Performance & Architecture Types

CISC 3310 — CUNY Brooklyn College
Lecture Notes

Performance & Architecture Types

In this chapter, we will:

1. Learn parallel computing principles, and explain how they improve a computer's performance.
2. Talk about Flynn's taxonomy/architecture types.
3. Distinguish between RISC and CISC computer architectures.

Performance & Architecture Types

How can we further improve a computer? We might add more hardware to it in a way that makes a computer run its programs faster, for example.

A computer that consists of multiple **CPU cores** can run code in a parallel manner. This feature, called **parallel computing**, can be used to either run several programs at the same time or even run sections of the same program on several cores at the same time, which leads this program to execute its code faster.

In fact, we've already seen an example of parallel behavior when we discussed the ability to perform parts of the fetch-decode-execute cycle in a parallel manner (pipelining!) back in topic 5's slides 50-54.

Pipelining, however, is parallelization that happens inside one CPU core. The concept of parallel computing is when you have multiple cores (at least two cores) on which you can run programs or parts of programs in parallel, which can be done *in addition to* and *independently from* pipelining!

Performance & Architecture Types

In these slides, we will focus specifically on the ability to run different parts of a program on two or more cores, which reduces the overall execution time of the program.

If a program, for example, contains repeating chunks of code (such as those in loops) that could run independently from one another — that is, a loop's iteration doesn't depend on the result from any of the previous iterations — we could run these chunks, each on its own CPU core, in parallel.

We call the action of running a chunk of a program's code on a core **a thread** of execution. Every program you ever get to run has 1 or more threads of execution!

In our course's scope, we just introduce the general idea of threads, and now we'll explain how they contribute to a computer's performance improvement. If you want to learn a bit more about this topic, which is out of this course's scope, you are welcome to look into the lecture notes on the topic of threads, which were written for CISC 3320: Operating Systems course that you will take by the time you graduate and that will cover the topic of threads more extensively.

Parallel Computing: Performance

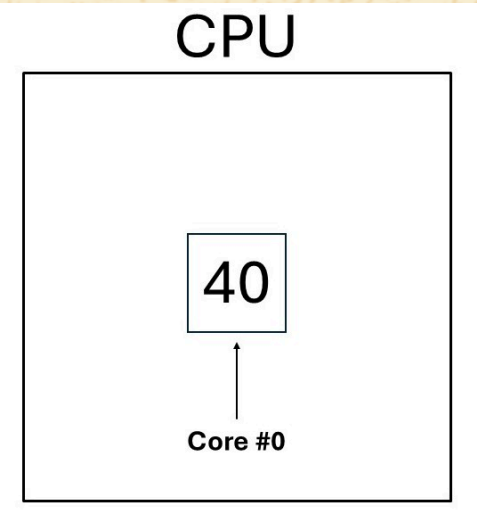
Let's bring an example: suppose that a Java or C++ program that you write contains a loop that needs to make 40 computations, all independent of one another. For example, your program has an array A of 40 integers, and you want to create an array B with integers that are as twice as large as those of A, so you need to multiply each of A's integers by 2, and then store it in B.

Because the doubling of each integer is independent from the doubling of any other integer (that is, you can multiply them separately,) we could run the code for multiplying each integer on separate CPU cores in parallel!

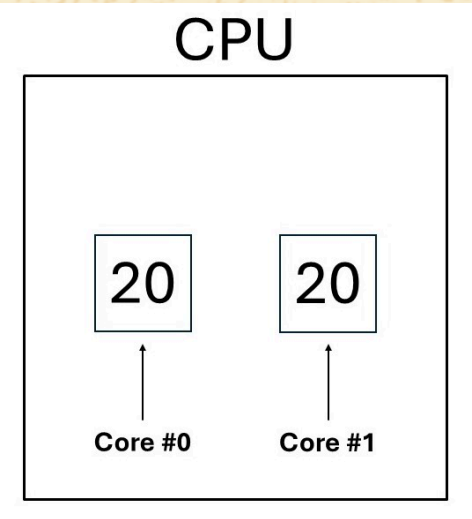
If our CPU only has 1 core, we have no choice but to run all of those computations on that single core. But if we have 2 cores, for instance, we could run 20 multiplications on one core, and the other 20 on the 2nd core. Similarly, if our CPU is a quad-core one, we can run 10 multiplications on each of the 4 cores. In other words, the more cores your computer has, the faster the program will get to complete its mission.

Parallel Computing: Performance

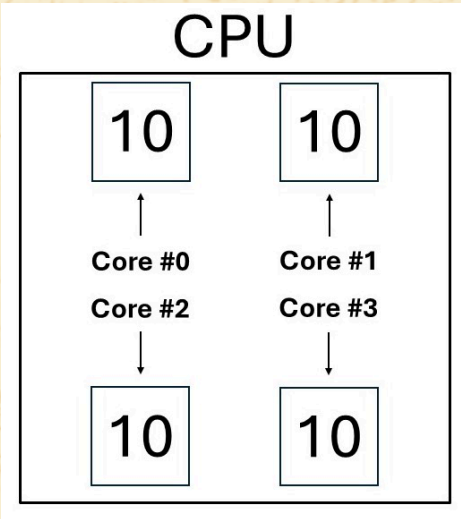
Suppose that the execution of this program on a single core takes 40 milliseconds. If we have 2 cores, we could split these 40 tasks among these two cores, as we explained on the previous slide, which will make the program run approximately 2 times faster (about 20 milliseconds). On a 4-core CPU, the program will run close to 4 times faster (about 10 milliseconds), etc.



Running all 40 multiplications on one core. Miriam Briskman, CC BY-NC 4.0.



Splitting the program between 2 cores. Miriam Briskman, CC BY-NC 4.0.



Splitting the program between 4 cores. Miriam Briskman, CC BY-NC 4.0.

Parallel Computing: Performance

From the idea on the previous slide, we understand that, if our computer's CPU has n cores, if we can split a program's code into threads that run in parallel on these n cores, the program will run *close to* n times faster.

Why *close to* n and not *exactly* n times faster? Because:

1. There are parts in every program that can't run in parallel, which means that only a portion of a program could be split over several cores. We have to run these parts only on one of the cores.
2. The computer needs to spend some extra time on the creation and management of each thread of execution on a core (because we need to prep the cores, copy some data around, etc.,) which reduces the overall speedup.

A pretty simple math formula that calculates the maximum possible speedup is called **Amdahl's Law**.

Parallel Computing: Amdahl's Law

The formula of Amdahl's Law is:

$$Speedup = \frac{1}{(1-p) + \frac{p}{n}}.$$

In the above equation, p is the portion of the program that runs in parallel (a ratio from 0 to 1,) and n is the number of available CPU cores in the device (integer greater than or equal to 1.)

That is, to calculate the maximum possible speedup, you need to know (1) what chunk of the program will be run by threads on multiple CPU cores, and (2) how many CPU cores the computer has. Then, just plug-in the quantities to find the actual speedup.

Parallel Computing: Amdahl's Law

An interesting fact that we learn from this formula is that the number of CPU cores in the device is a limiting factor: creating more and more threads won't make code run faster. To actually reach the maximum speedup, the program must create a number of threads that is equal to the number of existing CPU cores (not more and not less) that will run that section of code.

Example: If a computer has 4 CPU cores, and the portion of the program that is supposed to manifest parallelism is 30%, then the overall speedup of the whole program is:

$$Speedup = \frac{1}{(1-0.3)+\frac{0.3}{4}} = \frac{1}{0.7+0.075} = \frac{1}{0.775} = 1.2903,$$

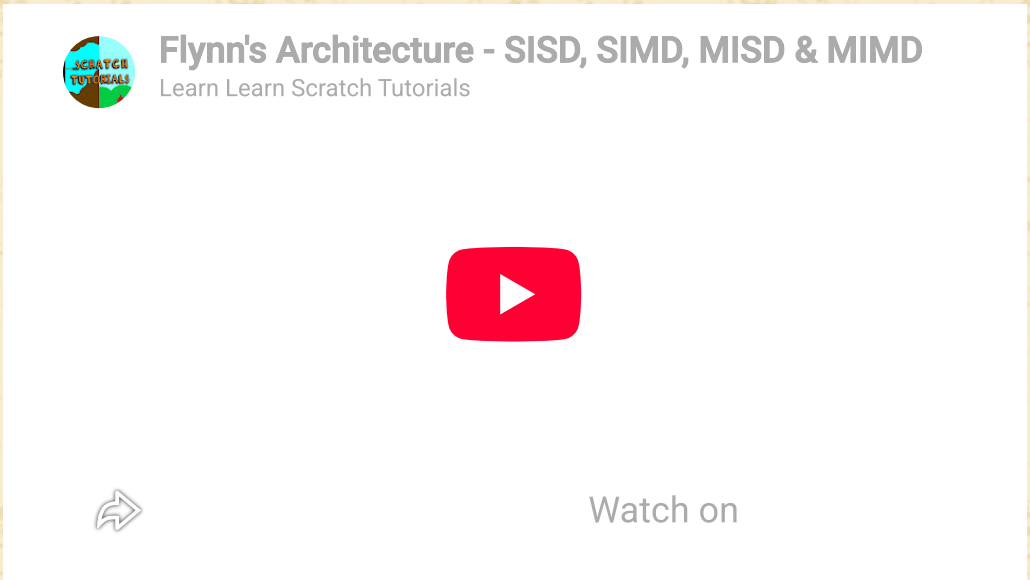
so the program will run about 1.3 times faster than if it were to run fully on a single core.

Parallel Computing: Amdahl's Law

Any questions?

Flynn's Taxonomy / Architectures

Flynn's Taxonomy divides all computers into 4 general types of computer architectures. Let's first watch the following video that introduces them:



https://www.youtube.com/watch?v=KVOc6369-Lo

Flynn's Taxonomy / Architectures

As the video on the previous slide says, **Flynn's taxonomy** is a classification system for computer architectures proposed by Michael J. Flynn in 1966.

It categorizes architectures based on the number of instruction streams and data streams they process; the four categories it defines are **SISD**, **SIMD**, **MISD**, and **MIMD**.

This taxonomy is widely used because it provides a clear framework for understanding parallel processing capabilities.

Although it is simplistic, it serves as a foundation for more modern and detailed classifications of computing systems.

Flynn's taxonomy has remained relevant since it highlights the fundamental ways processors handle instructions and data streams.

It is often taught in computer architecture courses as a starting point for learning about processing techniques.

Flynn's Taxonomy / Architectures

SISD stands for **Single Instruction, Single Data**, and it represents traditional sequential processing systems.

It is advantageous for simplicity and low cost but limited in performance since it cannot execute tasks in parallel.

SIMD, or **Single Instruction, Multiple Data**, involves executing one instruction across multiple data points simultaneously.

It is commonly used in applications like image processing and scientific simulations because it efficiently handles vectorized tasks.

While **SIMD** architectures are efficient, they require data to be in a specific format, which can limit their general-purpose applicability.

Modern **SIMD** architectures are seen in GPUs and specialized processors designed for parallel computation tasks. **SISD**, by contrast, is now primarily found in older or low-power devices like embedded systems and microcontrollers.

Flynn's Taxonomy / Architectures

MISD, or **Multiple Instruction, Single Data**, is a rare architecture involving multiple instructions acting on the same data. It is sometimes used in fault-tolerant systems like space shuttles, where redundancy is prioritized over speed or flexibility.

The **MIMD** category, or **Multiple Instruction, Multiple Data**, describes systems that can handle multiple tasks independently and in parallel. It is highly versatile and forms the basis for most modern multicore and multiprocessor systems, including servers and workstations.

While **MIMD** architectures offer flexibility and power, they are complex to program and require sophisticated task synchronization. They are widely used in high-performance computing environments for tasks like simulations and big data analysis.

Although **MISD** has niche applications, **MIMD** dominates in scenarios requiring diverse and concurrent processing.

Flynn's Taxonomy / Architectures

In short:

1. **SISD** architectures are simple and cost-effective but cannot handle the demands of modern, complex applications.
2. **SIMD** systems excel in parallel tasks with uniform operations, but they are constrained by the need for homogenous data.
3. **MISD** is suitable for highly specific and critical systems but is impractical for general-purpose computing.
4. **MIMD** architectures provide the greatest flexibility and performance but require more effort to optimize and maintain.

While **SISD** is largely outdated, **SIMD** and **MIMD** are critical to modern computing because they meet high-performance demands.

Understanding these architectures helps engineers select the right design for applications ranging from mobile devices to supercomputers.

Flynn's taxonomy remains a foundational tool for evaluating the trade-offs of different computing paradigms.

Flynn's Taxonomy / Architectures

Any questions?

RISC vs. CISC Architectures

RISC (**Reduced Instruction Set Computer**) is a computer architecture type using a small, highly optimized instructions.

CISC (**Complex Instruction Set Computer**,) on the other hand, includes a broader range of instructions, some of which can perform multiple operations in a single command.

The primary goal of **RISC** is to simplify the hardware by reducing the complexity of the instruction set and focusing on speed.

CISC architectures are designed to minimize the number of instructions per program by including more complex, higher-level commands.

While **RISC** prioritizes efficiency in executing simple instructions quickly, **CISC** aims to reduce the total number of instructions needed for a task.

Both architectures have been widely used in the computing industry, and each has unique advantages and trade-offs.

RISC vs. CISC Architectures

RISC systems support a smaller and more uniform set of instructions, which simplifies decoding and execution in the processor pipeline. Because of their simplified instruction set, **RISC** processors can execute instructions at a higher rate, often completing one per clock cycle.

CISC processors, while slower for individual instructions, reduce the number of instructions needed per program because of their complex commands.

Although **RISC** emphasizes hardware simplicity, it may require more software-level instructions, increasing the complexity of compilers.

CISC, on the other hand, shifts some complexity from software to hardware, which can lead to slower performance for simple tasks.

RISC vs. CISC Architectures

RISC processors are known for their high speed and efficiency because they use a fixed instruction length and simple addressing modes. They are easier to design and manufacture, but they can require more instructions to complete complex operations compared to **CISC**.

CISC architectures are better suited for programs requiring fewer instructions, but they are more challenging to optimize and can have slower execution speeds. The hardware complexity of **CISC** processors often increases power consumption and cost, which can limit their use in mobile devices.

RISC is ideal for applications requiring high performance and energy efficiency, while **CISC** is favored for backward compatibility with legacy systems.

Although **CISC** systems are less power-efficient, they can simplify programming and reduce memory usage for certain applications.

RISC vs. CISC Architectures

RISC architectures are widely used in modern devices such as smartphones, tablets, and embedded systems because of their efficiency. Examples of **RISC**-based processors include those in the **ARM** family, which power most mobile devices and IoT devices.

CISC architectures, such as **x86**, dominate in desktop computers, laptops, and servers because of their compatibility with a broad range of software. Many modern processors, including Intel and AMD CPUs, use **CISC** architectures but incorporate **RISC** principles for certain operations.

While **RISC** excels in energy efficiency, **CISC** provides better support for complex operating systems and older software environments.

Understanding the differences and applications of **RISC** and **CISC** helps in choosing the appropriate architecture for specific use cases.

RISC vs. CISC Architectures

RISC code typically uses simple 1-cycle instructions, each performing a single operation such as load, store, or add.

For example, adding two numbers stored in memory in a RISC system might require the following 4 instructions:

```
LOD R1, X
LOD R2, Y
ADD R3, R1
ADD R3, R2
```

in which we load the variables at the memory addresses `X` and `Y` into the CPU registers `R1` and `R2`, and then add these values and store them in register `R3`.

Note that it takes the CPU one clock cycle to process each of these instructions.

RISC vs. CISC Architectures

In contrast, CISC code often uses more complex instructions that can combine multiple operations into one.

For example, a single **CISC** instruction such as:

```
ADD X, Y
```

might load two numbers from memory, add them, and store the result back in memory (inside the variable whose address is `A`.) The computer will perform all of these actions as a consequence of calling just this one `ADD`.

Obviously, the instruction above will take multiple clock cycles to get completed.

While RISC code requires more lines, it allows for faster execution because each instruction is simpler and executes in a fixed time. On the other hand, CISC code reduces the number of instructions, but each instruction can take longer to decode and execute because of its complexity.

RISC vs. CISC Architectures

Converting CISC code to RISC code involves breaking down complex instructions into smaller, simpler instructions.

For example, the CISC instruction `INC counter`, which increments the value at memory location `counter` (that is: `counter++;`) could be converted into multiple RISC instructions.

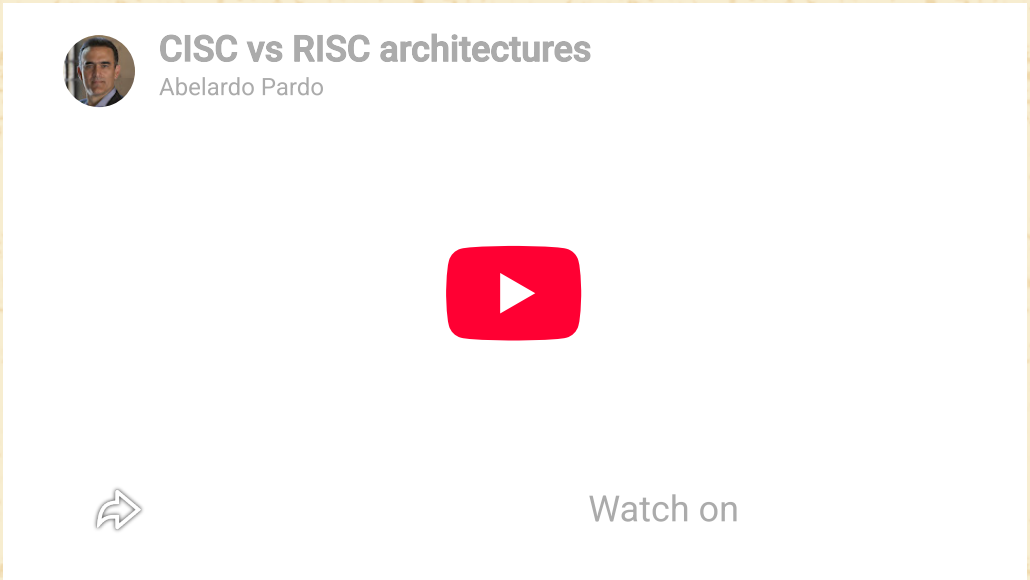
First, use a `LOD` (load) instruction to move the value into a register: `LOD R1, counter`. Next, perform the addition by one `ADD R1, 1` and store the result of the addition to register `R1`. Finally, store the result back into memory with a `STO` (store) instruction: `STO counter, R1`, where `counter` is the memory location of the original variable and the place where we'll store the result.

This approach demonstrates how RISC systems emphasize simplicity and modularity while maintaining compatibility with higher-level tasks.

If you get to use a CISC machine, you'll notice a great variation in the kinds of instructions it supports, e.g., addition of two registers (`ADD R1, R2`) addition of two memory variables (`ADD X, Y`), addition of a register and variable (`ADD X, R1`), etc.

RISC vs. CISC Architectures

A great video providing further insight into RISC vs. CISC:



https://www.youtube.com/watch?v=mDrUkjOVtAU

RISC vs. CISC Architectures

Any questions?

Performance & Architecture Types: Sources

Bergmann, Seth D. (2018). Section 9.3 and Section 9.4. *Computer Organization with MIPS*. License: CC BY-NC-ND: Attribution-NonCommercial-NoDerivs.

CS301: Computer Architecture. Unit 6.4, Unit 7.3, and Unit 8. *Saylor Academy*, 2010. License: CC BY: Attribution.

Murphy, Mike. System Architecture | The CPU | Instruction Set Architecture. *Coastal Carolina University*, 2023. License: CC BY-SA: Attribution-ShareAlike.

Viswanath, Divakar. Chapter 5. *Scientific Programming and Computer Architecture*. The MIT Press, 2017. License: CC BY-NC-ND: Attribution-NonCommercial-NoDerivs.

Wienand, Ian. (2019). Subsection 3.1.4. *Computer Science from the Bottom Up*. License: CC BY-SA: Attribution-ShareAlike.