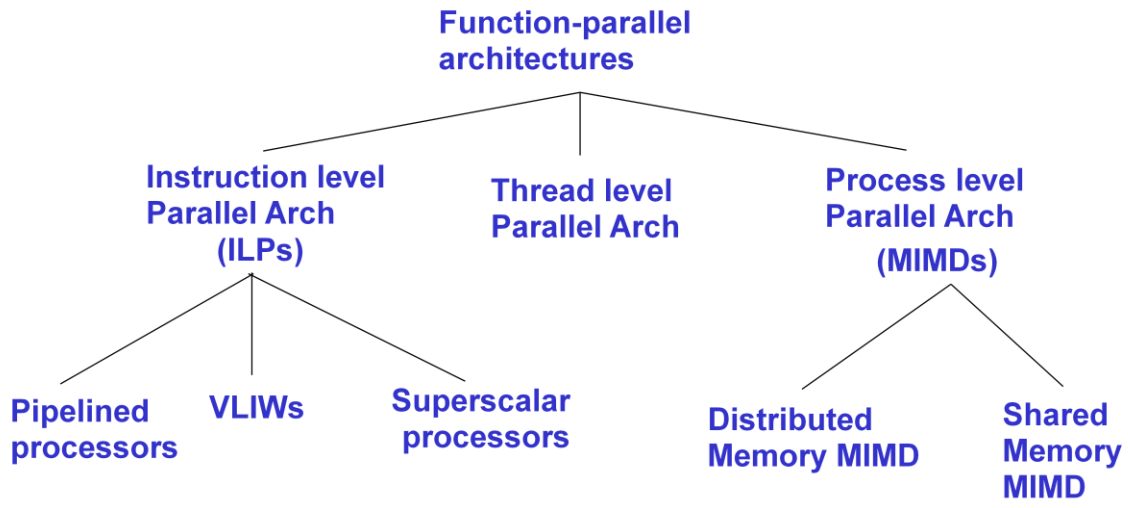
MIMD − multiple instruction, multiple data stream, which includes traditional multiprocessors (multi-core and multi-threaded) as well as the newer work of networks of workstations

General Parallel Computing Terminology

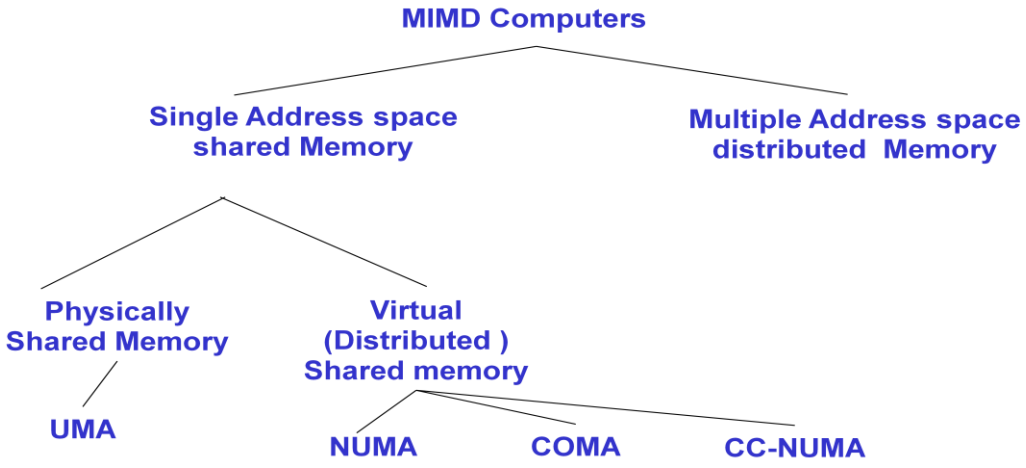




Functional Parallel Architectures



Classification of MIMD computers



Parallel Computer Memory Architectures (Refer PPTs)

- Shared Memory
- Distributed Memory
- Hybrid Distributed-Shared Memory

Parallel Programming Models

There are several parallel programming models in common use:

- Shared Memory (without threads)
- Threads
- Distributed Memory / Message Passing
- Data Parallel
- Hybrid
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

- **OpenMP: (Open Multi Processing):**
- API that Support multiprocessing in C, C++, Fortran. Now with Python also.
- **MPI: (Message Passing Interface):**
- C, C++, Fortran, Java, Python, Ocaml, R.....etc
- **CILK**: Customized C Language
- **CUDA** (Computer Unified Devise Architecture): for Nvidia GPU
- **Pthreads**

OpenMP Program to print **Hello-PESU** on **4 CPU** with **Shared Memory**

```
pragma omp parallel default(shared) private(iam, np)
 {
   np = omp_get_num_threads();
   iam = omp_get_thread_num();
   printf("Hello-PESU: from thread %d out of %d ", iam, np);
 }
```

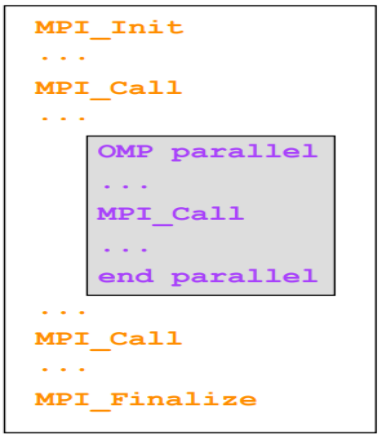
Hello-PESU from thread 0 out of 4
Hello-PESU from thread 2 out of 4
Hello-PESU from thread 1 out of 4
Hello-PESU from thread 3 out of 4
Hello-PESU from thread 0 out of 4
Hello-PESU from thread 2 out of 4
Hello-PESU from thread 1 out of 4
Hello-PESU from thread 3 out of 4

MPI Program to print Hello-PESU on 2 CPU with Distributed Memory

```
MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("Hello-PESU from rank %d out of %d processors\n",rank, numprocs);
```

Hello-PESU from rank 0 out of 2
Hello-PESU from rank 1 out of 2

Hybrid Programming



Hybrid (MPI + OpenMP) Program to print Hello-PESU on with 4 core and 2 CPU

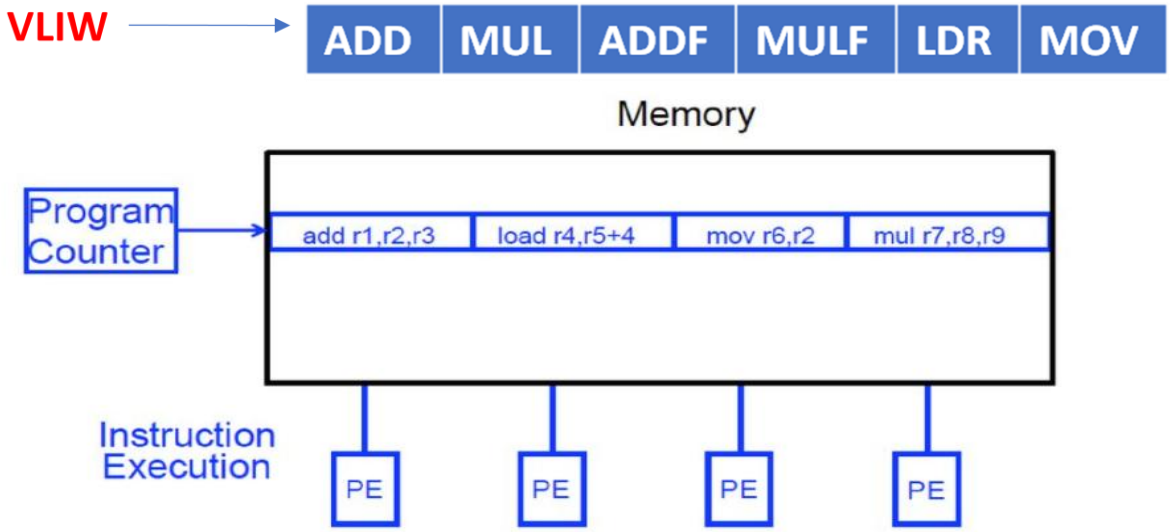
```
#include<omp.h>
#include<mpi.h>
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
#pragma omp parallel default(shared) private(iam, np)
  {
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello-PESU: from %d Thread out of %d Threads  & %d Rank of %d Processors",
iam, np,rank,numprocs);
  }
```

Hello-PESU from 0 Thread out of 4 Threads & 0 Rank of 2 Processors
Hello-PESU from 2 Thread out of 4 Threads & 1 Rank of 2 Processors
Hello-PESU from 1 Thread out of 4 Threads & 0 Rank of 2 Processors
Hello-PESU from 3 Thread out of 4 Threads & 1 Rank of 2 Processors
Hello-PESU from 0 Thread out of 4 Threads & 0 Rank of 2 Processors
Hello-PESU from 2 Thread out of 4 Threads & 1 Rank of 2 Processors
Hello-PESU from 1 Thread out of 4 Threads & 0 Rank of 2 Processors
Hello-PESU from 3 Thread out of 4 Threads & 1 Rank of 2 Processors

VLIW: Very Long Instruction Word

- Very long instruction word (VLIW) describes a computer processing architecture in which a language compiler or pre-processor breaks program instruction down into basic operations that can be performed by the processor in parallel (that is, at the same time)

- These operations are put into a very long instruction word which the processor can then take apart without further analysis, handing each operation to an appropriate functional unit.
- Multiple Independent Instructions are packed together by the Compiler



Draw Back of VLIW
If compiler cannot find the independent instructions to for Long Instructions
- Need to Recompile the code
- Need to insert NOP's

Truth of Parallel Execution

A program (or algorithm) which can be parallelized can be split up into two parts:

- **A part which cannot be parallelized**
- **A part which can be parallelized**
- T = Total time of serial execution
- T-F = Total time of non-parallizable part
- F = Total time of parallizable part (when executed serially, not in parallel)

T= F+ (T-F)

The total time to execute a program is set to 1. The parallelizable part of the programs consumes 60% of the execution time. What is the execution time of the program when executed on 2 processor ?

Solution:

The parallelizable part is thus equal = 0.6.

Time for non-parallelizable part is 1-0.6= 0.4 .

The execution time of the program with a parallelization factor of 2 (2 threads or CPUs executing the parallelizable part, so N is 2) would be:

T(2) = (1-0.6) + 0.6 / 2

= 0.4 + 0.6 / 2

= 0.4 + 0.3

= 0.7

Making the same calculation with a parallelization factor of 5 instead of 2 would look like this:

T(5) = (1-0.6) + 0.6 / 5

= 0.4 + 0.6 / 5

= 0.4 + 0.12

= 0.52

Gustafson's Law

Gustafon's Law: The proportion of the computations that are sequential, normally decreases as the problem size increases.

Note Gustafson's law is a "observed phenomena" and not a theorem

- Rather than assuming that the problem size is fixed, assume that the parallel execution time is fixed.
- As problem size is increases, increase the parallel processing units (N).
- Gustafson, makes the case that the serial section of the code does not increase with the problem size.

A Driving Metaphor

Amdahl's Law approximately suggests:

Problem: To travel from Parking area to University Gate in 500 CC Bullet

Step1: Reach the ***parking area gate*** in the speed 30 KM per hour

Step 2: Reach the ***University Main Gate*** 90 Km per hour.

Question: Did you reach fast? Though your speed was 90 Km/Hour in 500 CC Bullet?

Here Fixed Distance is Code size. Amdahl's law assumes code size to be constant.

Gustafson's Law approximately states:

Problem: To travel from Parking area to House in 500 CC Bullet

Step1: Reach the ***parking area gate*** in the speed 30 KM per hour

Step 2: Reach the **House** 90 Km per hour.

Question: Did you reach fast? when your speed was 90 Km/Hour in 500 CC Bullet, though your speed was 30 Km/hour to reach ***parking area gate***?

Limitations of Single Core

- **The Power Wall**

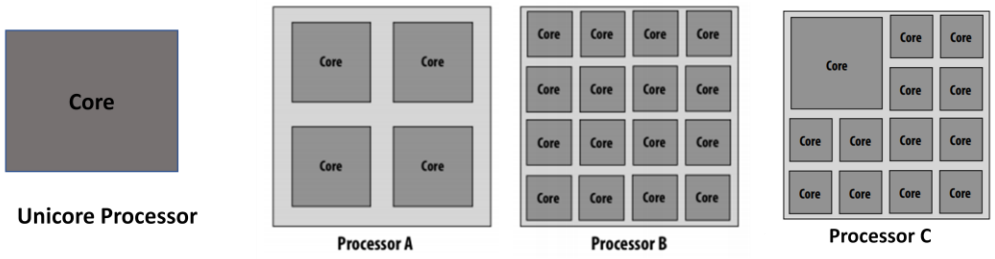
o Limit on the scaling of clock speeds.
o Ability to handle on-chip heat has reached a physical limit.

- **The Memory Wall**

o Need for bigger cache sizes.
o Memory access latency still not in line with processor speeds

- **The ILP Wall**

o Identifying Implicit Parallelism within the threads is limited in many Application
o Dependency
o Hardware Restrictions such as, Instruction Window Size (How many instructions can be fetched at a time).



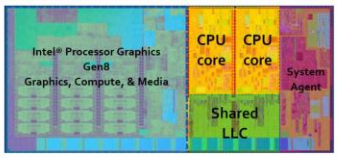
P = Total processing resources used to make Uni-core processor (e.g., transistors on a chip).
R= Resources dedicated to each processing core.

Let N=16

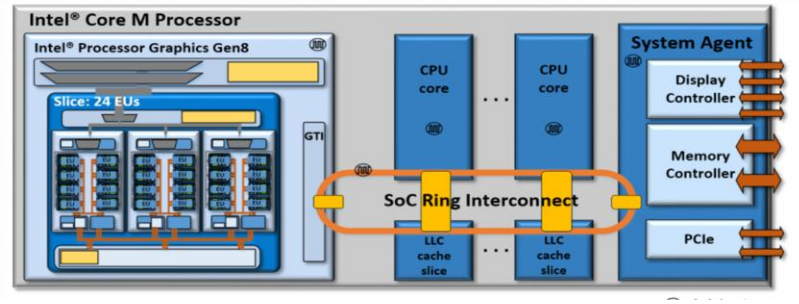
- Unicore Processor is made up of ***P=R=16*** resources.
- Each core in Processor A is made of ***R= 4*** resources.
- Each core in Processor B is made of ***R=1*** Resources.
- 1 Core of Processor C is made of ***R=4*** Resources and other 12 cores are made up of ***R=1*** Resources

Heterogeneous: Multiple Core Architecture

- Heterogeneous multicore systems have two or more cores that differ in architecture or microarchitecture.
- Example of heterogeneous multicore systems is the combination of a microprocessor core with a microcontroller class core (for example, mix of Cortex-A, Cortex-M or DSP cores.)
- Non-identical processor cores, support different Instruction Set Architecture (ISA).



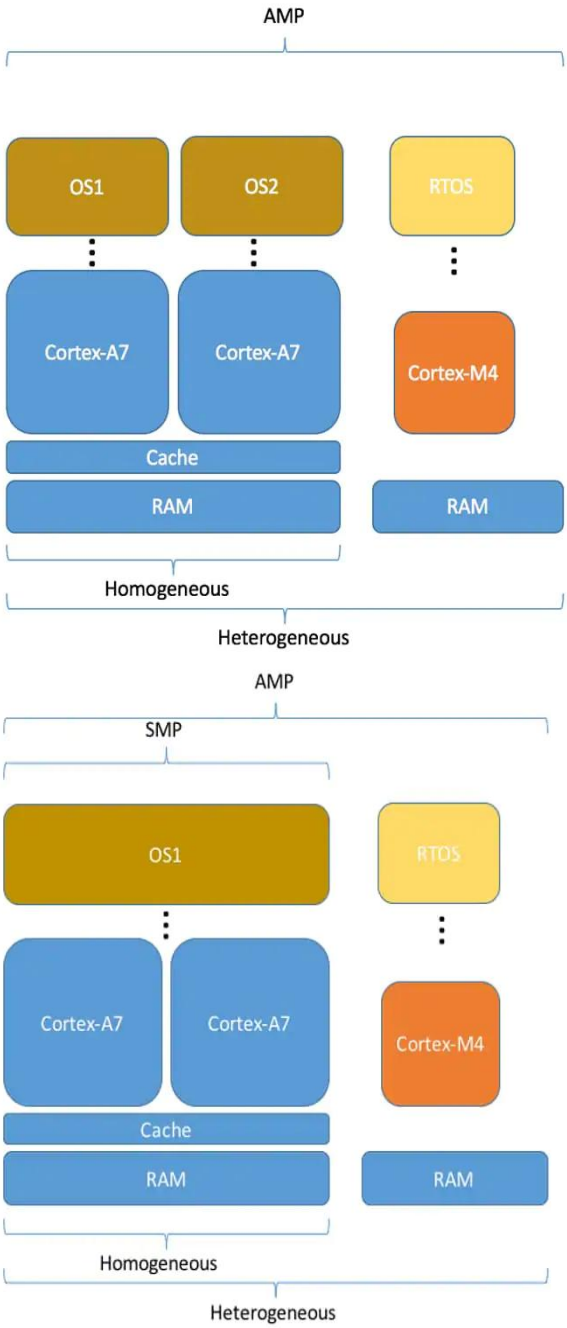
An Intel® Core™ M Processor SoC



Why?

- Most Efficient Processors are Heterogeneous.
- Power Efficient (Green Computing)

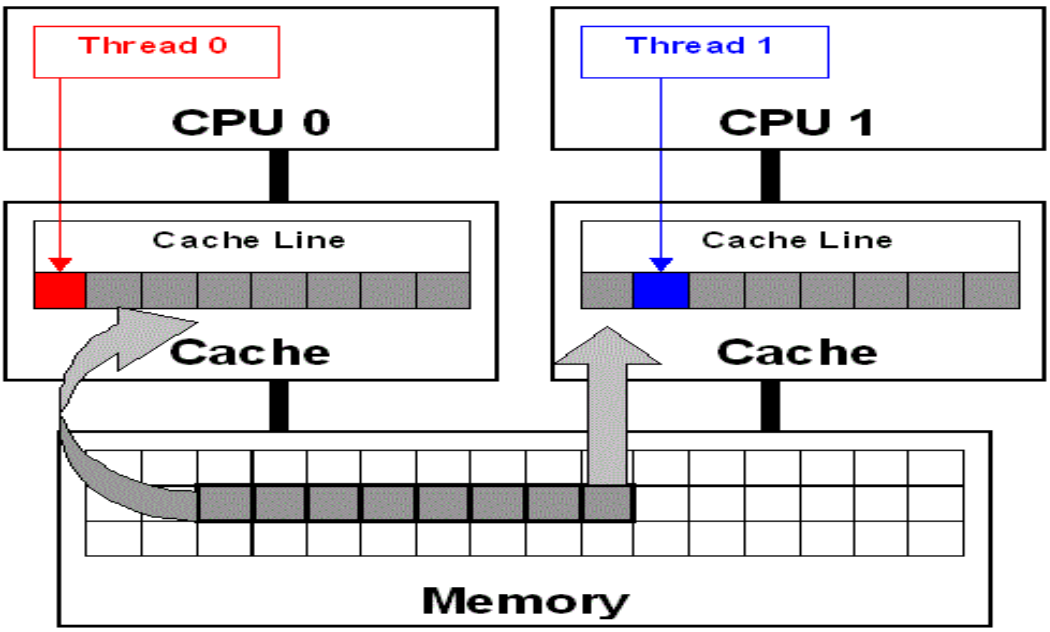
Heterogeneous ARM Processor



False Sharing in the shared Cache:

If two or more processors are writing data to different portions of the same cache line, then a lot of cache and bus traffic might result for effectively invalidating or updating every cached copy of the old line on other processors.

This is called "false sharing" or also "CPU cache line interference".



References:

https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial

https://pages.tacc.utexas.edu/~eijkhout/istc/html/parallel.html#Functionalparallelismversusdataparallelism

https://www.umsl.edu/~siegelj/information_theory/projects/Bajramovic/www.umsl.edu/_abdcf/Cs4890/link1.html

https://www.theverge.com/2018/7/19/17590242/intel-50th-anniversary-moores-law-history-chips-processors-future

https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial#Who

https://whatis.techtarget.com/definition/VLIW-very-long-instruction-word