National Telecommunication Institute

8-bit MIPS-like Microprocessor

Using VHDL

Made by:

Ahmed Tawfik AbdAllah

Abdelrahman Ahmed Esmat

Under supervision of : Dr/ Mohamed Youssef

Group Number:

S25-B6-FPGA-G5-E

1. Summary

This project involves the design and implementation of a simple **8-bit microprocessor** using **VHDL**, integrating fundamental modules such as the Program Counter (PC), Instruction Register (IR), Controller, Register File, Arithmetic Logic Unit (ALU), and Data Memory.

The microprocessor executes a predefined instruction set, performing arithmetic (ADD, SUB, MUL, DIV) and logic (AND, OR, NAND, XOR) operations, with status flags for zero, parity, and overflow detection. The **fetch–decode–execute** cycle is coordinated by the controller, which manages data transfer between the ALU, registers, and memory based on the opcode.

Designed as a modular system, each component handles a specific role: the PC generates instruction addresses, the IR stores the current instruction, the Controller decodes it, the Register File stores operands, the ALU executes operations, and Data Memory handles data storage.

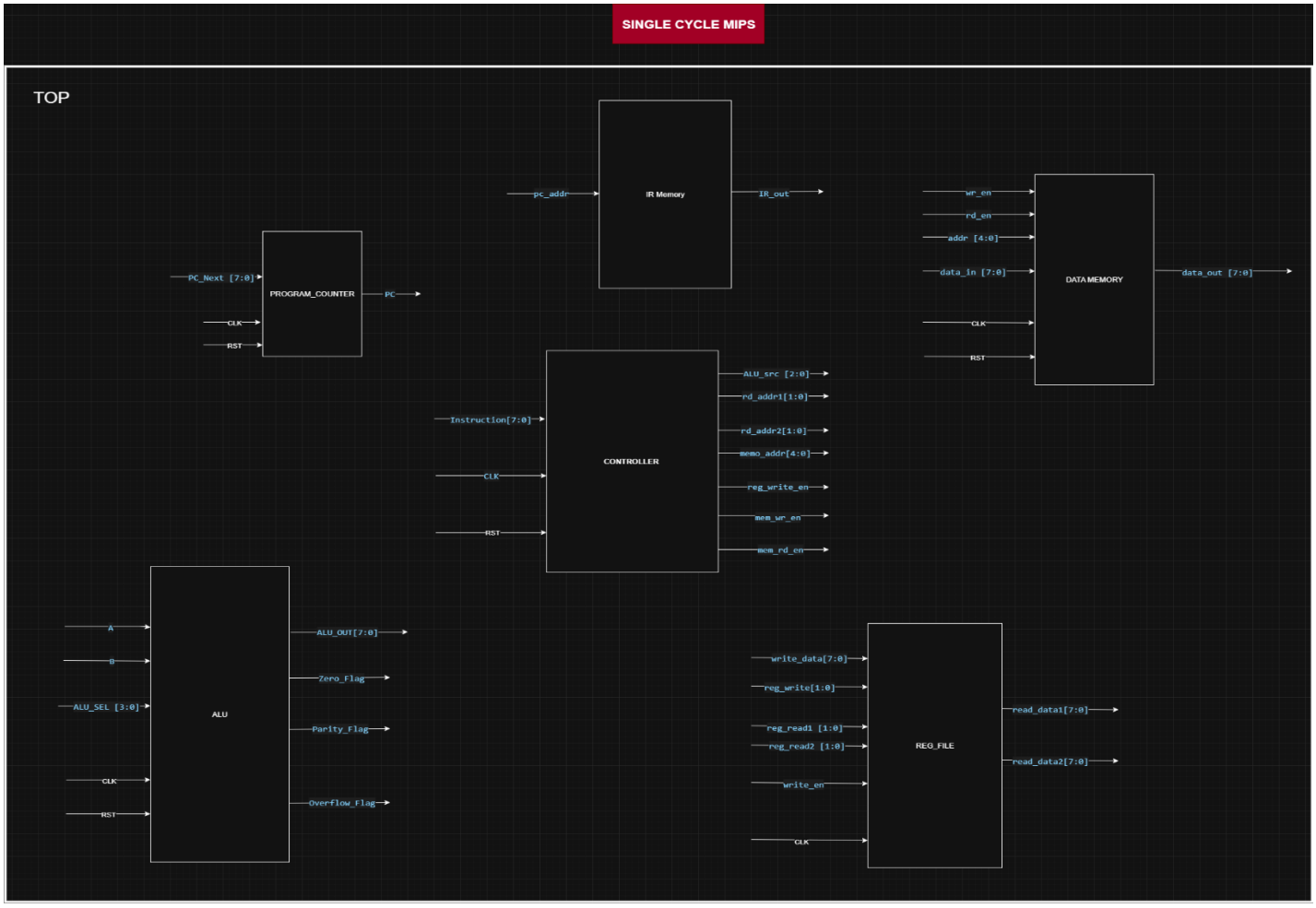
This project demonstrates core principles of **digital design** and **microprocessor architecture**, providing a foundation for future enhancements such as a wider instruction set or pipelining for improved performance.

2. System Overview

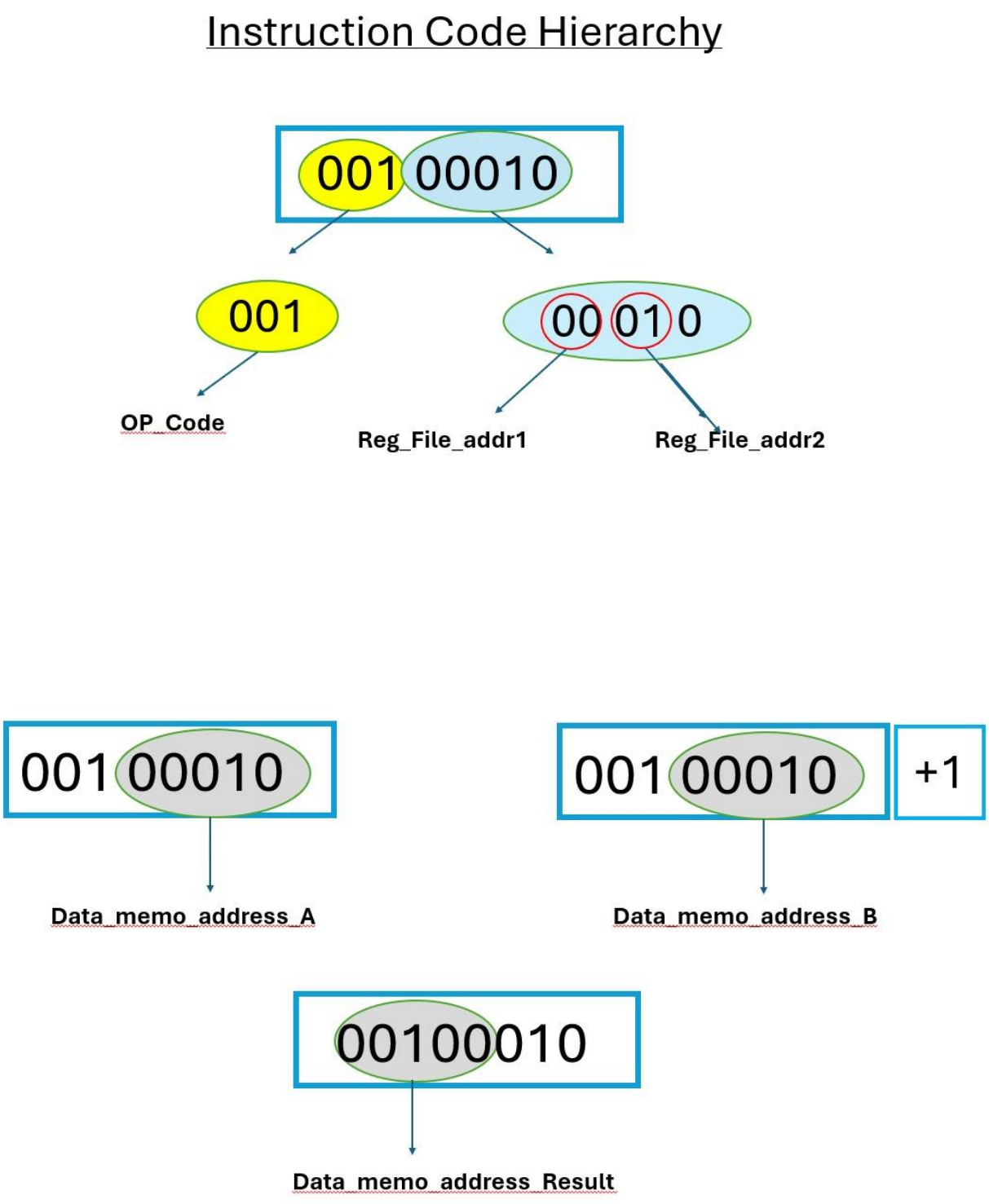
The designed 8-bit microprocessor follows a simple **fetch–decode–execute** cycle. It consists of interconnected modules that collectively fetch instructions from memory, decode them, execute the required operations, and store the results.

The main modules are:

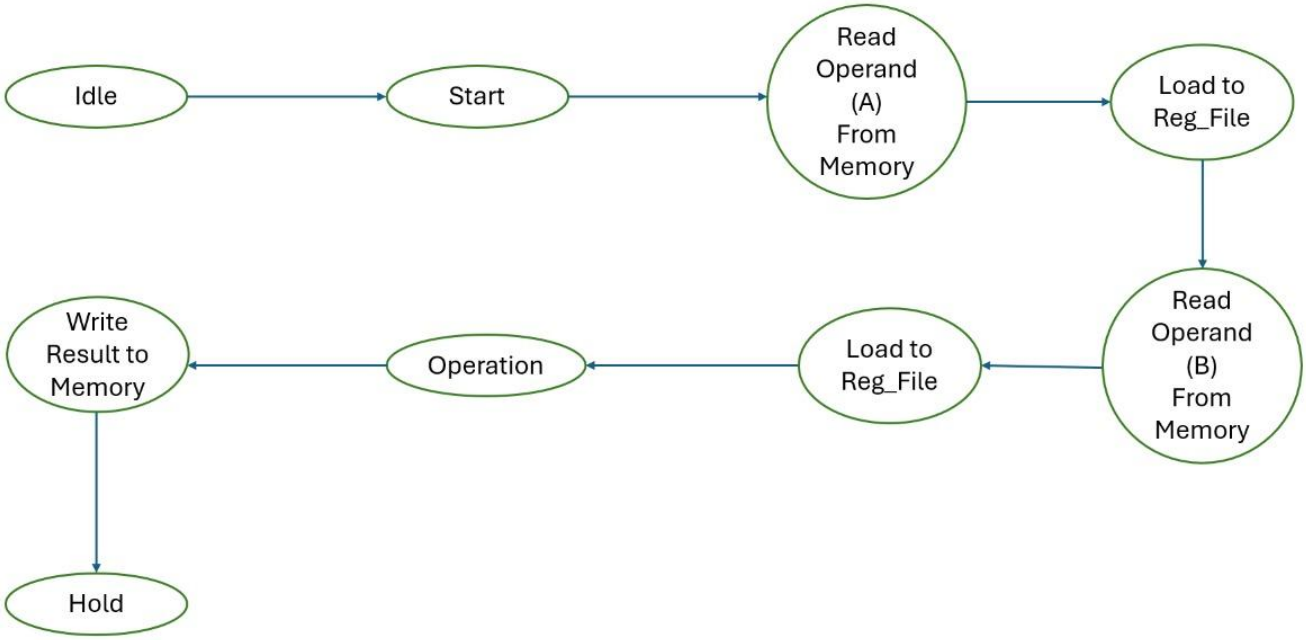
1. **Program Counter (PC)** – Generates the address of the next instruction to be executed.
2. **Instruction Register (IR)** – Holds the current instruction fetched from program memory.
3. **Controller** – Decodes the instruction and generates control signals for the other modules.
4. **Register File (RF)** – Stores temporary data and operands for the ALU.
5. **Arithmetic Logic Unit (ALU)** – Performs arithmetic and logical operations based on the control signals.
6. **Data Memory** – Stores data and the results of ALU operations.
7. **Interconnection Logic (Top Module)** – Connects all modules and manages data flow.



3. Microprocessor Execution Scenario



Scenario: Step-by-Step Operation of the 8-bit Microprocessor

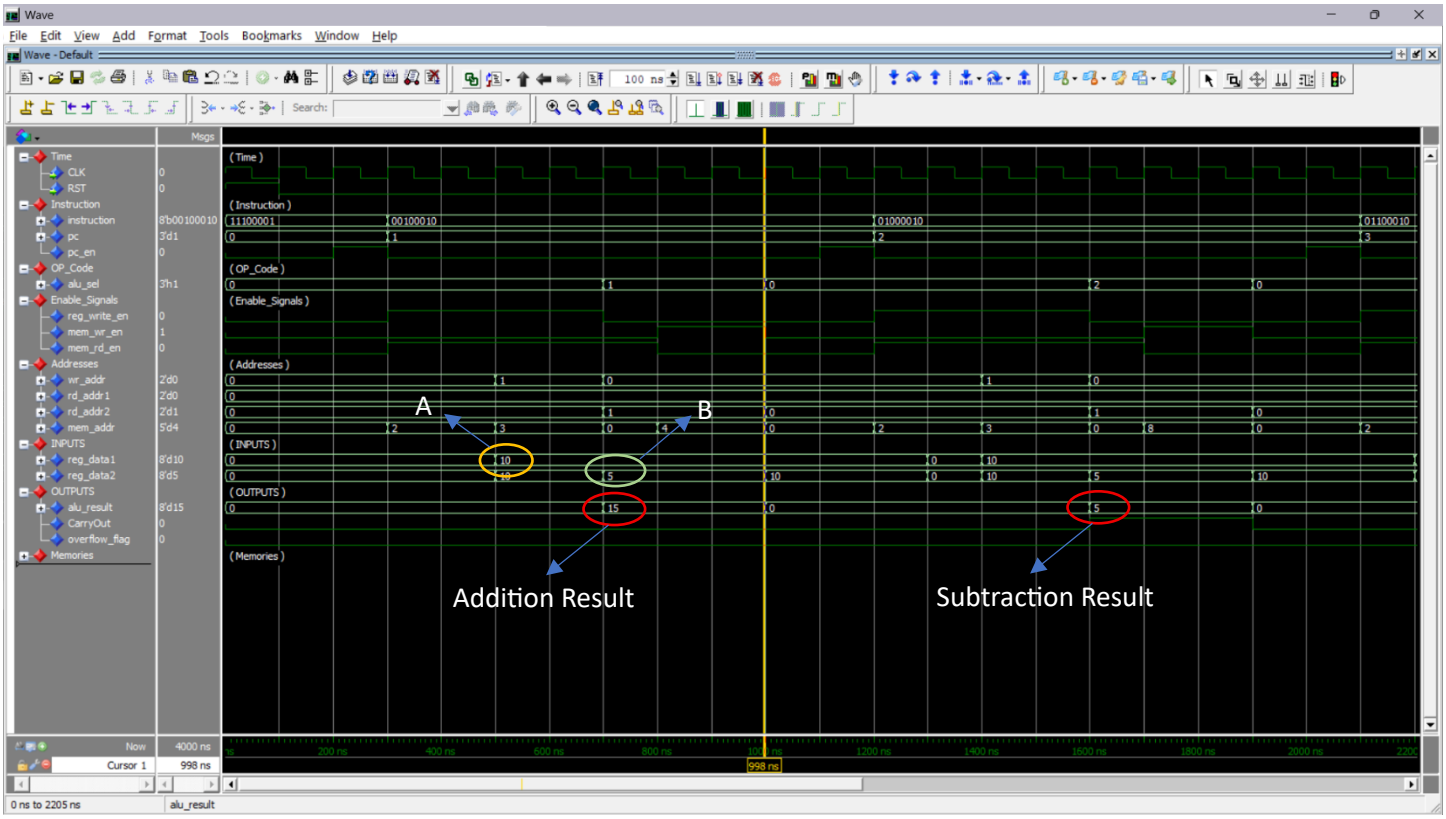


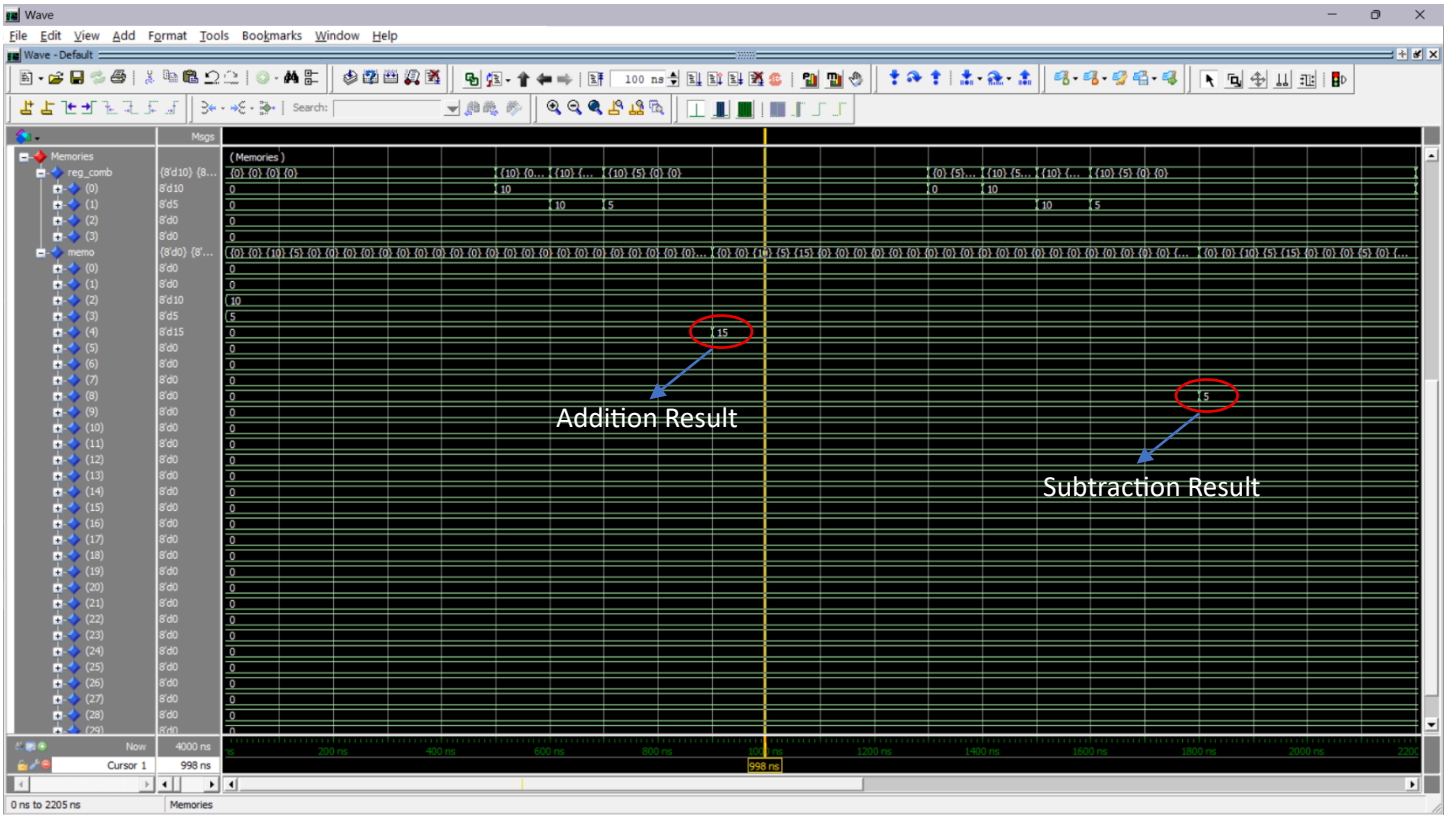
1. Idle State
    - The microprocessor starts in the idle state, waiting for a start signal or instruction fetch request.
    - No operations are performed in this state.
2. Start
    - Upon receiving the start signal, the processor transitions to fetch the first Instruction from Instruction Register.
3. Read Operand (A) from Memory
    - The processor reads the first operand (A) from the memory.
    - This operand will be used as one of the inputs for the ALU.
4. Load Operand (A) to Register File
    - Operand A is loaded into the designated register in the register file.
    - This allows fast access for the ALU during the operation stage.

5. Read Operand (B) from Memory
    - The processor reads the second operand (B) from memory.
    - Operand B is also required for the ALU computation.
6. Load Operand (B) to Register File
    - Operand B is stored in the register file, ready to be used in the operation.
7. Operation
    - The ALU performs the selected operation (ADD, SUB, MUL, AND, OR, XOR, etc.) on operands A and B.
    - Status flags such as Carry, Overflow, and Parity are updated according to the result.
8. Write Result to Memory
    - The result of the ALU operation is written back to the memory or a designated register.
    - This ensures that subsequent instructions can use this result if needed.
9. Hold
    - After the operation is complete and the result is stored, the processor enters the hold state.
    - In this state, the processor waits for the next instruction or command to execute.

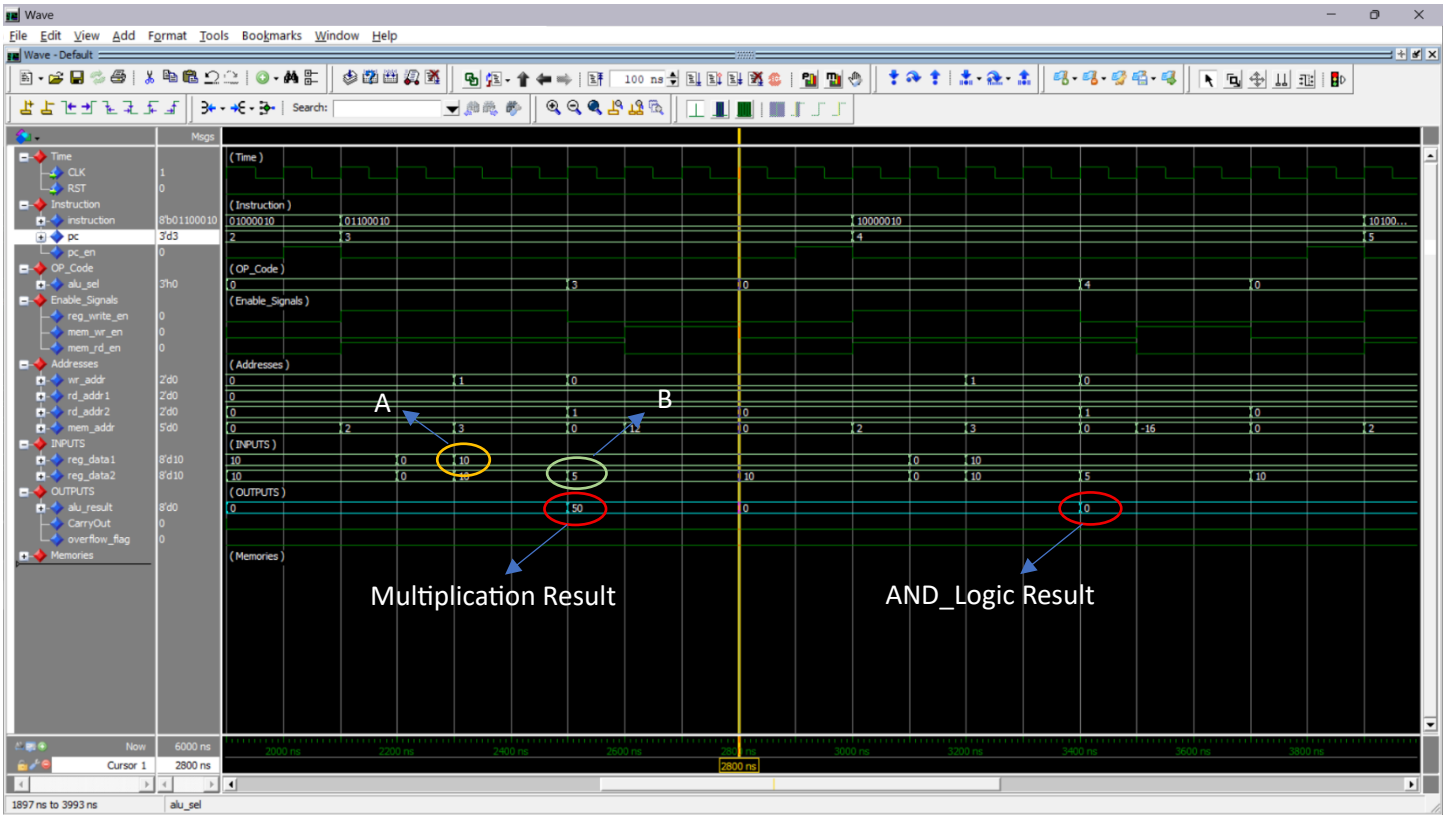
4. Simulation Results

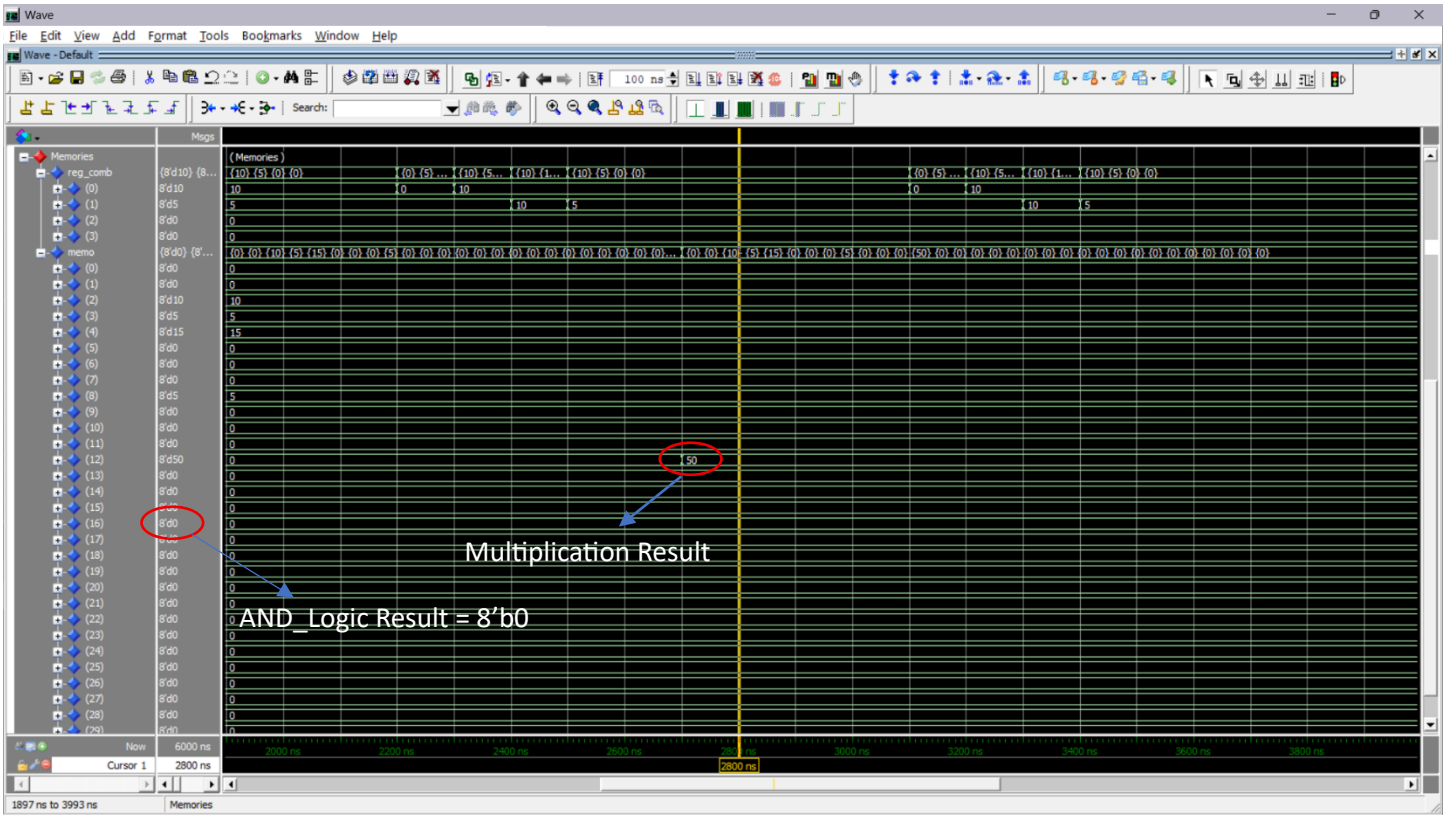
➢ Addition , Subtraction operation and Storing to the Memory



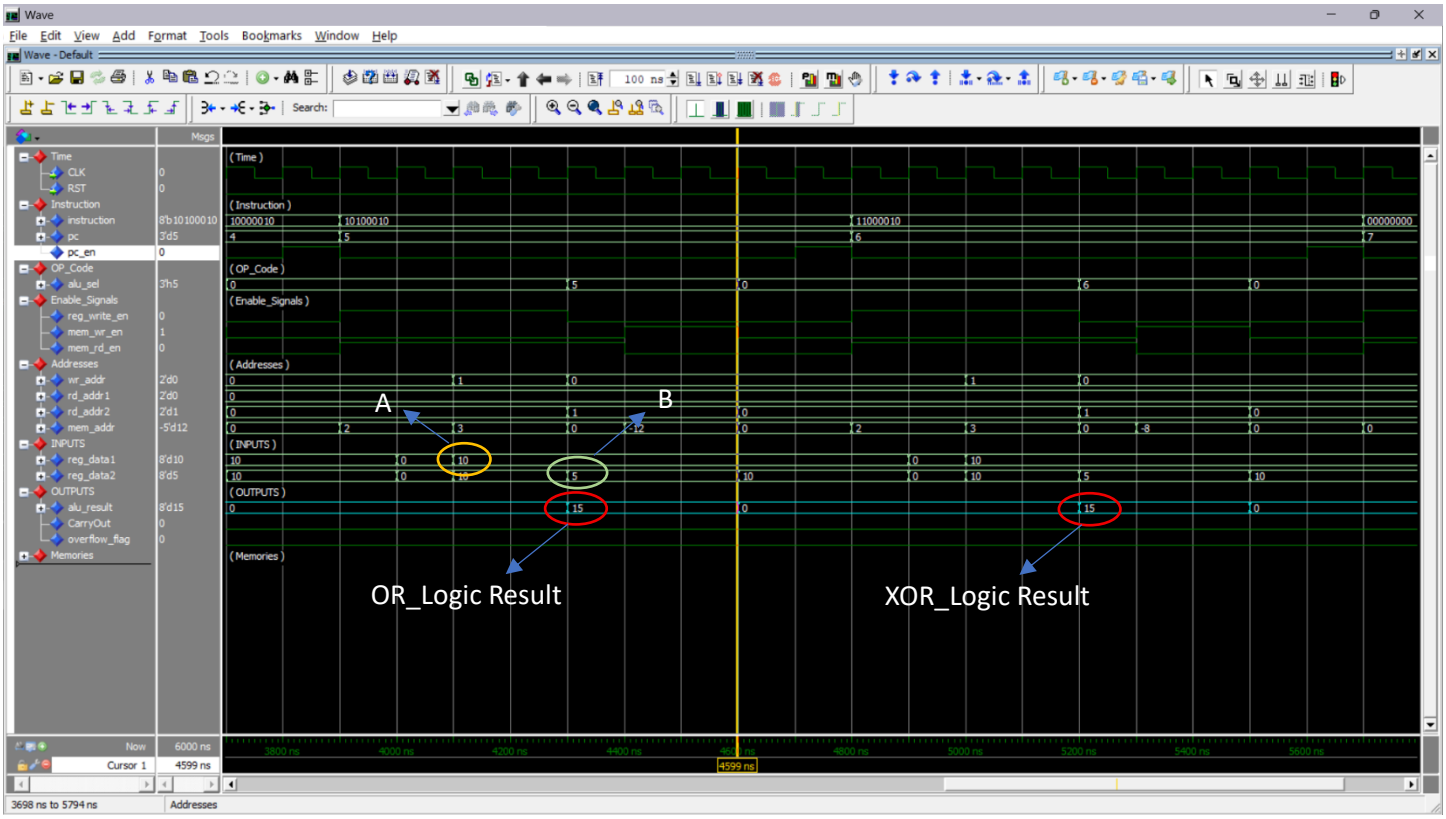


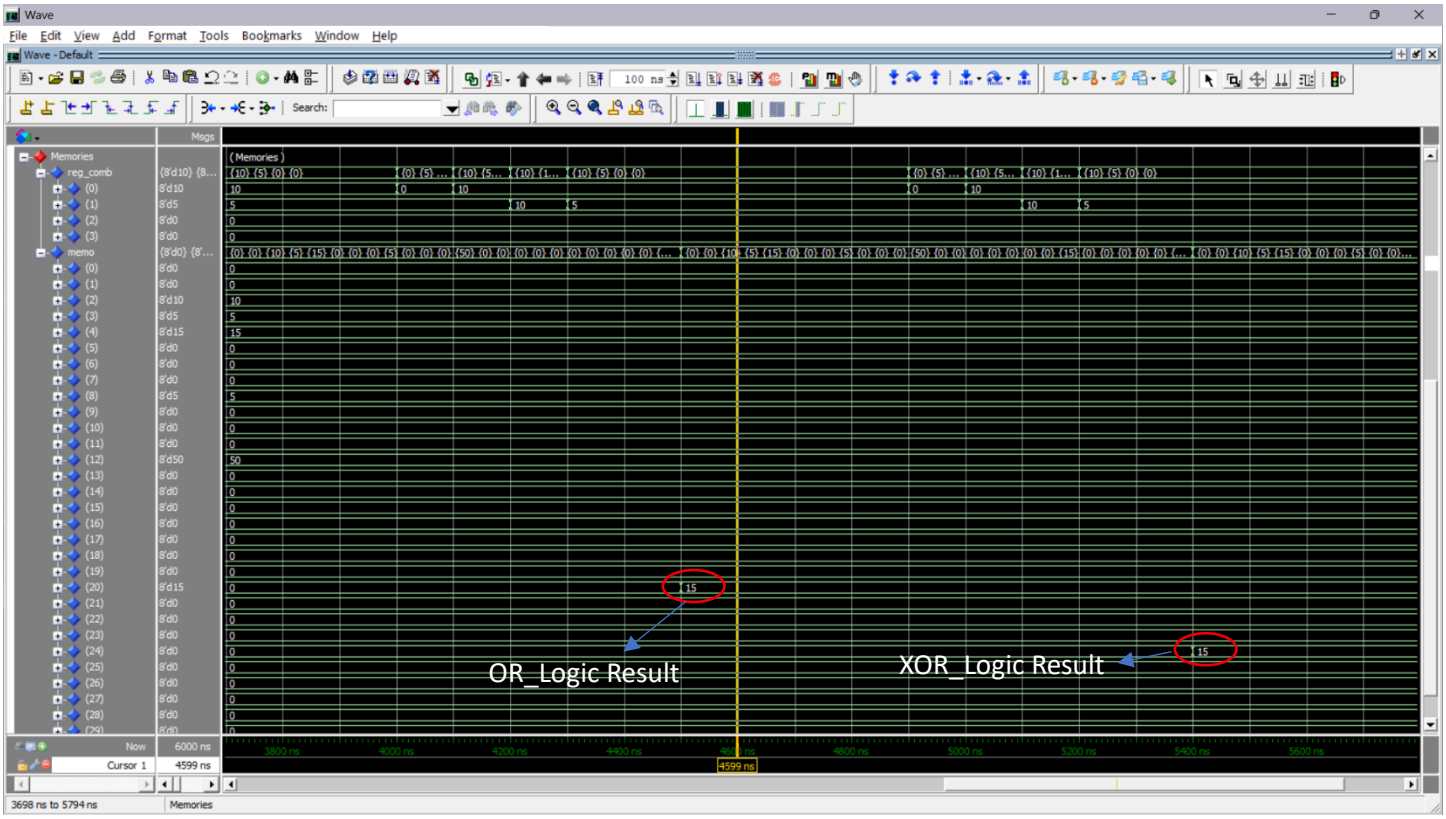
➢ Multiplication, And_Logic operation and Storing to the Memory





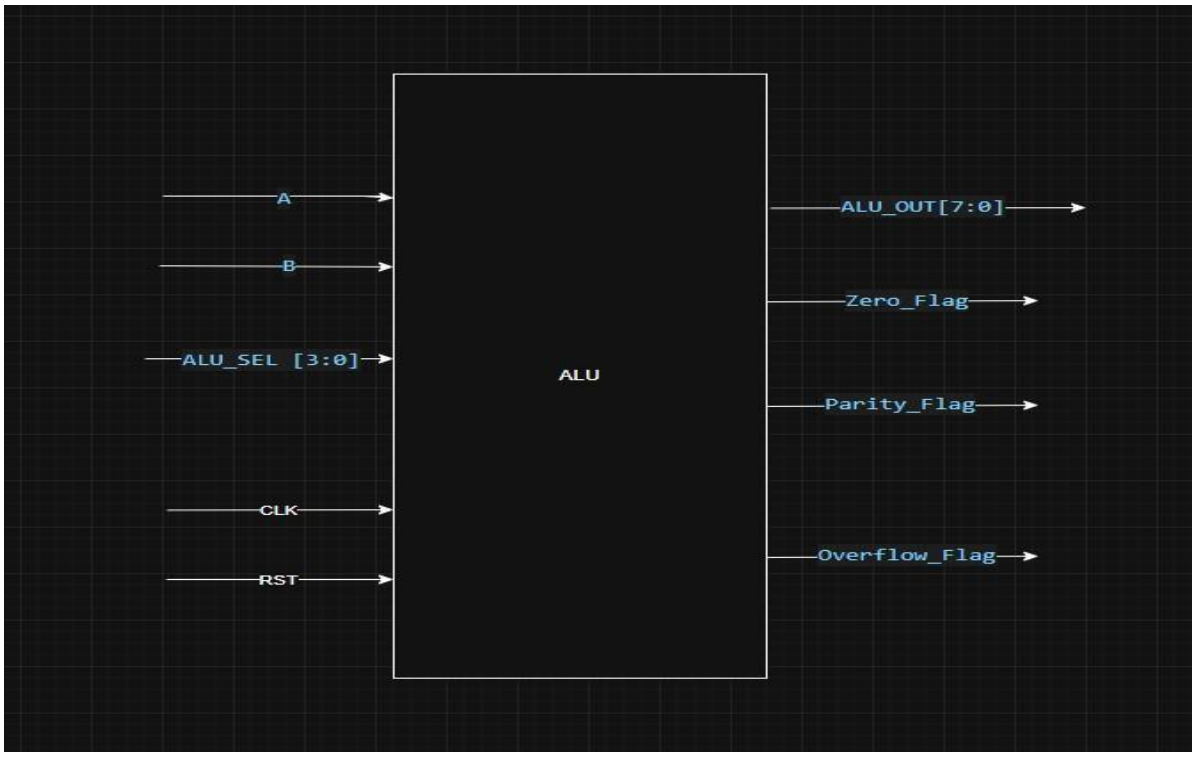
➢ OR, XOR Logic operations and storing to the memory





5. System Modules

5.1 ALU (Arithmetic Logic Unit)



Function:
Performs arithmetic and logical operations on two 8-bit inputs based on the control signal.

Inputs:

- a, b – Operands.
- ALU_Sel – Selects operation (ADD, SUB, MUL, DIV, AND, OR, NAND, XOR).

Outputs:

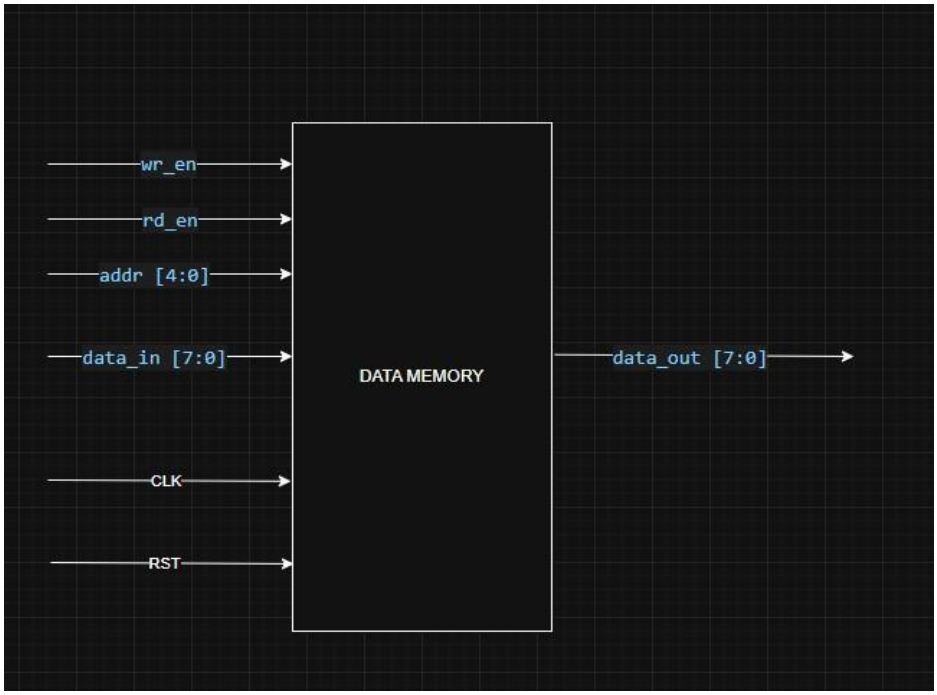
- ALU_Out – Result of the operation.
- Parity – High if result has even parity.
- Overflow – High if arithmetic overflow occurs.

Operation:

- Uses a case statement to select the required operation.
- Updates status flags after each operation.

Note: We have made the Alu by structural interpretation

5.2 Data Memory



Function:
Stores program data and allows reading/writing during execution.

Inputs:

- addr – Memory address.
- data_in – Data to write.
- Mem_write – Write enable.
- Mem_read – Read enable.
- clk.

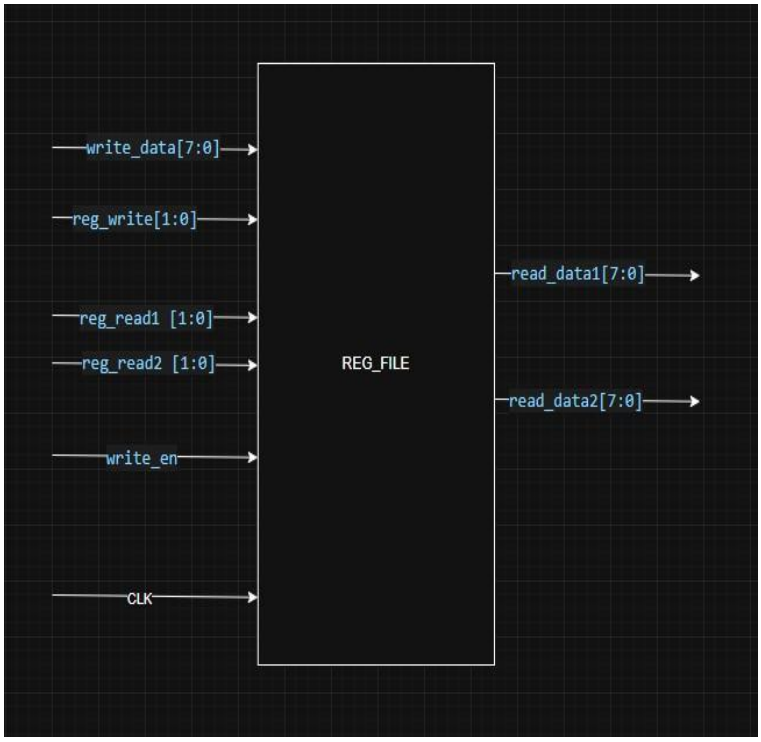
Outputs:

- data_out – Data read from memory.

Operation:

- On clk rising edge and Mem_write = 1, store data_in into memory at addr.
- On Mem_read = 1, output stored data at addr.

5.3 Register File (RF)



Function:
Stores operands and results for processor operations.

Inputs:

- write_data – Data to be written.
- reg_write – Address of register to write.
- reg_read1, reg_read2 – Addresses of registers to read.
- write_en – Enables register writing.
- clk, reset.

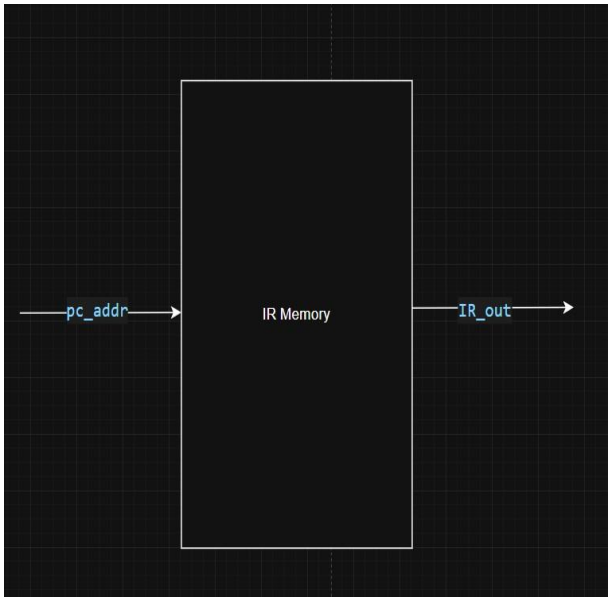
Outputs:

- read_data1, read_data2 – Data from the selected registers.

Operation:

- Synchronous write: On clk rising edge, if write_en = 1, store write_data into reg_write.
- Asynchronous read: Output read_data1 and read_data2 based on reg_read1 and reg_read2.

5.4 Instruction Register (IR)



Function:
Holds the current instruction fetched from memory and provides the opcode and operand fields to the Controller.

Inputs:

- clk – Clock signal.
- reset – Clears the register.
- IR_in – Instruction fetched from memory.
- load – Enables loading of a new instruction.

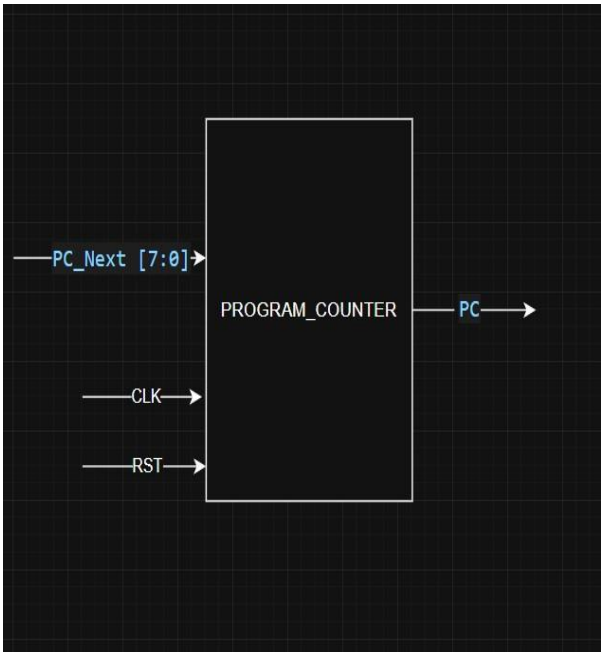
Outputs:

- opcode – Operation code (upper bits of instruction).
- operands – Address/register selection bits (lower bits of instruction).

Operation:

- On reset, clear the register.
- On rising edge of clk and load = 1, store IR_in.
- Split stored instruction into opcode and operands.

5.5 Program Counter (PC)



Function:
The Program Counter generates the address of the next instruction to be fetched from memory. It increments sequentially after each instruction fetch or loads a new address during branch/jump instructions (if implemented).

Inputs:

- clk – Clock signal for synchronous updates.
- reset – Resets the counter to zero.
- PC_in – New address to load (for branching/jumping).
- PC_load – Control signal to load a new address.

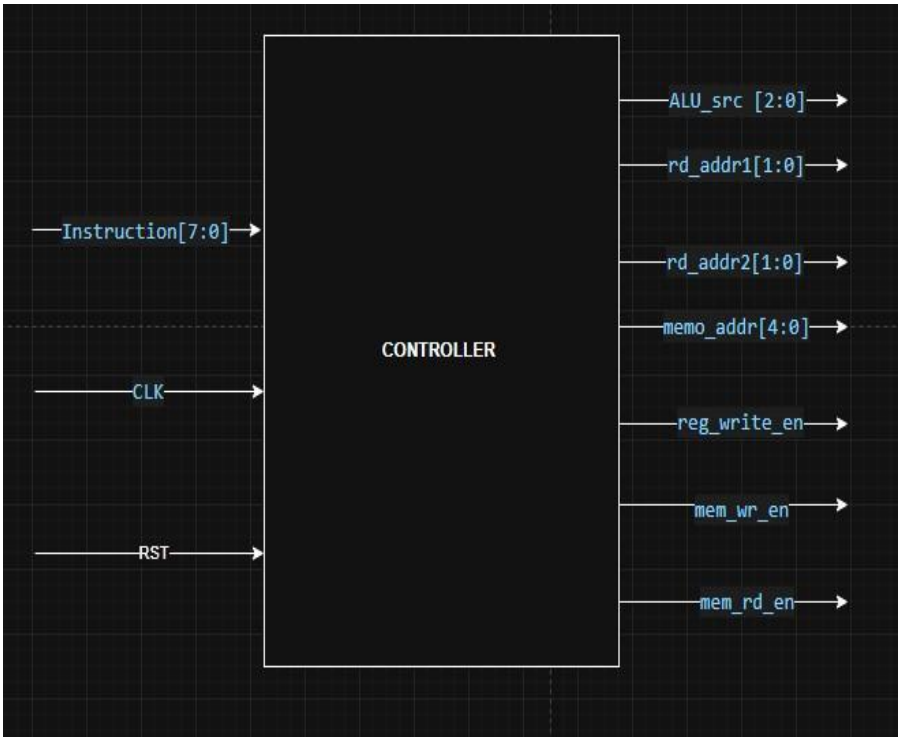
Outputs:

- PC_out – Current instruction address.

Operation:

- On reset, PC_out is set to 0.
- On each clock cycle, if PC_load is 1, load PC_in; otherwise, increment PC_out by 1.

5.6 Controller



Function:
Decodes the opcode and generates control signals for the ALU, Register File, Data Memory, and PC.

Inputs:

- opcode – From IR.
- clk, reset – System control signals.

Outputs:

- ALU_control – Selects the ALU operation.
- Reg_write_en – Enables writing to the Register File.
- Mem_read, Mem_write – Memory control.
- PC_load – Enables PC to load a new address.

Operation:

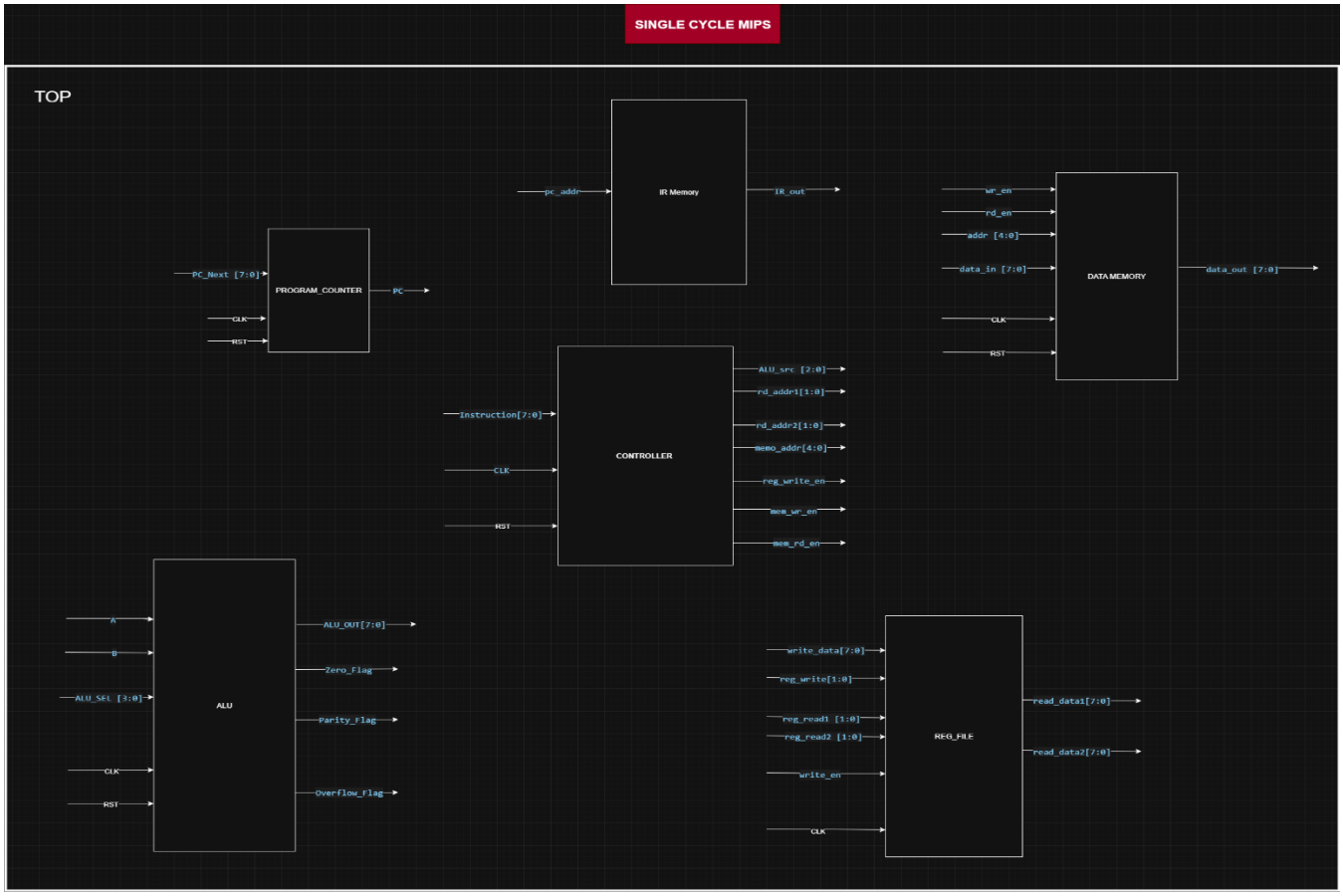
- On each clock cycle, reads opcode and sets control signals accordingly.
- Implements a simple finite state machine (FSM) for the fetch–decode–execute cycle.

5.7 Top_Module (MIPS_Top)

Function:
Integrates all components into a complete microprocessor.

Connections:

- PC outputs address to instruction memory.
- Instruction Memory outputs instruction to IR.
- IR outputs opcode to Controller and operand addresses to Register File.
- Register File outputs operands to ALU.
- ALU outputs results to Register File or Data Memory.
- Controller generates control signals for all modules.



6. Conclusion

In this project, we successfully designed and implemented an 8-bit MIPS-like microprocessor capable of executing basic arithmetic and logical instructions. The processor includes all key modules: Program Counter, Instruction Register, Control Unit, Register File, ALU, and Data Memory, each working together to execute instructions in a sequential manner.

The ALU performs operations like ADD, SUB, MUL, AND, OR, and XOR, with proper status flag updates (Carry, Overflow, Parity). The register file allows reading and writing of operands, while the instruction memory and data memory enable instruction fetch and data storage.

Through simulation, we verified that instructions are executed correctly, intermediate results are accurately processed, and flags are updated as expected. This demonstrates a functional pipeline of instruction execution, like a simplified MIPS architecture, providing a strong foundation for understanding microprocessor design principles.

Overall, this project strengthens understanding of digital design, processor architecture, and VHDL implementation, and it can be extended in the future for more complex instructions or a pipelined version for higher efficiency.