Observe ... Same program written in C++, Go lang, and Python. Common thing is the asynchronous nature of the program.

In **C++, async functions and callbacks** can be used in combination with multithreading to achieve asynchronous and concurrent execution:

```
#include <iostream>
#include <thread>
#include <future>
// Async function that performs a lengthy computation
int lengthyComputation(int a, int b)
{std::this_thread::sleep_for(std::chrono::seconds(3));
// Simulating a time-consuming task
return a + b; }
// Callback function to be executed when the async computation is complete
void callback(int result)
{ std::cout << "The result is: " << result << std::endl; }
int main() {
// Launching the lengthy computation asynchronously
std::future<int> futureResult = std::async(std::launch::async,lengthyComputation, 5, 7);
// Do some other work concurrently while waiting for the result
// Registering a callback function to be executed when the async computation is complete
auto callbackTask = std::async(std::launch::async, [](std::future<int>& future) { int result =
future.get();
// Waiting for the future to complete and obtain the result
callback(result);
// Calling the callback function
},
std::ref(futureResult));
// Do some other work concurrently
// Wait for the callback task to complete
callbackTask.wait();
return 0;
}
```

In the above example, we have a lengthyComputation function that performs a time-consuming computation and returns the result. We want to execute this function asynchronously to avoid blocking the main thread. Using std::async, we launch the lengthyComputation function asynchronously and obtain a std::future<int> object, futureResult, that represents the result of the computation. While waiting for the result, we can perform other tasks concurrently. In this example, we register a callback function, callback, using another std::async call. This callback function will be executed when the async computation is complete. The callback task captures the futureResult by reference and waits for it to be ready by calling future.get(). Once the result is obtained, the callback function is called with the result as an argument. Finally, we wait for the callback task to complete using callbackTask.wait() to ensure that the program doesn't exit before the callback execution finishes. This example demonstrates the use of async functions, callbacks, and multithreading to achieve concurrent execution and asynchronous behavior in C++. The main thread can continue with other tasks while the lengthy computation is being performed, and the callback is executed when the computation is complete.

In Go, async functions and callbacks are achieved through goroutines and channels. Goroutines are lightweight threads managed by the Go runtime, and channels are used for communication and synchronization between goroutines.

```
package main
import ( "fmt" "time" )
// Asynchronous function that performs a lengthy computation
func lengthyComputation(a, b int, result chan<- int) {
time.Sleep(3 * time.Second)
// Simulating a time-consuming task
result <- a + b }
// Callback function to be executed when the async computation is complete
func callback(result int) {
fmt.Println("The result is:", result) }
func main() {
// Creating a channel to receive the result
resultChan := make(chan int)
```

```
// Launching the lengthy computation asynchronously as a goroutine
go lengthyComputation(5, 7, resultChan)
// Do some other work concurrently while waiting for the result
// Receiving the result from the channel (blocking operation)
result := <-resultChan
// Calling the callback function with the result
callback(result)
// Do some other work concurrently // Wait for a key press to exit
fmt.Scanln()
}
```

In the above example, we have a lengthyComputation function that performs a time-consuming computation and sends the result through a channel (resultChan) to the caller. Using go, we launch the lengthyComputation function as a goroutine, passing the arguments and the result channel. The goroutine executes concurrently with the main goroutine. While waiting for the result, we can perform other tasks concurrently. In this example, we directly receive the result from the resultChan using the expression result := <-resultChan. This is a blocking operation that waits until a value is sent on the channel. Once the result is received, we call the callback function with the result as an argument. The program continues executing other tasks concurrently, and in this example, we wait for a key press to exit using fmt.Scanln(). This example demonstrates the use of goroutines, channels, and callbacks in Go to achieve asynchronous behavior and concurrent execution. The goroutine allows the lengthyComputation function to execute concurrently, and the channel enables the communication between goroutines. The callback function is invoked when the computation is complete and the result is received from the channel.

In Python, async functions and callbacks can be implemented using asynchronous programming techniques such as the `asyncio` module and `async`/`await` keywords. **Although Python's multithreading capabilities are limited due to the Global Interpreter Lock (GIL)**, we can still achieve concurrency by utilizing the `asyncio` module.

```
import asyncio
# Asynchronous function that performs a lengthy computation async
```

```
def lengthy_computation(a, b):
await asyncio.sleep(3)
# Simulating a time-consuming task
return a + b
# Callback function to be executed when the async computation is complete
def callback(result):
print("The result is:", result)
async def main():
# Launching the lengthy computation asynchronously
task = asyncio.create_task(lengthy_computation(5, 7))
# Do some other work concurrently while waiting for the result
# Registering a callback function to be executed when the async computation is complete
result = await task callback(result)
# Do some other work concurrently
# Run the main coroutine
asyncio.run(main())
```

In the above example, we define an async function called lengthy_computation that performs a time-consuming computation and returns the result. We use the asyncio.sleep() function to simulate the delay. The callback function is a regular function that will be executed when the async computation is complete. It simply prints the result. The main function is an async function that serves as the entry point. We launch the lengthy computation asynchronously using asyncio.create_task(), which returns a task object representing the computation. While waiting for the result, we can perform other tasks concurrently. In this example, we await the completion of the task using await task. This pauses the execution of the main function until the task is complete, allowing other tasks to proceed concurrently. Once the result is obtained, we call the callback function with the result. Finally, we run the main coroutine using asyncio.run() to initiate the event loop and execute the asynchronous code. This example demonstrates how to use async functions, callbacks, and the asyncio module in Python to achieve asynchronous behavior and concurrent execution. The asyncio module manages the event loop and allows us to await the completion of tasks, enabling non-blocking and concurrent execution.

Using go lang for logic example: The program uses concurrency and parallelism to handle tasks more efficiently. The various logical constructs used in the program are as follows:

1. **Concurrency Logic:** This is where Go's goroutines and channels come into play.
   a. Goroutines: The go lengthyComputation(5, 7, resultChan) statement in the main() function starts a new goroutine, which is a lightweight thread managed by the Go runtime. This is an example of concurrent execution in Go.
   b. Channels: The resultChan := make(chan int) and result <- a + b statements illustrate the use of channels for communication between goroutines. In this case, the lengthyComputation() function sends the result of a computation to the main() function via the resultChan channel.

2. **Asynchronous Logic:** The function lengthyComputation(a, b int, result chan<- int) is run as a goroutine, indicating that it is intended to be run asynchronously. It will execute in parallel with the main thread, without blocking its execution. The channel communication result <- a + b also hints at an asynchronous operation, where the result is sent to the channel and retrieved asynchronously.
3. **Synchronous Logic:** The statement result := <-resultChan is a blocking operation, meaning it will stop execution of the main() function until data is received on the resultChan channel. This is an example of synchronous behavior in Go.
4. **Functional Logic:** The callback(result) in the main() function is an example of functional programming style in Go. The callback function is a higher-order function that gets executed after the async computation is complete.
5. **I/O Logic:** The functions fmt.Println("The result is:", result) and fmt.Scanln() are examples of input/output operations. The fmt.Println() function is used to print data to the standard output, while fmt.Scanln() is used to read input from the standard input.

Computer instructions are created by combining logic gates in various ways. The most basic logic gates are the AND gate, the OR gate, the NOT gate, and the XOR gate. These gates can be combined to create more complex logic circuits that can perform a variety of tasks. For example, an AND gate can be used to check if two conditions are both true, while an OR gate can be used to check if either of two conditions is true. A NOT gate can be used to invert the truth value of a signal, and an XOR gate can be used to check if two signals are different. The patterns in which

logic gates are assembled are called logic circuits. Logic circuits can be used to perform a variety of tasks, such as addition, subtraction, multiplication, division, and logical operations such as AND, OR, and NOT. Logic circuits are also used to control the flow of data in a computer system. There are many different types of logic circuits, each with its own specific purpose. Some common types of logic circuits include:

- **Adders:** Adders are used to add two or more numbers together.
- **Subtractors:** Subtractors are used to subtract two or more numbers from each other. **Multiplexers:** Multiplexers are used to select one of several input signals to send to an output.
- **Demultiplexers:** Demultiplexers are used to distribute one input signal to several output signals.
- **Comparators:** Comparators are used to compare two numbers and determine which is greater.
- **Sequencers:** Sequencers are used to generate a series of signals in a specific order. Logic circuits are essential for the operation of any computer system. They are used to perform the basic operations that allow computers to compute and control.

Some high-level circuits used in computers are:

- **Combinational logic:** Combinational logic is a type of logic circuit that performs a specific operation on the input signals, regardless of the previous state of the circuit.
- **Sequential logic:** Sequential logic is a type of logic circuit that stores the previous state of the circuit and uses it to determine the current output.
- **Finite state machine:** A finite state machine is a sequential logic circuit that has a finite number of states. The state of the machine determines the output signals.
- **Microcontroller:** A microcontroller is a small computer that is embedded in a larger system. Microcontrollers are often used to control devices such as appliances, automobiles, and industrial machinery.

As per ISA (Instruction Set Architecture) there are primarily 2 types of processors that have succeeded over other architecture types - namely CISC and RISC-based architectures. Many microprocessor companies either use one, or both, or add their proprietary accelerator circuits to their chipsets. M1 and M2 from Apple are a good example of this. ARM's deviations from RISC-

V is another example. But overall, to understand the concepts, understanding Instruction Set Architecture helps.

Some essential logic circuits used in CISC-based processors (Intel, AMD, etc).

Arithmetic logic units (ALUs): ALUs are the most complex logic circuits in a processor. They are responsible for performing all of the arithmetic and logical operations that are required to execute instructions. ALUs are typically made up of a large number of logic gates, and they can be very complex.

Register files: Register files are used to store data and instructions. They are typically made up of a number of registers, each of which can store a single word of data. Register files are very fast, and they are used to store data that is being used by the ALU or the control unit.

Control units: Control units are responsible for sequencing the execution of instructions. They do this by fetching instructions from memory, decoding the instructions, and then sending signals to the other parts of the processor to execute the instructions. Control units are typically made up of a number of logic gates, and they can be very complex.

Memory units: Memory units are used to store data and instructions. They are typically made up of a number of memory cells, each of which can store a single bit of data. Memory units are much slower than register files, but they can store much more data.

Input/output units: Input/output units are used to transfer data between the processor and other devices. They are typically made up of a number of logic gates, and they can be very complex. **Some essential logic circuits used in RISC-based processors (ARM-based mobile processors, ARC, PowerPC, Apple etc).**

Arithmetic logic units (ALUs): ALUs are the most complex logic circuits in a processor. They are responsible for performing all of the arithmetic and logical operations that are required to execute instructions. ALUs are typically made up of a large number of logic gates, and they can be very complex.

Register files: Register files are used to store data and instructions. They are typically made up of a number of registers, each of which can store a single word of data. Register files are very fast, and they are used to store data that is being used by the ALU or the control unit. Instruction

decoders: Instruction decoders are used to decode instructions into a series of signals that can be executed by the processor. Instruction decoders are typically made up of a number of logic gates, and they can be very complex.

Data hazards units: Data hazards units are used to detect and resolve data hazards, which are situations where two instructions try to access the same data at the same time. Data hazard units are typically made up of a number of logic gates, and they can be very complex.

Branch prediction units: Branch prediction units are used to predict whether a branch instruction will be taken or not. This allows the processor to start executing instructions after the branch instruction, even if the branch is not taken. Branch prediction units are typically made up of a number of logic gates, and they can be very complex.

Caches: Caches are used to store frequently used data in fast memory. This can improve the performance of the processor by reducing the number of times it has to access slower memory. Caches are typically made up of a number of memory cells, and they can be very complex.

The Apple M1 and M2 chips are RISC-based processors, but they have a number of design features that make them different from other RISC processors. The M1/M2 chip uses a custom instruction set architecture (ISA). This means that the M1 chip can execute instructions that are not available on other RISC processors. The M1/M2 chip has a number of specialized accelerators. These accelerators can perform tasks such as graphics processing, machine learning, and cryptography much faster than a general-purpose processor. The M1/M2 chip is manufactured using a 5-nanometer process. This is the smallest manufacturing process used for any commercial processor, and it allows the M1/M2 chip to be more energy efficient than other RISC processors. As a result of these design features, the Apple M1/M2 chip is able to deliver significantly better performance than other RISC processors. For example, the M1 chip can deliver up to 3.5 times faster CPU performance and up to 5 times faster graphics performance.