**MIPS Simulator Report**

* Nathan Mathew Verghese IMT2022022
* Aaditya Gole IMT2022087

Ours is a Python program that simulates the execution of a series of MIPS assembly instructions using a simplified pipeline architecture. It reads and processes instructions from two input files, "FactorialCode.txt" and "SortingCode.txt." The code defines a dictionary for MIPS opcodes and registers, and emulates a simplified MIPS pipeline with basic instruction fetch, decode, execute, memory access, and write-back stages.

**1)Non- Pipeline**

Code Structure:

The code takes input from the files present in the folder and decodes it from the following functions.

1. Opcode and Register Definitions:

- `mipsOpcodes`: A dictionary containing binary opcode values for various MIPS instructions.

- `mipsRegisters`: A list mapping binary register numbers to their corresponding MIPS register names.

- `regMemory`: A dictionary representing registers and their initial values.

- `controlSignals`: A dictionary storing control signals used during instruction execution.

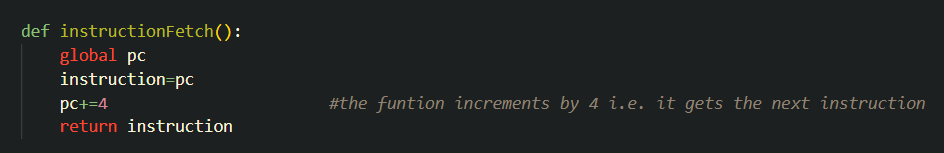
- `memAddress`: A dictionary representing the data memory.

2. Functions:

- `clock()`: Increments a global clock variable to advance one cycle.

- `resetSignals()`: Resets control signals to their initial state.

- `instructionFetch()`: Retrieves the next instruction and increments the program counter (pc).



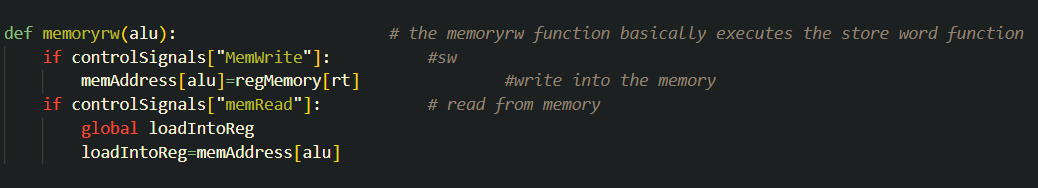
- `instructionDecode`: Decodes the instruction and sets control signals accordingly.



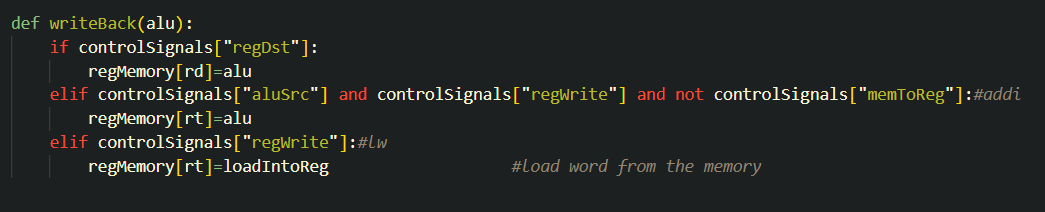
- `execute()`: Executes the instruction using the ALU and returns the result.



- `memoryrw`: Performs memory read and write operations.



- `writeBack`: Writes back the result of an instruction to registers.



3. File Input:

- The code reads MIPS assembly instructions from two input files: "FactorialCode.txt" and "fibo.txt." It stores the instructions in the `code` list.

4. Pipeline Execution:

- The program emulates a simple MIPS pipeline, fetching, decoding, executing, accessing memory, and writing back instructions.

- It repeatedly fetches, decodes, and executes instructions in a loop.

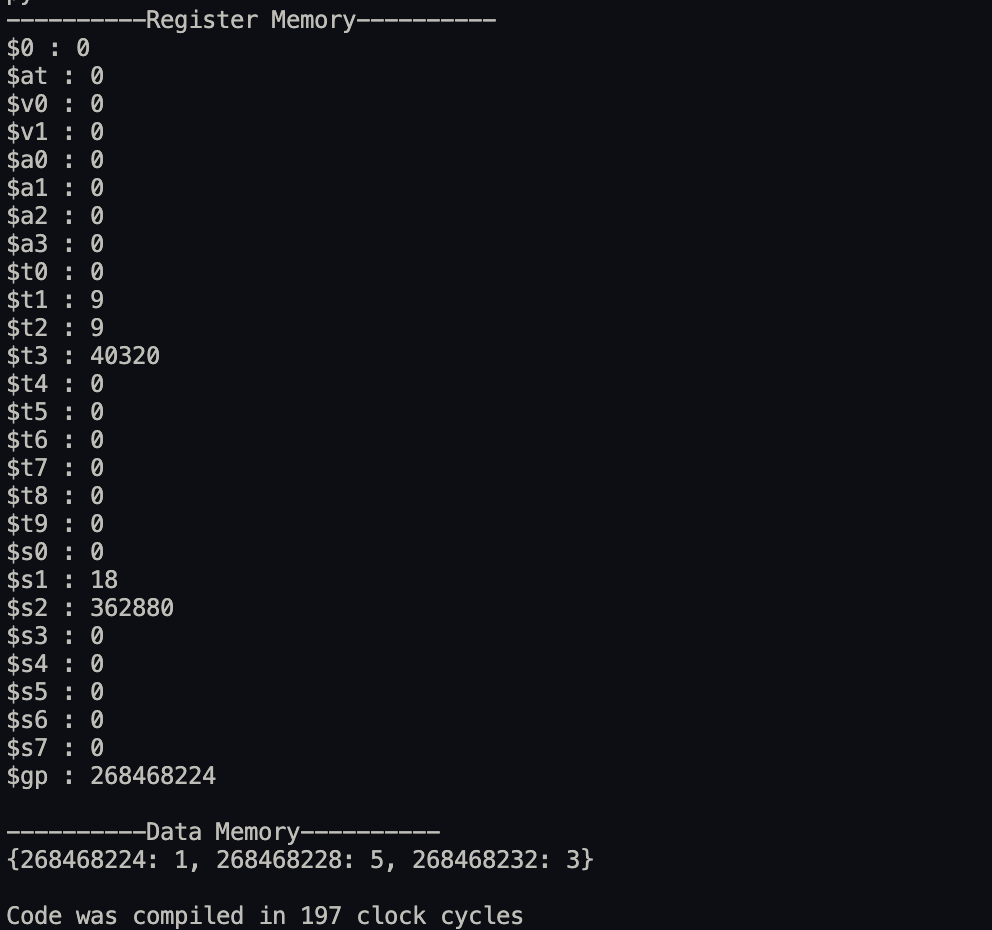
- The instruction sequence is determined by the instructions in the input files.

5. Output:

- The code prints the content of register memory and data memory after execution.

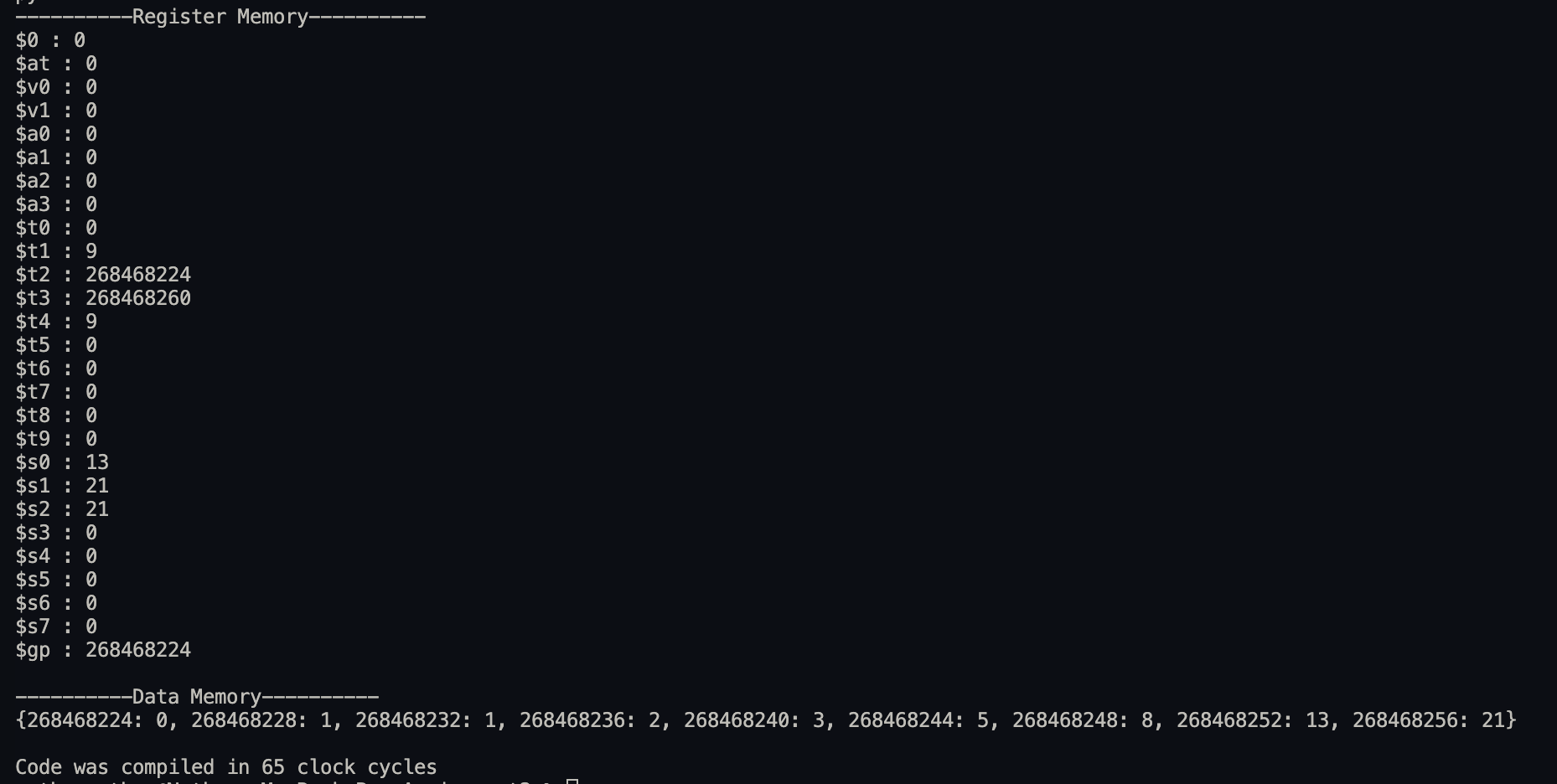
- It also displays the number of clock cycles required to execute the instructions.

**FactorialCode.txt**



The $s2 register has an output of 9! which is 362880.

**fibo.txt**



The output of data memory has the first 9 Fibonacci sequence numbers.

Conclusion:

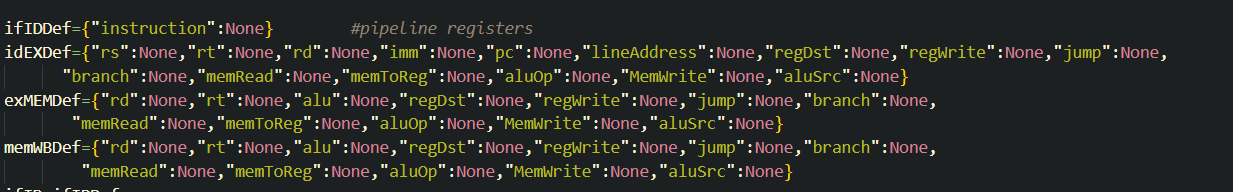
This code simulates the execution of MIPS assembly instructions using a simplified pipeline architecture. It reads instructions from input files, processes them through various pipeline stages, and tracks the state of the processor's register memory and data memory. The provided output shows the final state of these memory locations and the total number of clock cycles required to complete the execution.

**2)Pipeline**

**MIPS Simulator Pipeline**

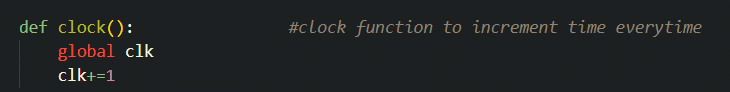
Our Python program simulates the execution of MIPS assembly instructions using a simplified pipeline architecture. The code is structured into various sections and functions, each serving a specific purpose. The following is a detailed report explaining the functionality of the code:

The code takes input from the files present in the folder and decoding it from the following functions.

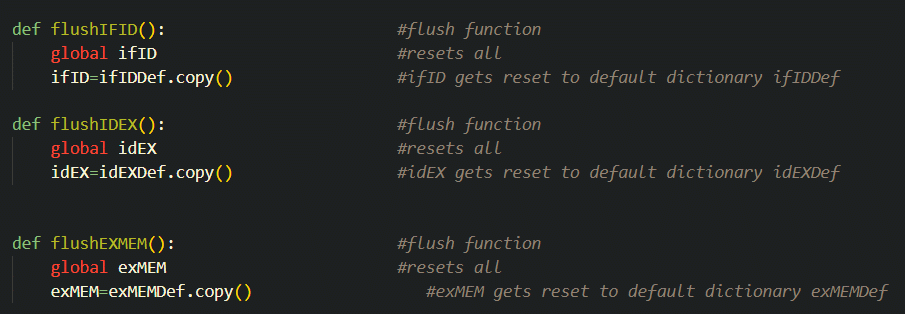
1. **MIPS Opcodes and Registers:**
   * mipsOpcodes: A dictionary that maps binary values to MIPS opcode mnemonics, such as "add," "sub," "addi," and more.
   * mipsRegisters: A list that pairs binary register numbers with their corresponding MIPS register names, like "$t0," "$t1," "$s0," and so on.
2. **Register Memory and Data Memory Initialization:**
   * regMemory: A dictionary that initializes the values of MIPS registers. The special register "$gp" is set to 0x10008000.
   * memAddress: A dictionary representing data memory with a few initial values.
3. **Pipeline Registers:**
   * The code defines four pipeline registers: ifID, idEX, exMEM, and memWB, each initialized with default values. These registers hold information at different stages of instruction execution.
   * 

Untitled

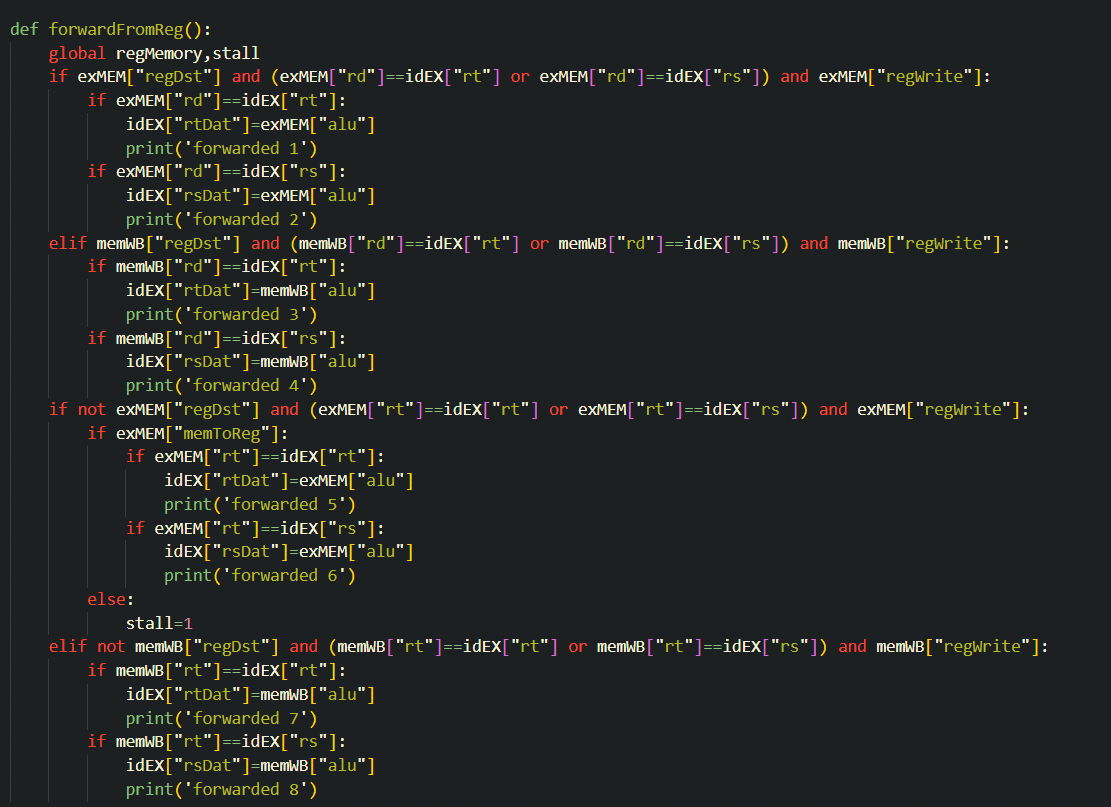
1. **Clock Function (clock):**
   * This function increments a global clock variable (clk) to simulate the passage of time. It is used to count clock cycles.

Untitled

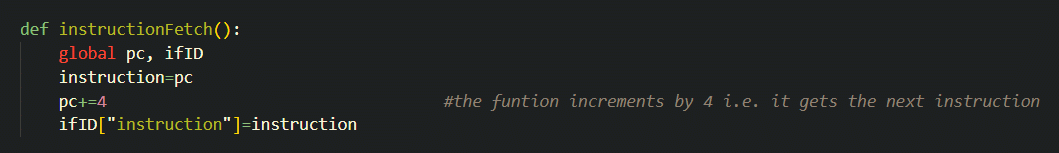
1. **Flush Functions (flushIFID, flushIDEX, flushEXMEM):**
   * These functions reset the content of the corresponding pipeline registers to their default values.

Untitled

1. **Forwarding Logic (forwardFromReg):**
   * This function handles forwarding logic to resolve data hazards. It checks and forwards data between pipeline stages when necessary.

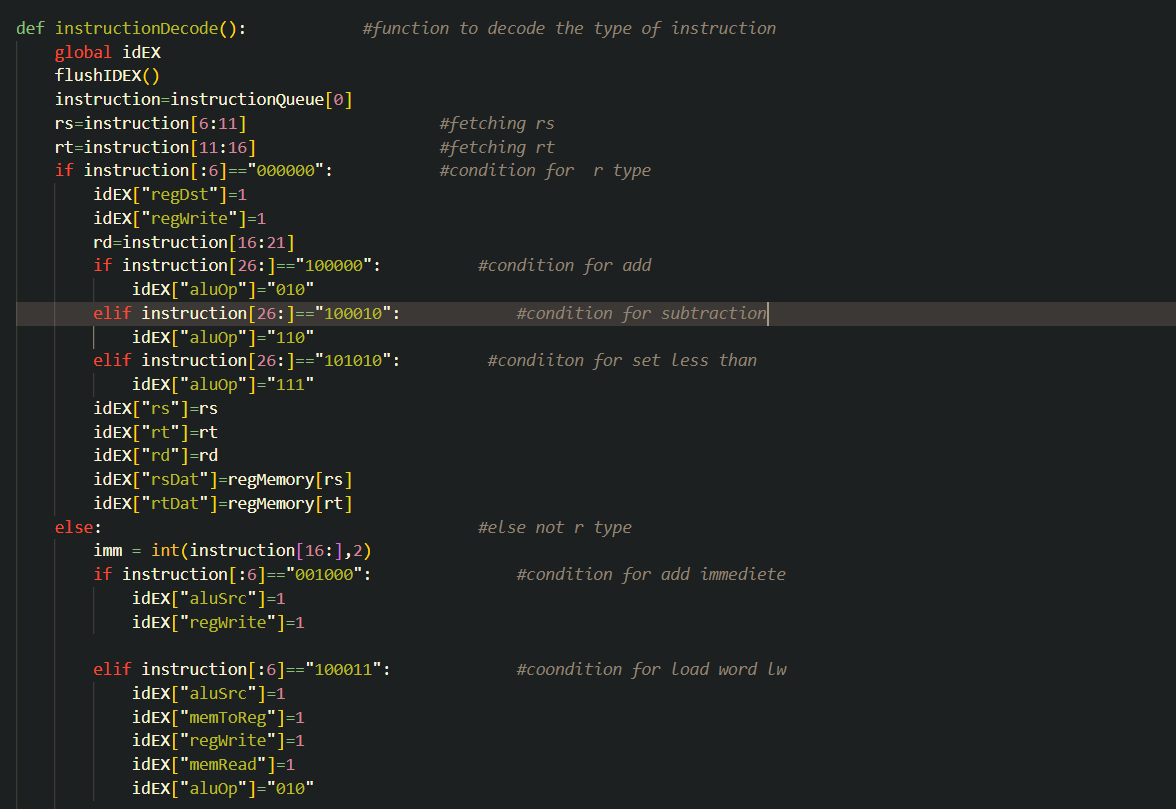


Untitled

1. **Instruction Fetch (instructionFetch):**
   * This function fetches the next instruction by incrementing the program counter (pc) and stores it in the ifID register.
   * 

Untitled

1. **Instruction Decode (instructionDecode):**
   * This function decodes the type of instruction and sets control signals accordingly. It extracts opcode, source registers, immediate values, and more.

Untitled

Untitled

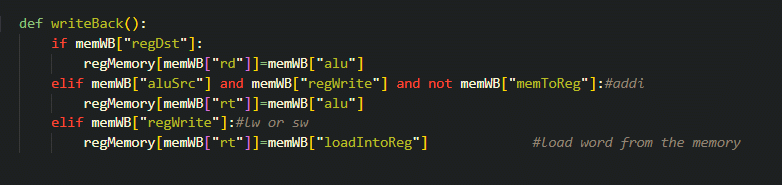
1. **Execute (execute):**
   * This function executes the instruction based on its type (R-type, I-type, or J-type). It performs arithmetic operations, checks for branching conditions, and handles jumps.
   * 

Untitled

1. **Memory Read/Write (memoryrw):**
   * This function simulates memory operations, including storing and reading data from memory. It updates the memWB register with relevant information.
   * 

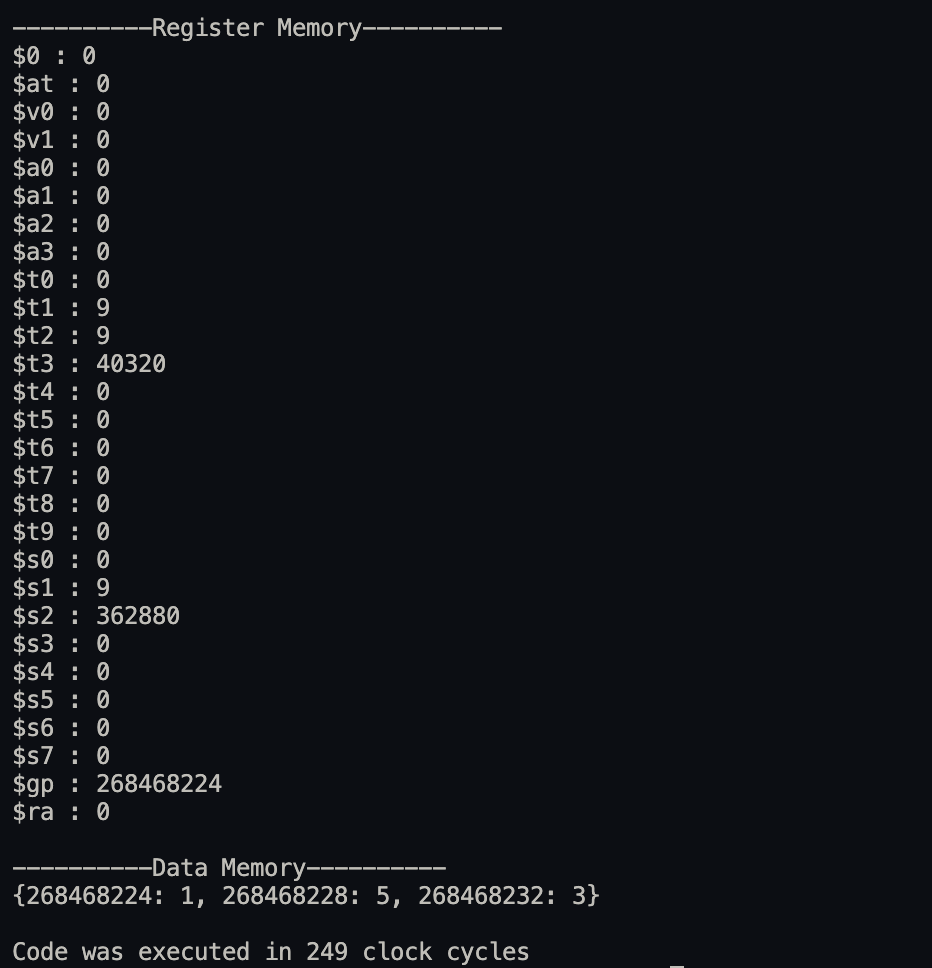
Untitled

1. **Write Back (writeBack):**
   * This function writes the result of an instruction back to the appropriate register in the
   * regMemory.

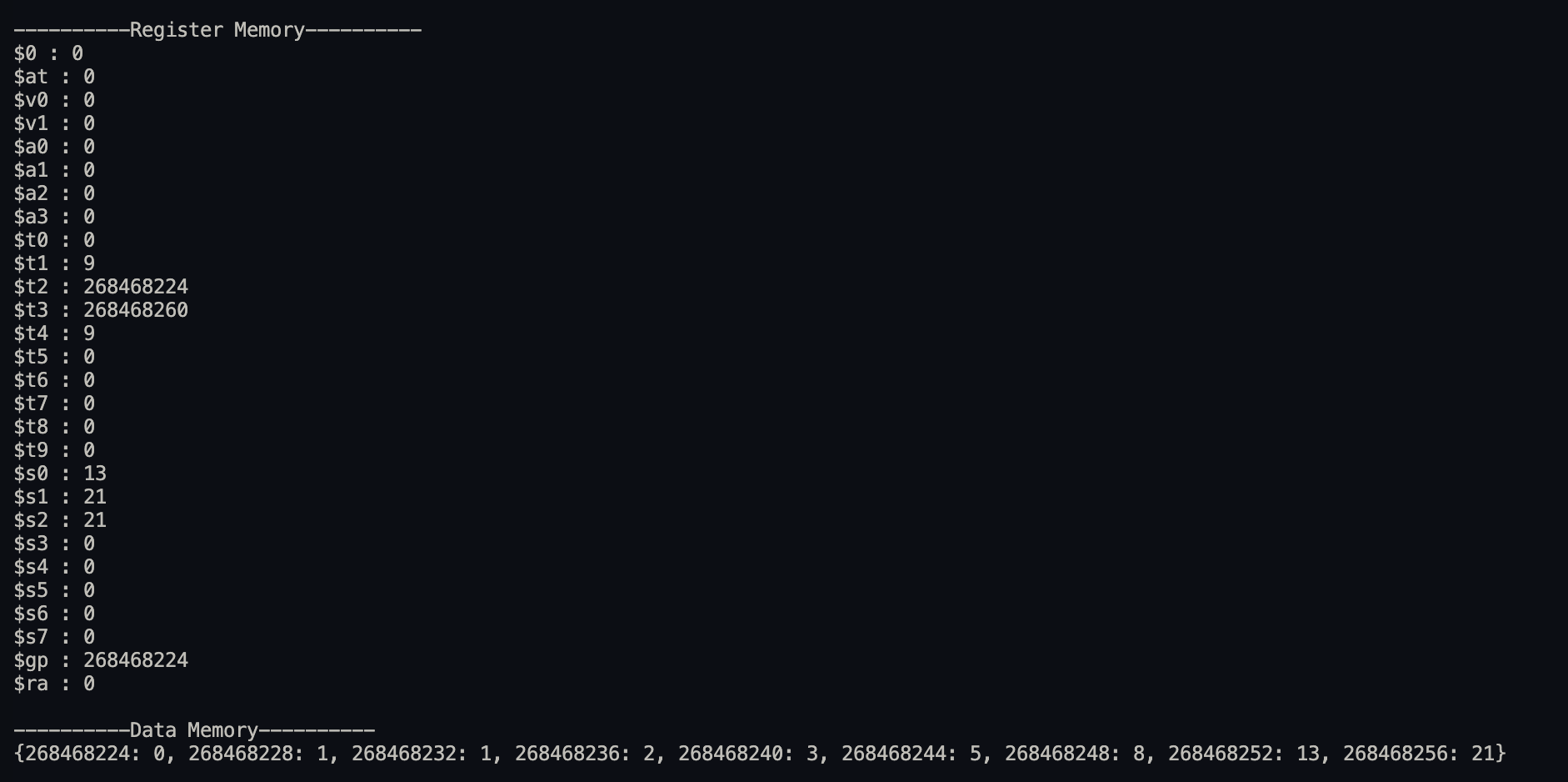
Untitled

1. **File Input and Code List:**
   * The code reads MIPS assembly instructions from input files ("FactorialCode.txt" and "fibo.txt") and stores them in the code list.
2. **Pipeline Execution:**
   * The code simulates a simplified MIPS pipeline by repeatedly fetching, decoding, executing, accessing memory, and writing back instructions.
   * It uses five-stage instruction queues to manage the flow of instructions through the pipeline.
3. **Clock Cycle Management:**
   * Clock cycles are counted and printed to track the execution progress.
   * The code handles stalls and branching by flushing the pipeline stages when necessary.
4. **Output:**
   * After execution, the code prints the final state of the register memory and data memory, along with the total number of clock cycles.

**FactorialCode.txt**



**fibo.txt**



* The code provides a simulation of a MIPS pipeline, managing instruction execution, data forwarding, and hazard handling. It demonstrates the functionality of various stages in a simplified processor architecture.