**Math Co-Processor and Multi-Core Processor**

Table of Contents

[Math Co-Processor 2](#_Toc83227262)

[CISC and RISC Processors 3](#_Toc83227263)

[Multi-Core Processors 4](#_Toc83227264)

[Parallel Programming 5](#_Toc83227265)

[Intel Processors 8](#_Toc83227266)

[Dual Core and Core 2 Duo 8](#_Toc83227267)

[Core i3 8](#_Toc83227268)

[Core i5 9](#_Toc83227269)

[Core i7 9](#_Toc83227270)

[Core i9 9](#_Toc83227271)

## Math Co-Processor

A **Co-Processor** is a processor that assists the main processor. One co-processor is the **Math Co-processor**, which performs mathematical operations. For Intel, the family of Math Co-processors is labelled as 80x87, which supports the 80x86 processors.

All the math co-processors have almost identical instruction sets and programming. They are able to:

* Add
* Subtract
* Multiply
* Divide
* Find square roots
* Calculate partial tangents
* Calculate partial arctangents
* Compute logarithms

Shifting this work to a separate microprocessor **reduces pressure** on the main microprocessor.

In a sense, the **Math Co-Processors** also create a **multi-core** processor environment, since there are now two separate processors working together.

## CISC and RISC Processors

Depending on their functionalities, we can divide microprocessors into two broad categories, Complex Instruction Set Computers (CISC) and Reduced Instruction Set Computers (RISC).

**CISC** was created to make **compiler development** simpler. It shifts most of the burden of generating machine instructions to the **processor**. For example, instead of making the compiler write long machine instructions to calculate a square root, a CISC processor would have a built-in function for this.

This is opposed to **RISC**, where we utilize a **small, highly optimized** set of instructions.

Typically, **multi-core** processors use **CISC**, since the instruction set required to communicate between two separate processors can get quite complicated. In contrast, **simple** microprocessors, such as those in microcontrollers, use **RISC**.

In summary,

|  |  |
| --- | --- |
| **CISC** | **RISC** |
| Extensive instructions | Reduced instruction set |
| Complex and efficient machine instructions | Less complex, simple instructions |
| Relatively few registers in use | Many symmetric registers (same size) in use |
| Extensive addressing capabilities for memory operations |  |
|  | Pipelining is used to allow for simultaneous execution of parts, or stages of instructions |

## Multi-Core Processors

From 1986 – 2002, microprocessor performance was increasing at an average of **50%** per yet. This increase was attributed to increasing **transistor density**, but this inherently had some problems. **Smaller transistors** gave us **faster processors**, but this used **more power**. The increased power consumption in turn caused **more heat**, which made the processors **unreliable**. These problems caused the rate of processor performance increases to drop to about **20%** per year.

To deal with this issue, instead of designing faster processors, it was proposed that we put **multiple processors** on a **single IC**. Thus, **multi-core processors** were born.

Note that the different processors are **synchronized**. They can have their own separate local memories, called a **loosely coupled system**, or a common, shared memory, called a **tightly coupled system**.

As mentioned before, technically speaking, the math co-processors also created a multi-core environment. The key difference is that with proper multi-core processors, all the processors had the **same capacity**, meaning they all performed equally well. This is not true for the math co-processors, which were weaker in comparison to the main processor.

### Parallel Programming

Having multiple processors available won’t help much if programmers don’t make use of them. **Serial programs** don’t benefit from multiple processors, so we need to move away from **single-core** (single CPU) systems and move towards **multi-core** (multiple CPU) systems by introducing **parallelism**.

Consider that we want to compute the expression . This computation can be done in three parts as **three separate tasks** if we are using a **single processor**. However, with **multiple processors**, each processor executes **one of the tasks**, which gets us the results faster.

One of the processors acts as a **master processor**, assigning the tasks to each of the processors. It also retrieves the **results** of each task and accumulates it. The assignment is done by the OS. In order to communicate between the microprocessors, some special instructions are needed.

Consider that we have the following piece of code:

int sum = 0;  
for (int i = 0; i < n; i++)  
{  
 x = compute\_next\_value();  
 sum += x;  
}

C++

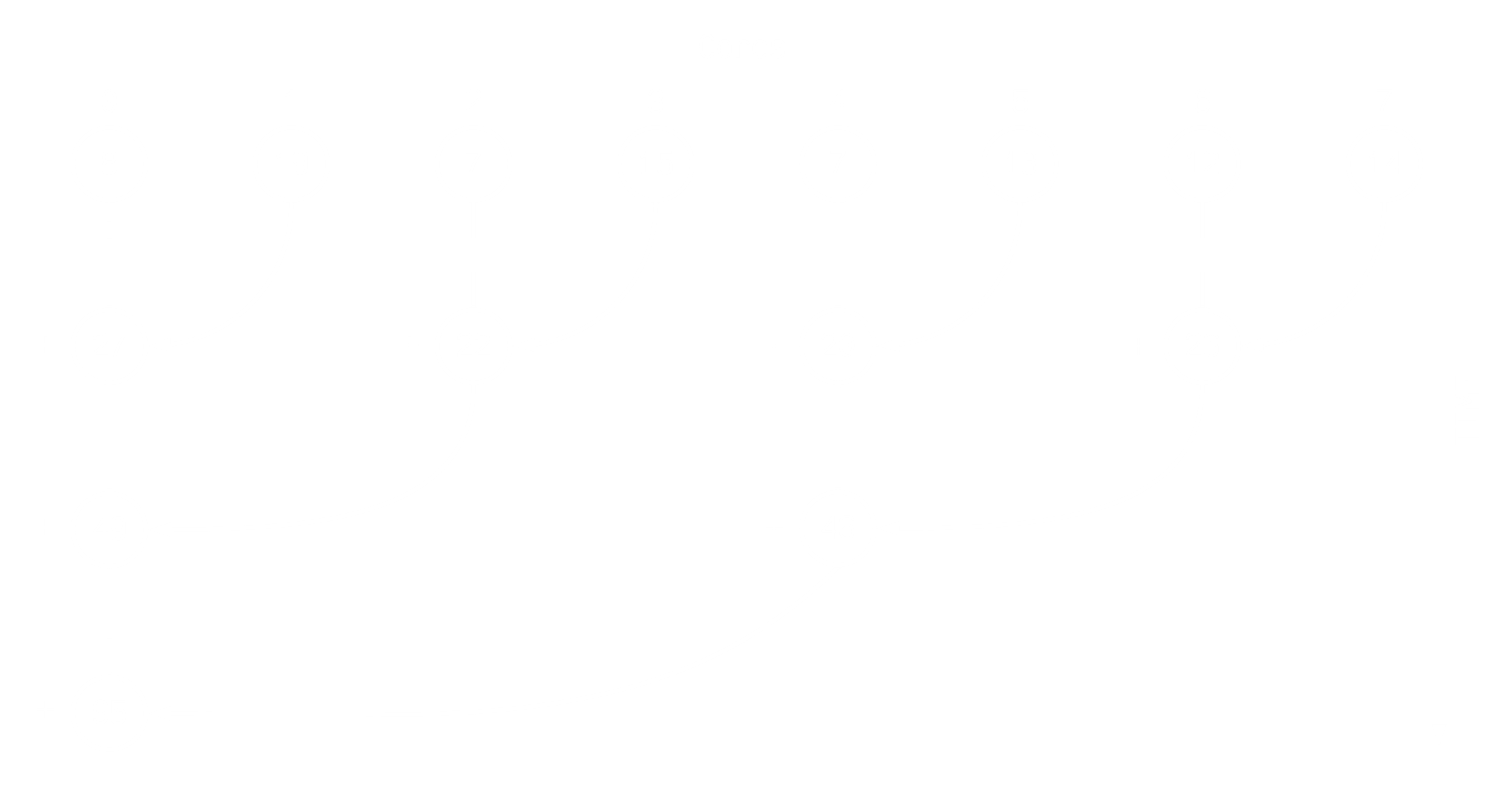
Here, we are computing values and adding them together. This is a **serial** approach and requires  **iterations to complete**.

Suppose instead that and we have  **cores** available to us. Thus, we can assign each of the cores the task of summing up just 3 values, which will require **3 iterations** in total (since the tasks are running parallelly). Finally, we just have to add up the results from each core in the master core, which is just a single loop. Thus, there are **10 iterations** in total.

if (I am the master core)  
{  
 sum = my\_x;  
 for each core other than myself  
 {  
 receive value from core;  
 sum += value  
 }  
}  
else  
{  
 send my\_x value to the master;  
}

PSEUDOCODE

There is however, an even better way to compute the global sum. We could tell **each core** to compute the **sum of three values**, and then tell **each other core** to compute the **sum of itself and the core just after it**. This could continue **step-by-step** until we have a **single core** remaining. The code for this is complicated, so this diagram should suffice.



After the initial 3 iterations, we needed just 3 more steps to get the global sum, so there are now **6 iterations** in total.

In the **first solution**, the **master core** received values from **7 other cores**, which meant **7 more additions**. In the **second solution**, it received values from **3 other cores**, which meant just **3 more additions**. This is an improvement factor of .

If we were working with  **cores**, in the **first solution**, the master core would receive values from other cores and perform  **more additions**, but in the **second solution**, it would receive values from just other cores and perform  **more additions**. This is an improvement factor of .

## Intel Processors

### Dual Core and Core 2 Duo

Both the **Dual Core** and the **Core 2 Duo** have **two cores**, with the prior being older and the latter being more modern and thus advanced. Dual Core has clock speeds of around 2.33 GHz while the Core 2 Duo has clock speeds of around 3.33 GHz.

### Core i3

The **Core i3** is an entry-level processor with **2 to 4 cores**. In addition, it has **4 threads**.

A **thread** of execution is the **smallest sequence** of programming instructions that can be managed independently by the scheduler, which is typically a part of the OS. **Multiple threads** existing within a process means each thread **executes concurrently** but **shares resources** like memory (which different processes do not share). The scheduler assigns these tasks to different cores in the required order. For example, for , the additions need to take place at the same time while the multiplication needs to take place afterwards.

The Core i3 also supports **hyper-threading**, which has efficient use of processor resources. It has a **3 to 4 MB cache** and uses **32 nm Silicon**, with the small size causing less heat and using less energy.

The Core i3 is suitable for basic tasks like word processing, email and web surfing. It even supports 64-bit versions of the Windows OS.

### Core i5

The **Core i5** is a mid-range processor with **2 to 4 cores** and **4 threads**. It supports **hyper-threading** and **turbo mode**, where a core that is not in use is turned off. It has a **3 to 8 MB cache** and uses **32 to 45 nm Silicon**.

The Core i5 is suitable for video editing and gaming in addition to basic tasks.

### Core i7

The **Core i7** is a high-end processor with **4 to 8 cores** and **8 threads**. It supports **hyper-threading** and **turbo mode**. It has a **4 to 8 MB cache** and uses **32 to 45 nm Silicon**.

### Core i9

The **Core i9** is a 10th and upper generation processor with **8 to 16 cores**. It supports **hyper-threading** and **turbo mode**. It has up to **16 MB** of **smart cache** and uses **1400 billion transistors**.