National Telecommunication Institute

Design and Implementation of an 8-Bit 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. Abstract

This project presents the design and implementation of a simple 8-bit microprocessor using Hardware Description Languages (VHDL). The processor is composed of fundamental modules including a Program Counter, Instruction Register, Controller, Register File, Arithmetic Logic Unit (ALU), and Data Memory.

The ALU is implemented to perform arithmetic and logical operations such as addition, subtraction, multiplication, division, AND, OR, NAND, and XOR, with status flags for zero, parity, and overflow detection. The remaining components are implemented to handle instruction fetching, decoding, execution control, and data storage.

The microprocessor operates on an instruction set stored in the instruction register and sequentially executes instructions by incrementing the program counter. Data transfer between registers, ALU, and memory is managed by the controller based on the opcode.

2. Introduction

Microprocessors are the heart of modern digital systems, capable of performing arithmetic, logical, and control operations to execute programmed instructions. They are widely used in embedded systems, communication devices, and computing applications. Understanding how a microprocessor works at the hardware level is essential for students and engineers in the field of digital electronics and computer architecture.

This project focuses on the design and implementation of a simple 8-bit microprocessor using Hardware Description Languages (VHDL for control and structural modules). The microprocessor is designed to execute a small set of predefined instructions, enabling operations such as addition, subtraction, multiplication, division, and bitwise logic.

The system consists of key modules — Program Counter (PC), Instruction Register (IR), Controller, Register File, Arithmetic Logic Unit (ALU), and Data Memory — interconnected to form a functional processing unit. The Program Counter generates instruction addresses, the Instruction Register holds the current instruction, and the Controller decodes it to generate control signals for the ALU, registers, and memory. The Register File stores operand, the ALU executes the operations, and the Data Memory stores results or retrieves data when needed.

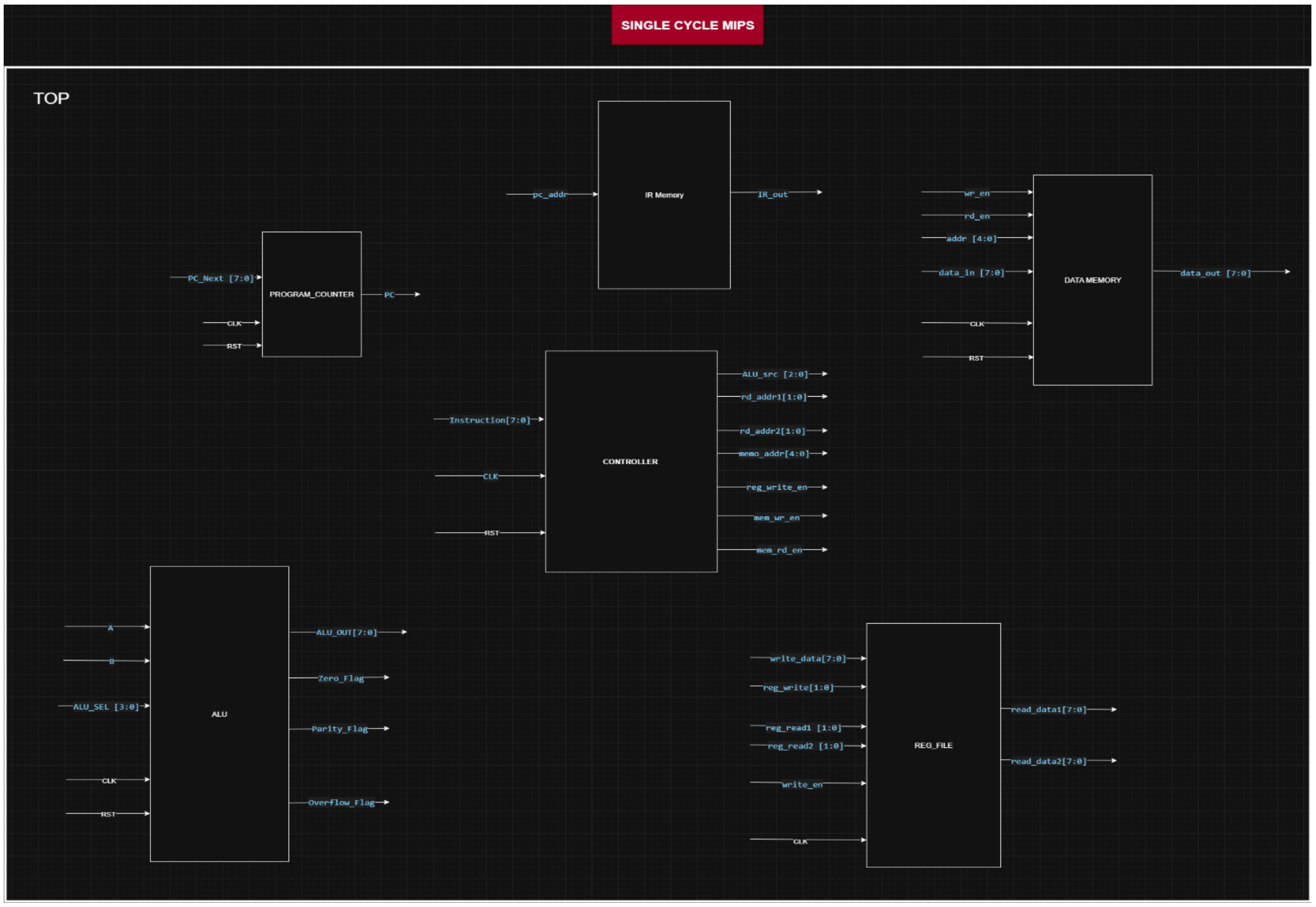
This project not only demonstrates the fundamental working of a processor but also serves as a practical exercise in digital design, combining multiple HDL modules into a fully functional system. The modular approach facilitates testing and provides a foundation for future scalability, such as expanding the instruction set or increasing data width.

3. System Overview

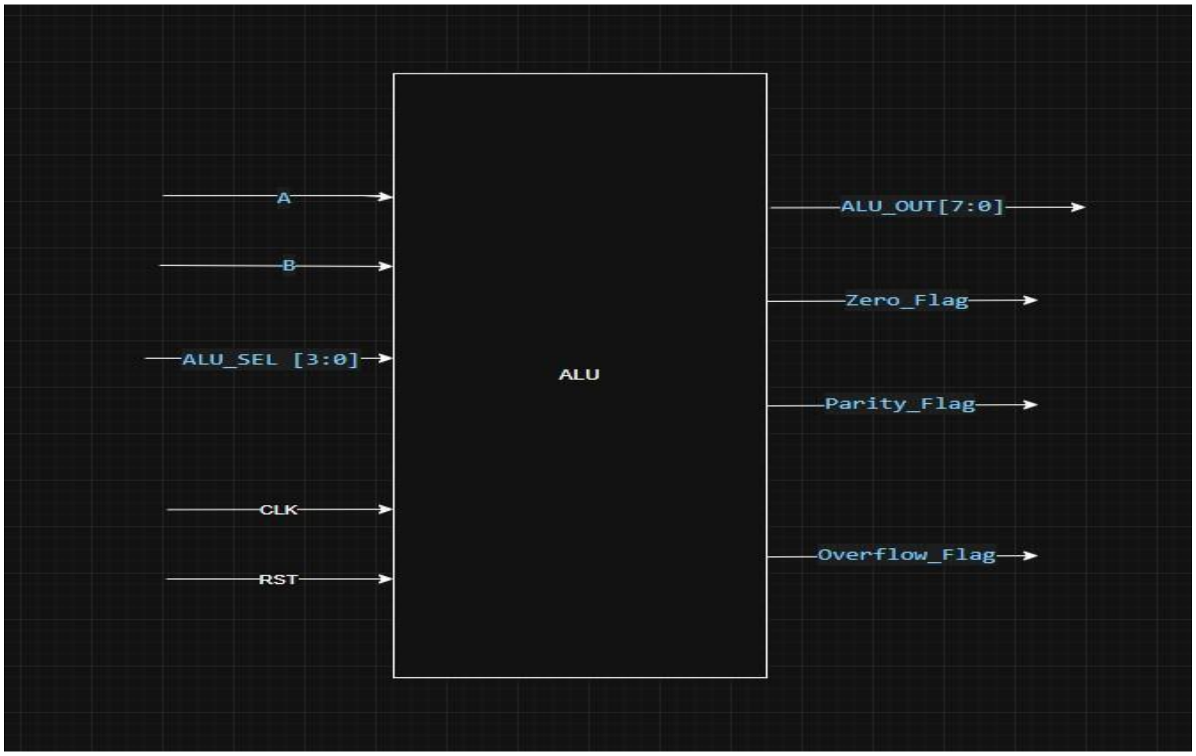
The designed 8-bit microprocessor follows a simple **fetch–decode–execute** cycle. It consists of interconnected modules that collectively fetch instructions from memory, decode them, execute the required operations, and store the results.

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.



3.1 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.

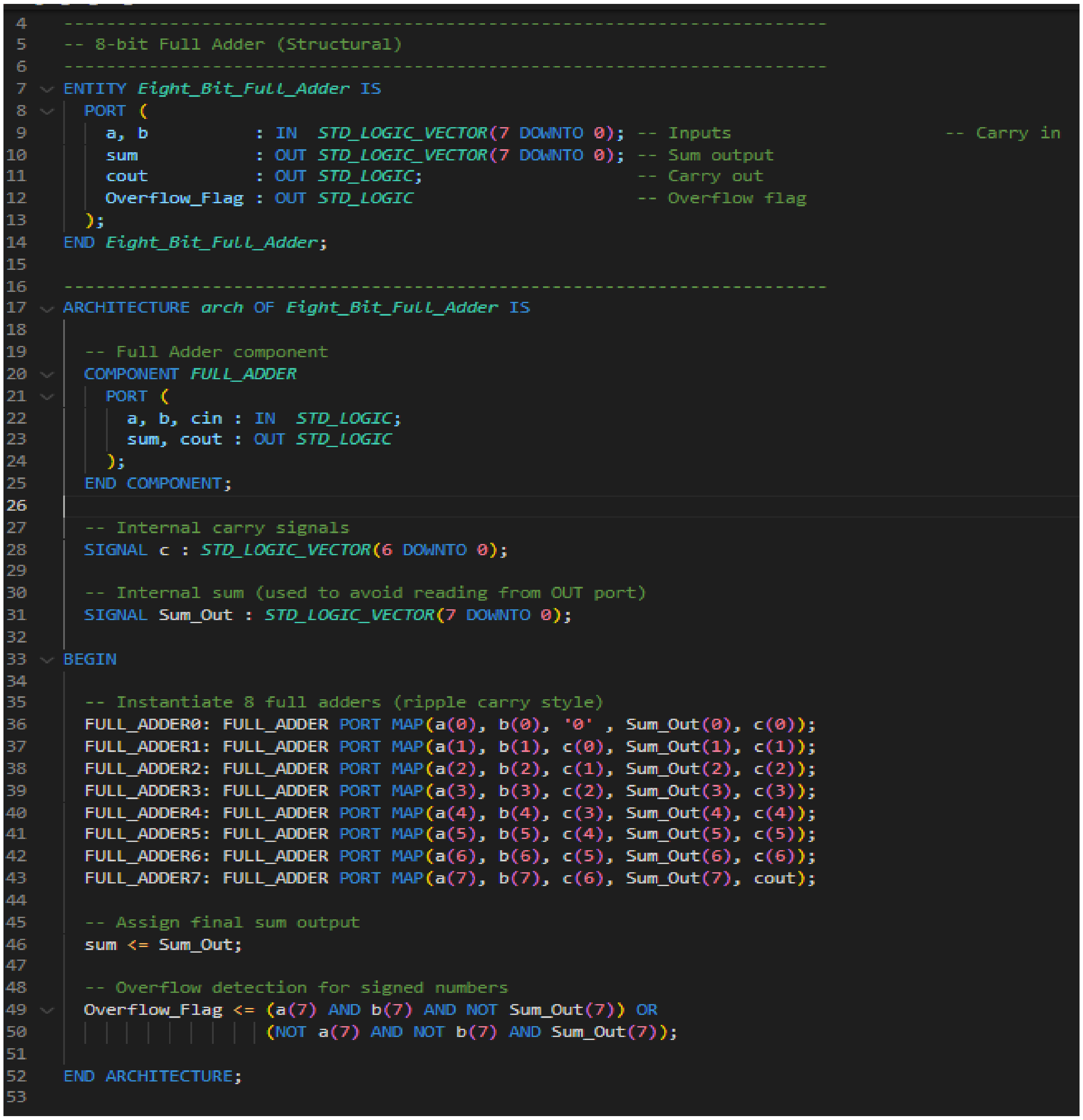
Operation:

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

Note: We have made the Alu by structural interpretation

3.1.1 ALU Building blocks

-- 8-Bit Full Adder



```vhdl
------------------------------------------------------------------------
-- 8-bit Full Adder (Structural)
------------------------------------------------------------------------
ENTITY Eight_Bit_Full_Adder IS
  PORT (
    a, b          : IN  STD_LOGIC_VECTOR(7 DOWNTO 0); -- Inputs                    -- Carry in
    sum           : OUT STD_LOGIC_VECTOR(7 DOWNTO 0); -- Sum output
    cout          : OUT STD_LOGIC;                    -- Carry out
    Overflow_Flag : OUT STD_LOGIC                     -- Overflow flag
  );
END Eight_Bit_Full_Adder;

------------------------------------------------------------------------
ARCHITECTURE arch OF Eight_Bit_Full_Adder IS

  -- Full Adder component
  COMPONENT FULL_ADDER
    PORT (
      a, b, cin : IN  STD_LOGIC;
      sum, cout : OUT STD_LOGIC
    );
  END COMPONENT;

  -- Internal carry signals
  SIGNAL c : STD_LOGIC_VECTOR(6 DOWNTO 0);

  -- Internal sum (used to avoid reading from OUT port)
  SIGNAL Sum_Out : STD_LOGIC_VECTOR(7 DOWNTO 0);

BEGIN

  -- Instantiate 8 full adders (ripple carry style)
  FULL_ADDER0: FULL_ADDER PORT MAP(a(0), b(0), '0' , Sum_Out(0), c(0));
  FULL_ADDER1: FULL_ADDER PORT MAP(a(1), b(1), c(0), Sum_Out(1), c(1));
  FULL_ADDER2: FULL_ADDER PORT MAP(a(2), b(2), c(1), Sum_Out(2), c(2));
  FULL_ADDER3: FULL_ADDER PORT MAP(a(3), b(3), c(2), Sum_Out(3), c(3));
  FULL_ADDER4: FULL_ADDER PORT MAP(a(4), b(4), c(3), Sum_Out(4), c(4));
  FULL_ADDER5: FULL_ADDER PORT MAP(a(5), b(5), c(4), Sum_Out(5), c(5));
  FULL_ADDER6: FULL_ADDER PORT MAP(a(6), b(6), c(5), Sum_Out(6), c(6));
  FULL_ADDER7: FULL_ADDER PORT MAP(a(7), b(7), c(6), Sum_Out(7), cout);

  -- Assign final sum output
  sum <= Sum_Out;

  -- Overflow detection for signed numbers
  Overflow_Flag <= (a(7) AND b(7) AND NOT Sum_Out(7)) OR
                   (NOT a(7) AND NOT b(7) AND Sum_Out(7));

END ARCHITECTURE;
```

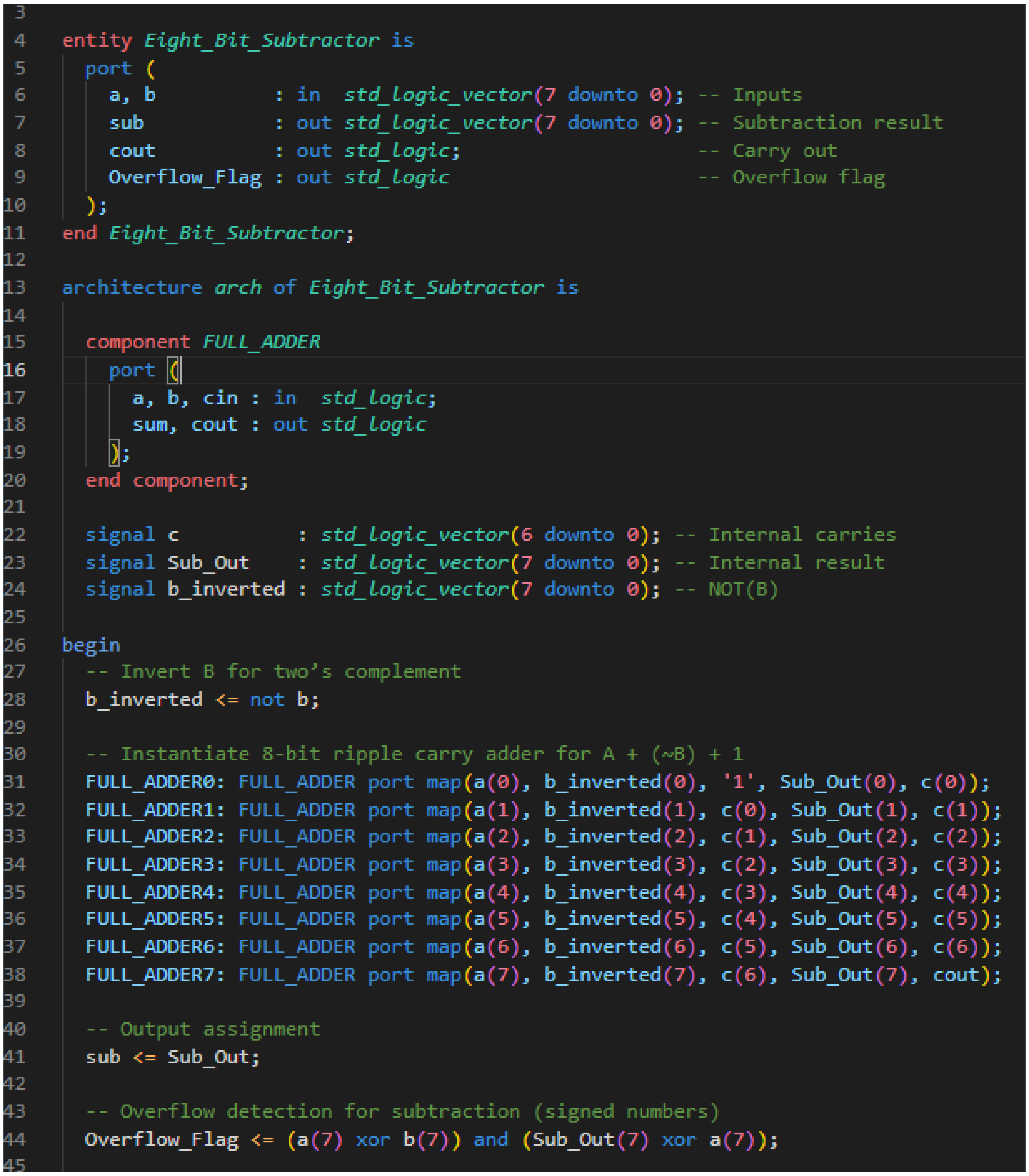
Purpose: Adds two 8-bit numbers and b.

How it works:

- Internally uses 8 single-bit FULL_ADDER units connected in ripple-carry style.
- First full adder takes cin = '0'.
- Carry from each bit is passed to the next stage.
- Produces:
  - sum → the 8-bit result.

- cout → final carry out.
- Overflow_Flag → detects signed overflow (different from carry)

-- 8_Bit_Subtractor



```vhdl

entity Eight_Bit_Subtractor is
  port (
    a, b          : in  std_logic_vector(7 downto 0); -- Inputs
    sub           : out std_logic_vector(7 downto 0); -- Subtraction result
    cout          : out std_logic;                    -- Carry out
    Overflow_Flag : out std_logic                     -- Overflow flag
  );
end Eight_Bit_Subtractor;

architecture arch of Eight_Bit_Subtractor is

  component FULL_ADDER
    port (
      a, b, cin : in  std_logic;
      sum, cout : out std_logic
    );
  end component;

  signal c          : std_logic_vector(6 downto 0); -- Internal carries
  signal Sub_Out    : std_logic_vector(7 downto 0); -- Internal result
  signal b_inverted : std_logic_vector(7 downto 0); -- NOT(B)

begin
  -- Invert B for two’s complement
  b_inverted <= not b;

  -- Instantiate 8-bit ripple carry adder for A + (~B) + 1
  FULL_ADDER0: FULL_ADDER port map(a(0), b_inverted(0), '1', Sub_Out(0), c(0));
  FULL_ADDER1: FULL_ADDER port map(a(1), b_inverted(1), c(0), Sub_Out(1), c(1));
  FULL_ADDER2: FULL_ADDER port map(a(2), b_inverted(2), c(1), Sub_Out(2), c(2));
  FULL_ADDER3: FULL_ADDER port map(a(3), b_inverted(3), c(2), Sub_Out(3), c(3));
  FULL_ADDER4: FULL_ADDER port map(a(4), b_inverted(4), c(3), Sub_Out(4), c(4));
  FULL_ADDER5: FULL_ADDER port map(a(5), b_inverted(5), c(4), Sub_Out(5), c(5));
  FULL_ADDER6: FULL_ADDER port map(a(6), b_inverted(6), c(5), Sub_Out(6), c(6));
  FULL_ADDER7: FULL_ADDER port map(a(7), b_inverted(7), c(6), Sub_Out(7), cout);

  -- Output assignment
  sub <= Sub_Out;

  -- Overflow detection for subtraction (signed numbers)
  Overflow_Flag <= (a(7) xor b(7)) and (Sub_Out(7) xor a(7));

```

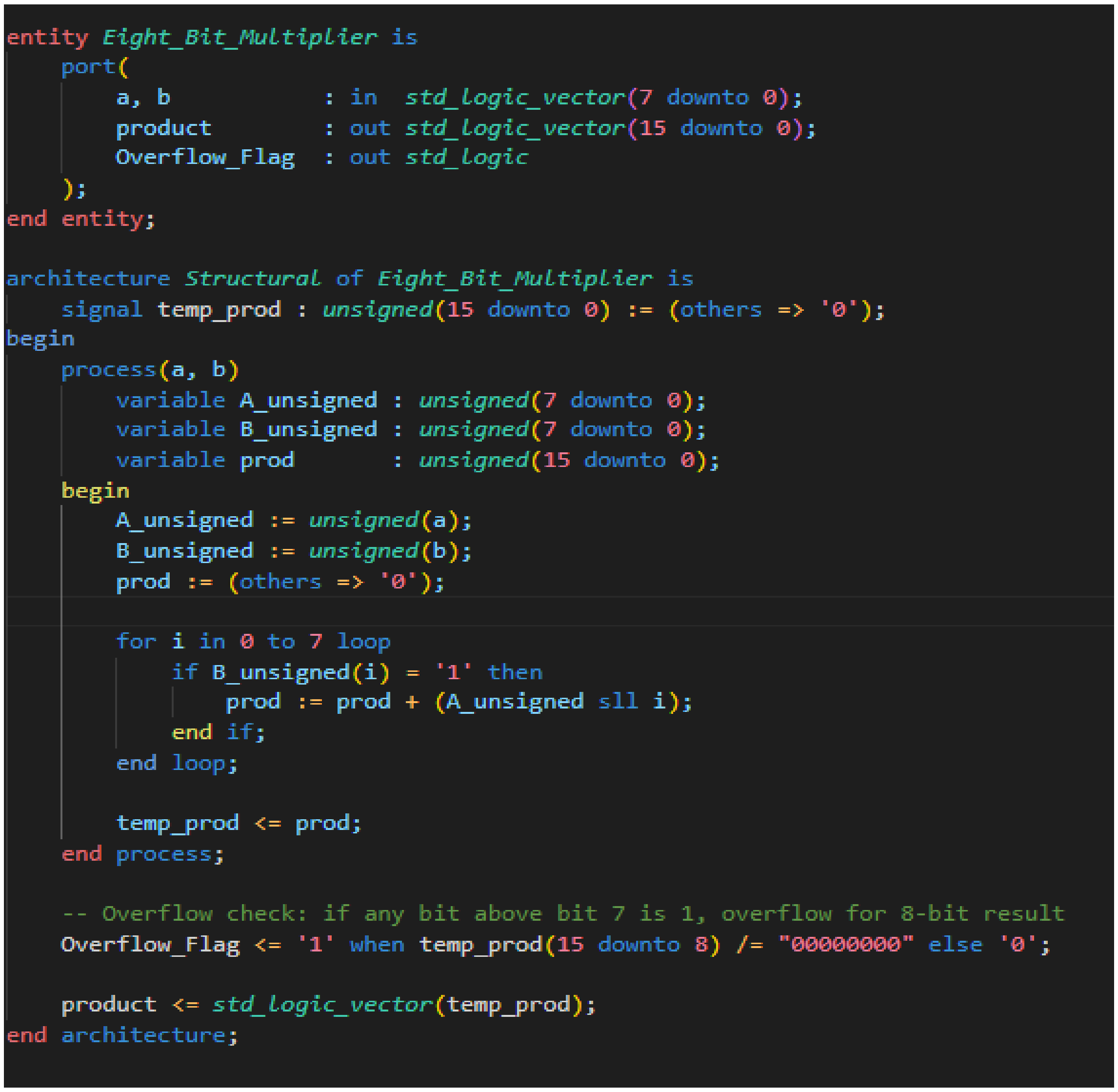
Purpose: Subtracts b from a (a- b).

How it works:

- Uses two’s complement subtraction: invert b and add 1.

- Internally uses the same FULL_ADDER ripple style.
- Produces:
    - sub → the subtraction result.
    - cout → carry out.
    - Overflow_Flag → detects signed subtraction overflow.

-- 8_Bit_Multiplier



```vhdl
entity Eight_Bit_Multiplier is
    port(
        a, b            : in  std_logic_vector(7 downto 0);
        product         : out std_logic_vector(15 downto 0);
        Overflow_Flag   : out std_logic
    );
end entity;

architecture Structural of Eight_Bit_Multiplier is
    signal temp_prod : unsigned(15 downto 0) := (others => '0');
begin
    process(a, b)
        variable A_unsigned : unsigned(7 downto 0);
        variable B_unsigned : unsigned(7 downto 0);
        variable prod       : unsigned(15 downto 0);
    begin
        A_unsigned := unsigned(a);
        B_unsigned := unsigned(b);
        prod := (others => '0');

        for i in 0 to 7 loop
            if B_unsigned(i) = '1' then
                prod := prod + (A_unsigned sll i);
            end if;
        end loop;

        temp_prod <= prod;
    end process;

    -- Overflow check: if any bit above bit 7 is 1, overflow for 8-bit result
    Overflow_Flag <= '1' when temp_prod(15 downto 8) /= "00000000" else '0';

    product <= std_logic_vector(temp_prod);
end architecture;
```

How the ALU Works (Step-by-Step)

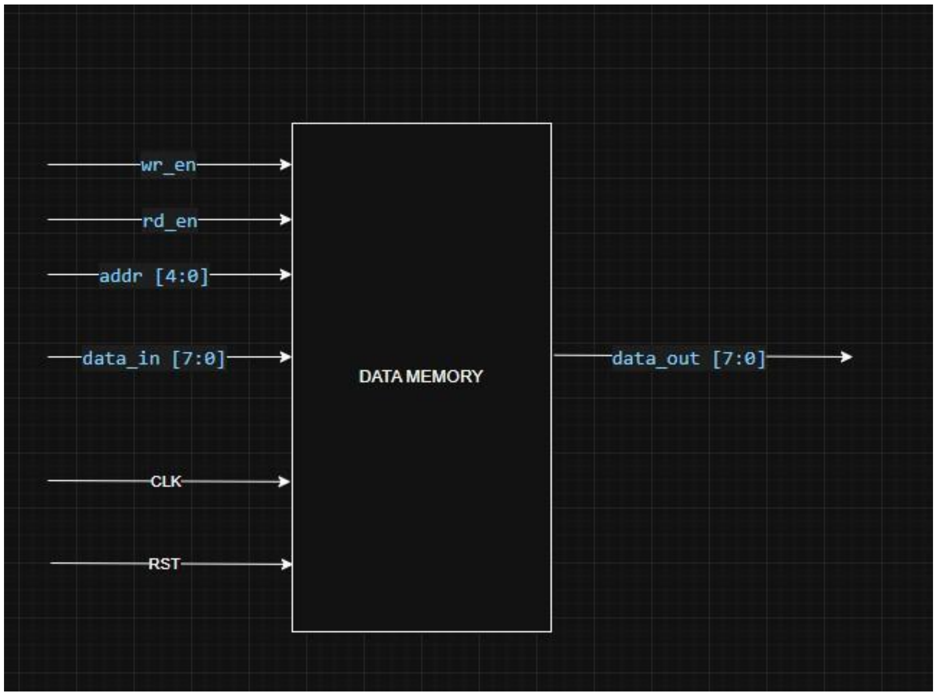
Step 1 – Inputs

- You feed:
  - A → 8-bit operand 1.
  - B → 8-bit operand 2.
  - ALU_SEL → 3-bit control code telling the ALU which operation to do.

Step 2 – Parallel Processing

- Regardless of ALU_SEL, **all components run in parallel**:
  - **Adder** calculates A + B.
  - **Subtractor** calculates A- B.
  - **Multiplier** calculates A × B.
  - **AND**, **OR**, **XOR** calculate their respective bitwise results.

3.2 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.

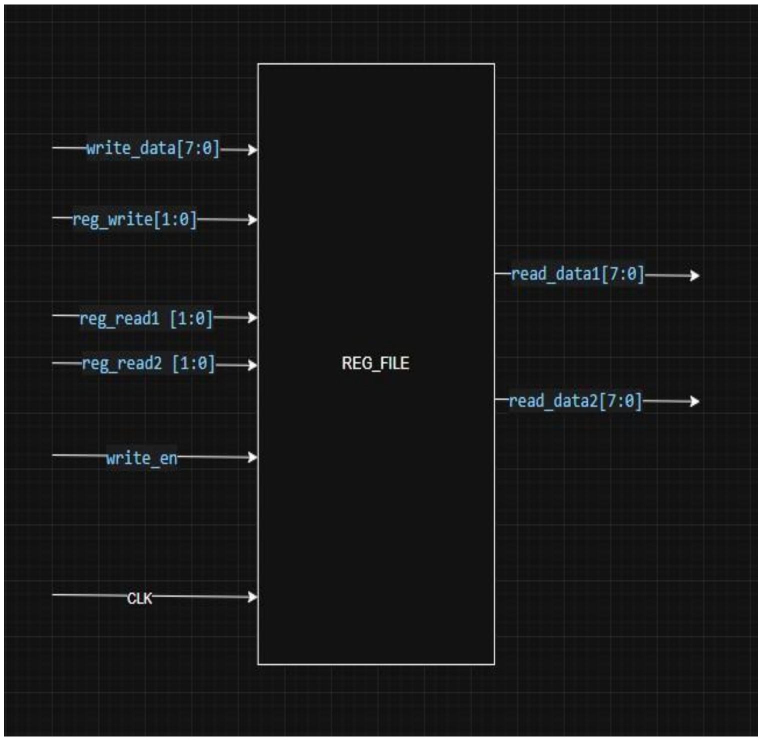
Outputs:

- 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.

3.3 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.

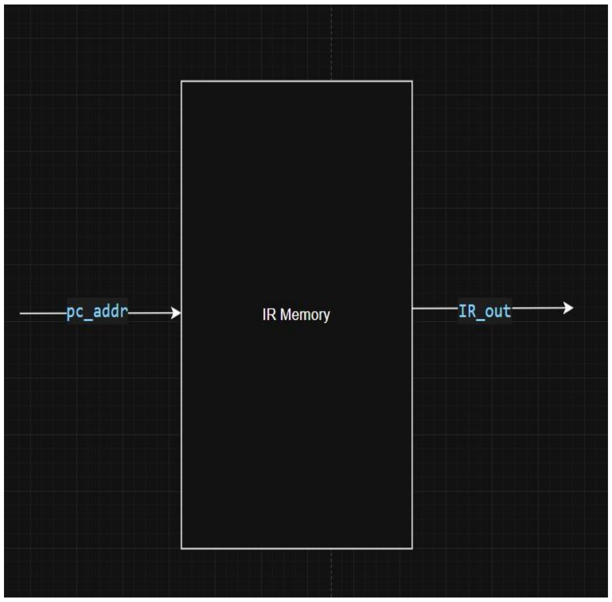
Outputs:

- 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.

3.4 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.

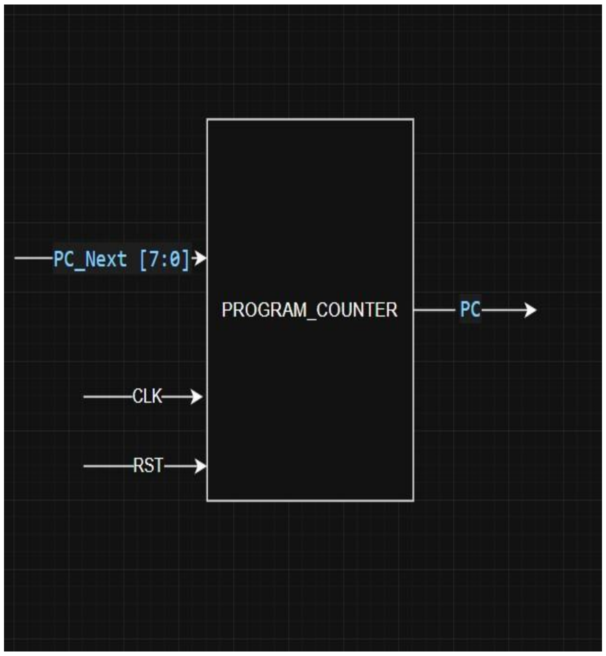
Outputs:

- 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.

3.5 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.

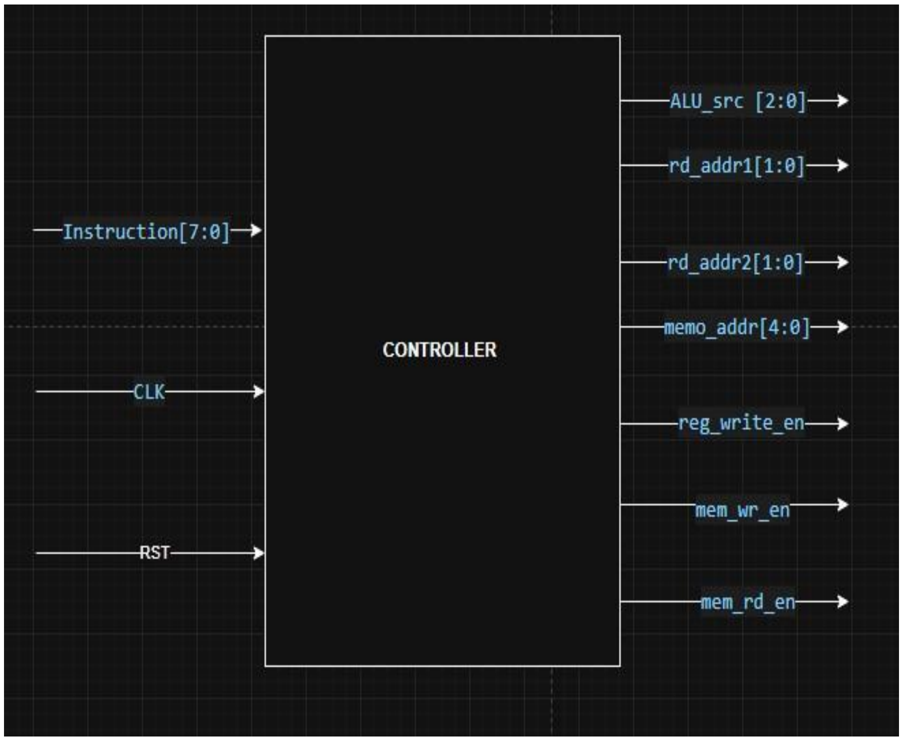
Outputs:

- 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.

3.6 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.

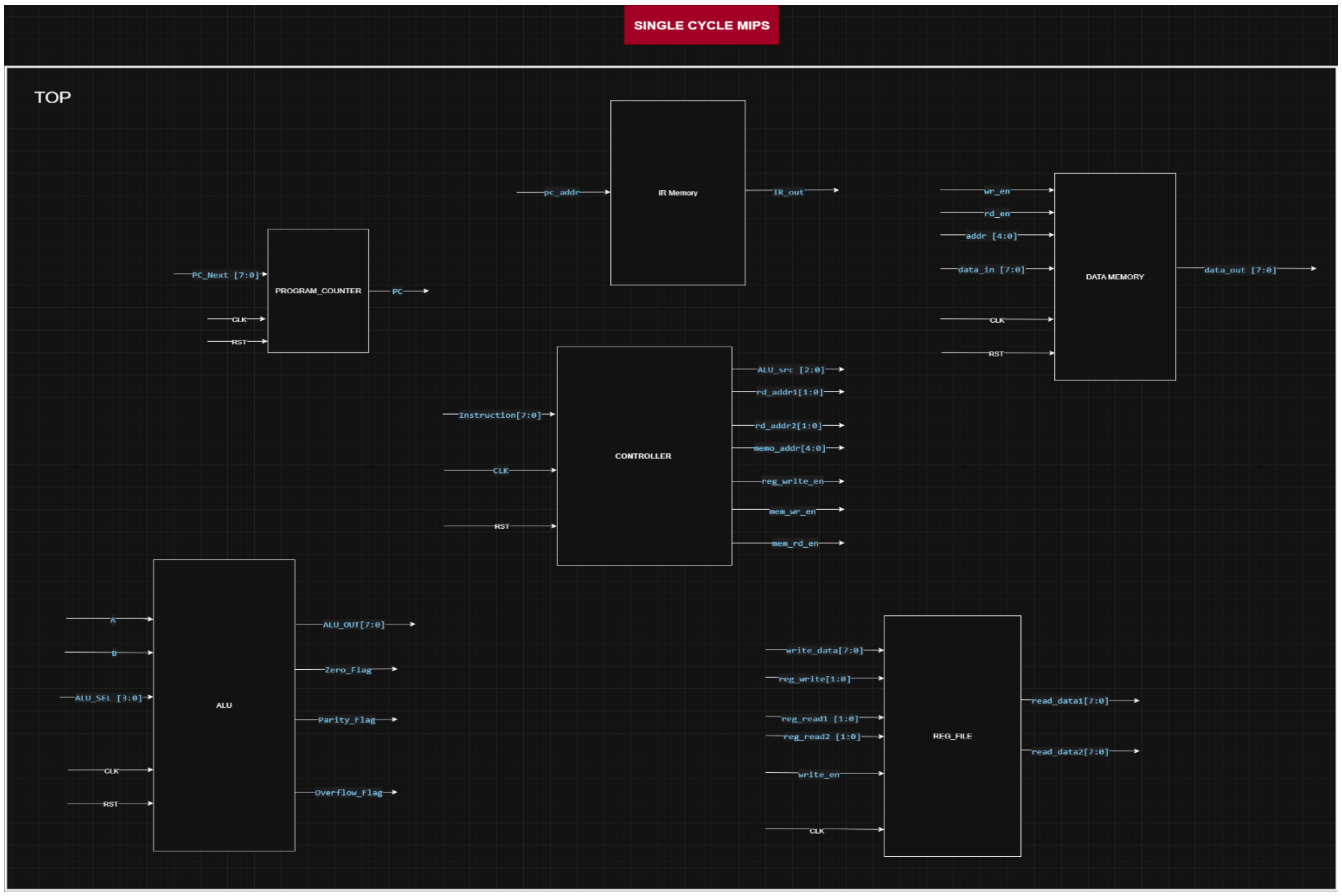
3.7 Top_Module (MIPS_Top)

Function:

Integrates all components into a complete microprocessor.

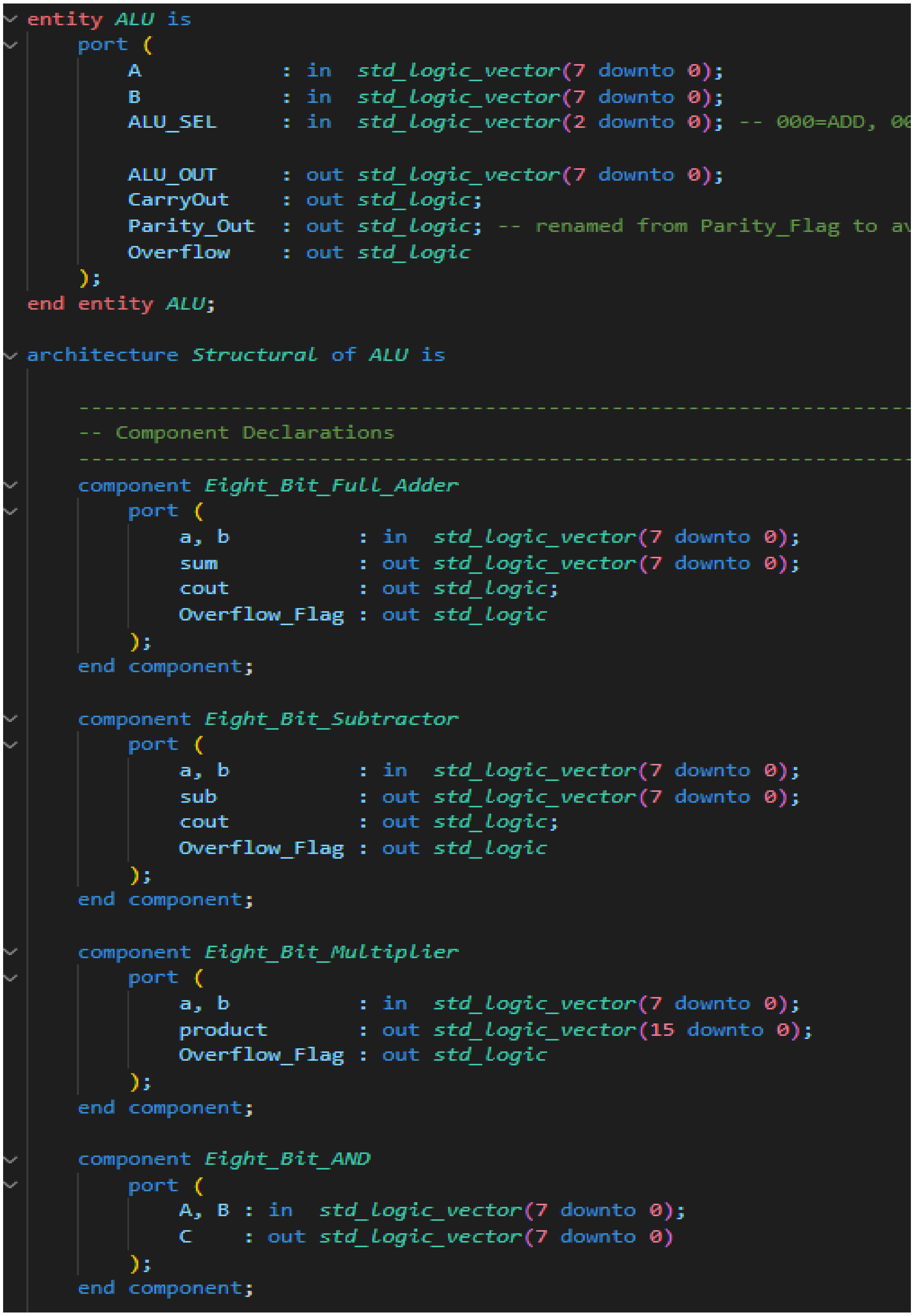
Connections:

- 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.

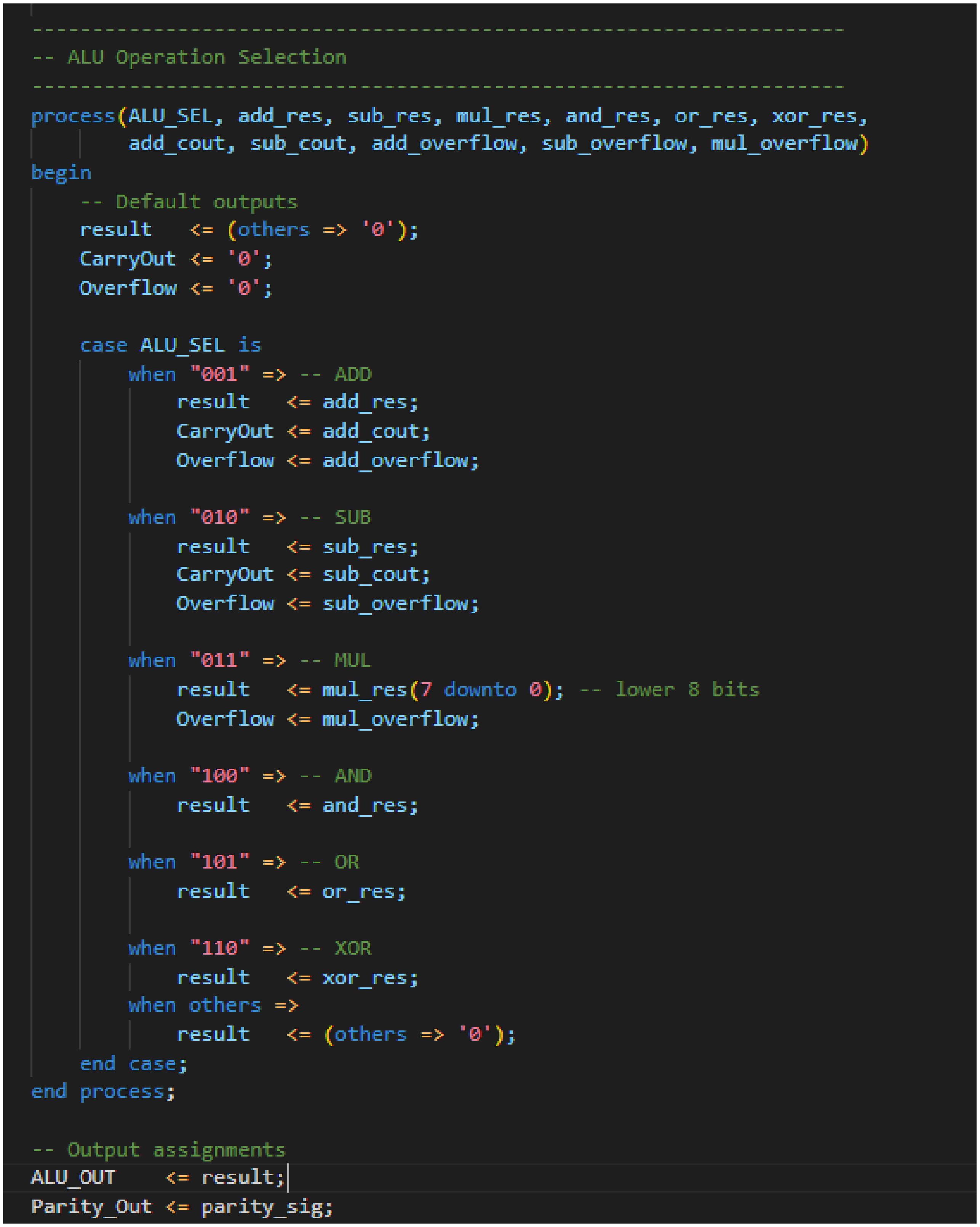


4. VHDL codes

4.1 ALU



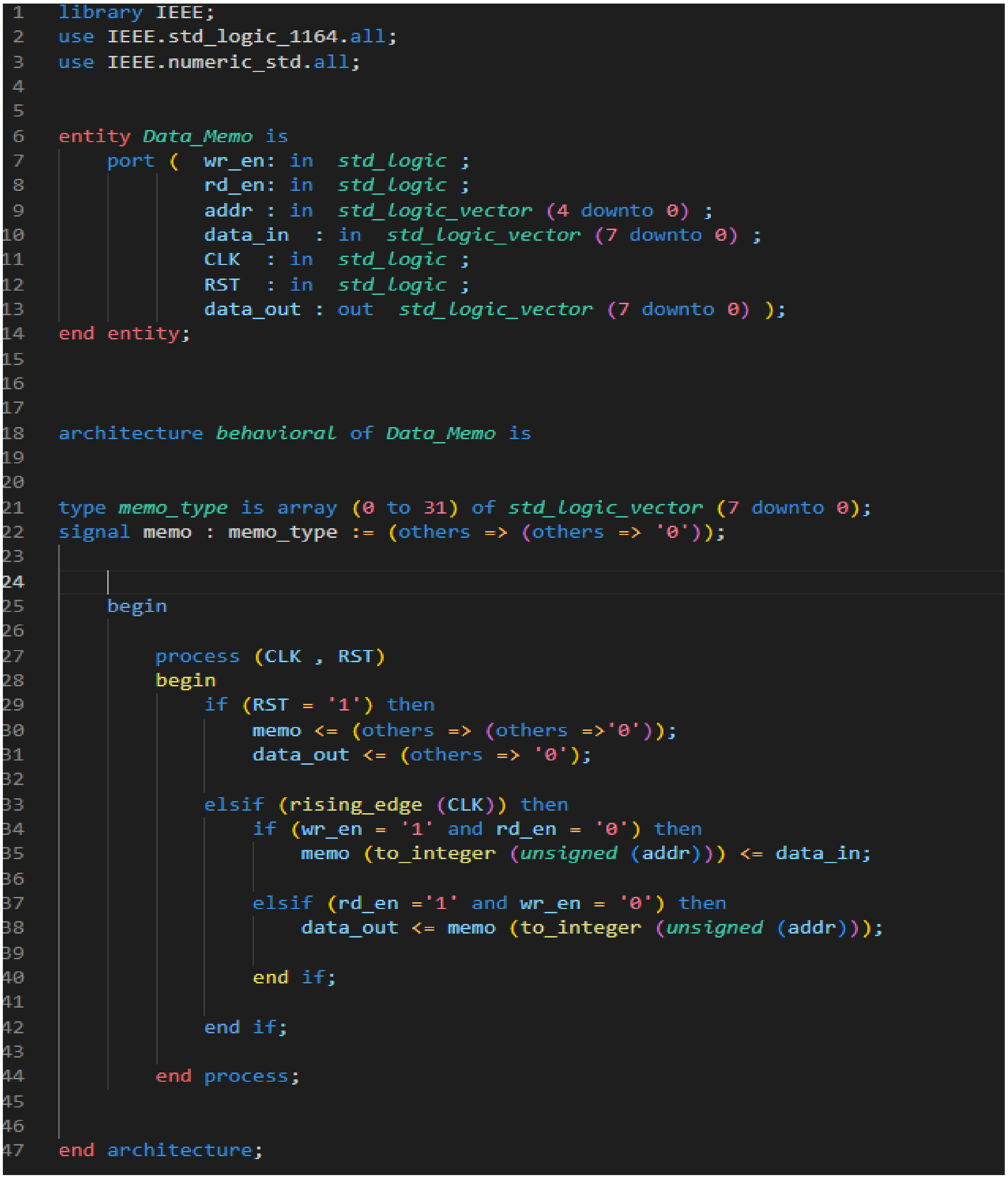
```
entity ALU is
    port (
        A           : in  std_logic_vector(7 downto 0);
        B           : in  std_logic_vector(7 downto 0);
        ALU_SEL     : in  std_logic_vector(2 downto 0); -- 000=ADD, 00

        ALU_OUT     : out std_logic_vector(7 downto 0);
        CarryOut    : out std_logic;
        Parity_Out  : out std_logic; -- renamed from Parity_Flag to av
        Overflow    : out std_logic
    );
end entity ALU;

architecture Structural of ALU is

    ------------------------------------------------------------------
    -- Component Declarations
    ------------------------------------------------------------------
    component Eight_Bit_Full_Adder
        port (
            a, b          : in  std_logic_vector(7 downto 0);
            sum           : out std_logic_vector(7 downto 0);
            cout          : out std_logic;
            Overflow_Flag : out std_logic
        );
    end component;

    component Eight_Bit_Subtractor
        port (
            a, b          : in  std_logic_vector(7 downto 0);
            sub           : out std_logic_vector(7 downto 0);
            cout          : out std_logic;
            Overflow_Flag : out std_logic
        );
    end component;

    component Eight_Bit_Multiplier
        port (
            a, b          : in  std_logic_vector(7 downto 0);
            product       : out std_logic_vector(15 downto 0);
            Overflow_Flag : out std_logic
        );
    end component;

    component Eight_Bit_AND
        port (
            A, B : in  std_logic_vector(7 downto 0);
            C    : out std_logic_vector(7 downto 0)
        );
    end component;
```



```
------------------------------------------------------------------
-- ALU Operation Selection
------------------------------------------------------------------
process(ALU_SEL, add_res, sub_res, mul_res, and_res, or_res, xor_res,
        add_cout, sub_cout, add_overflow, sub_overflow, mul_overflow)
begin
    -- Default outputs
    result   <= (others => '0');
    CarryOut <= '0';
    Overflow <= '0';

    case ALU_SEL is
        when "001" => -- ADD
            result   <= add_res;
            CarryOut <= add_cout;
            Overflow <= add_overflow;

        when "010" => -- SUB
            result   <= sub_res;
            CarryOut <= sub_cout;
            Overflow <= sub_overflow;

        when "011" => -- MUL
            result   <= mul_res(7 downto 0); -- lower 8 bits
            Overflow <= mul_overflow;

        when "100" => -- AND
            result   <= and_res;

        when "101" => -- OR
            result   <= or_res;

        when "110" => -- XOR
            result   <= xor_res;
        when others =>
            result   <= (others => '0');
    end case;
end process;

-- Output assignments
ALU_OUT    <= result;
Parity_Out <= parity_sig;
```

4.2 Data Memory

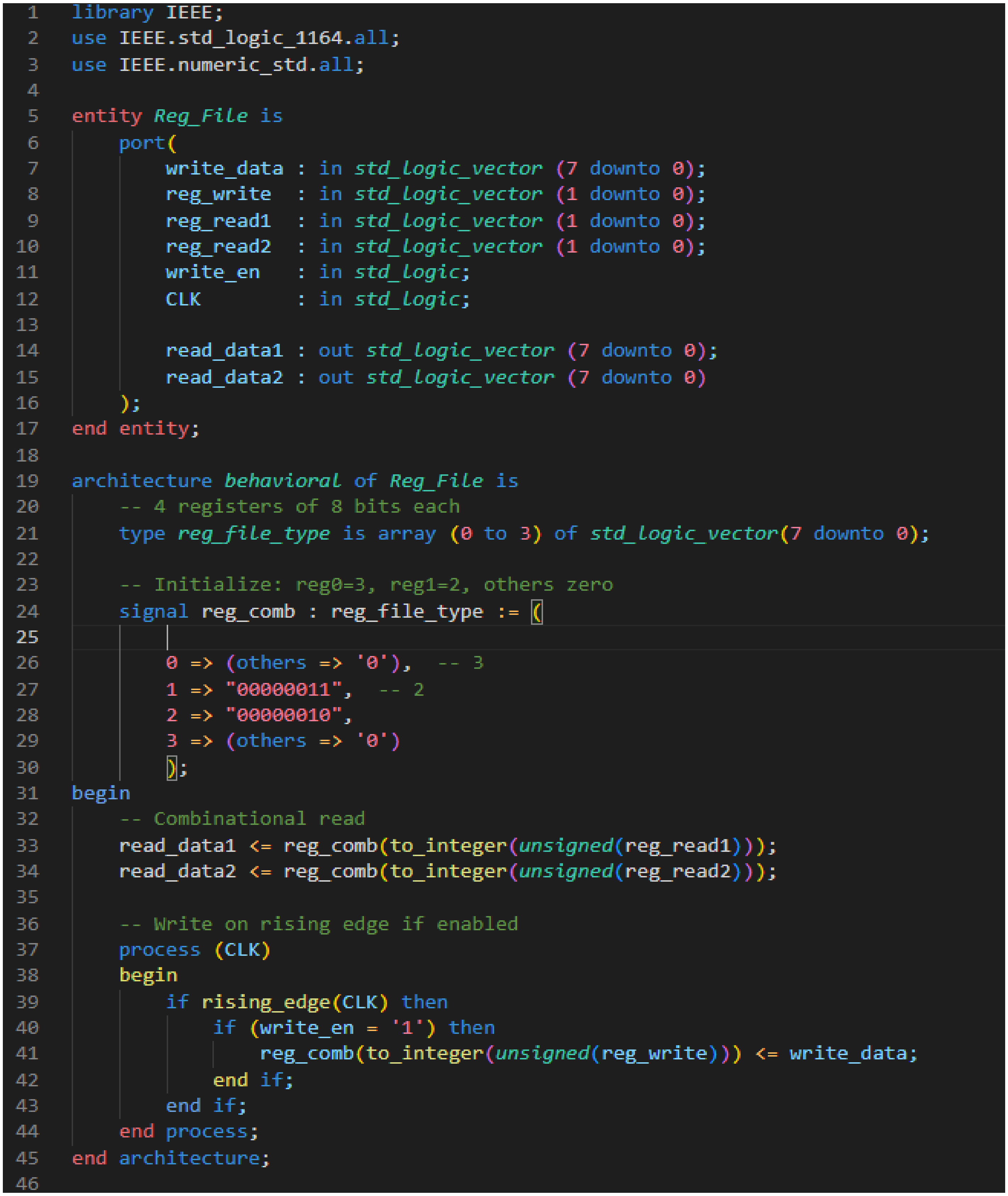
Code snippets



```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;


entity Data_Memo is
    port (  wr_en: in  std_logic ;
            rd_en: in  std_logic ;
            addr : in  std_logic_vector (4 downto 0) ;
            data_in  : in  std_logic_vector (7 downto 0) ;
            CLK  : in  std_logic ;
            RST  : in  std_logic ;
            data_out : out  std_logic_vector (7 downto 0) );
end entity;



architecture behavioral of Data_Memo is


type memo_type is array (0 to 31) of std_logic_vector (7 downto 0);
signal memo : memo_type := (others => (others => '0'));


    begin

        process (CLK , RST)
        begin
            if (RST = '1') then
                memo <= (others => (others =>'0'));
                data_out <= (others => '0');

            elsif (rising_edge (CLK)) then
                if (wr_en = '1' and rd_en = '0') then
                    memo (to_integer (unsigned (addr))) <= data_in;

                elsif (rd_en ='1' and wr_en = '0') then
                    data_out <= memo (to_integer (unsigned (addr)));

                end if;

            end if;

        end process;


end architecture;
```

4.3 Register File

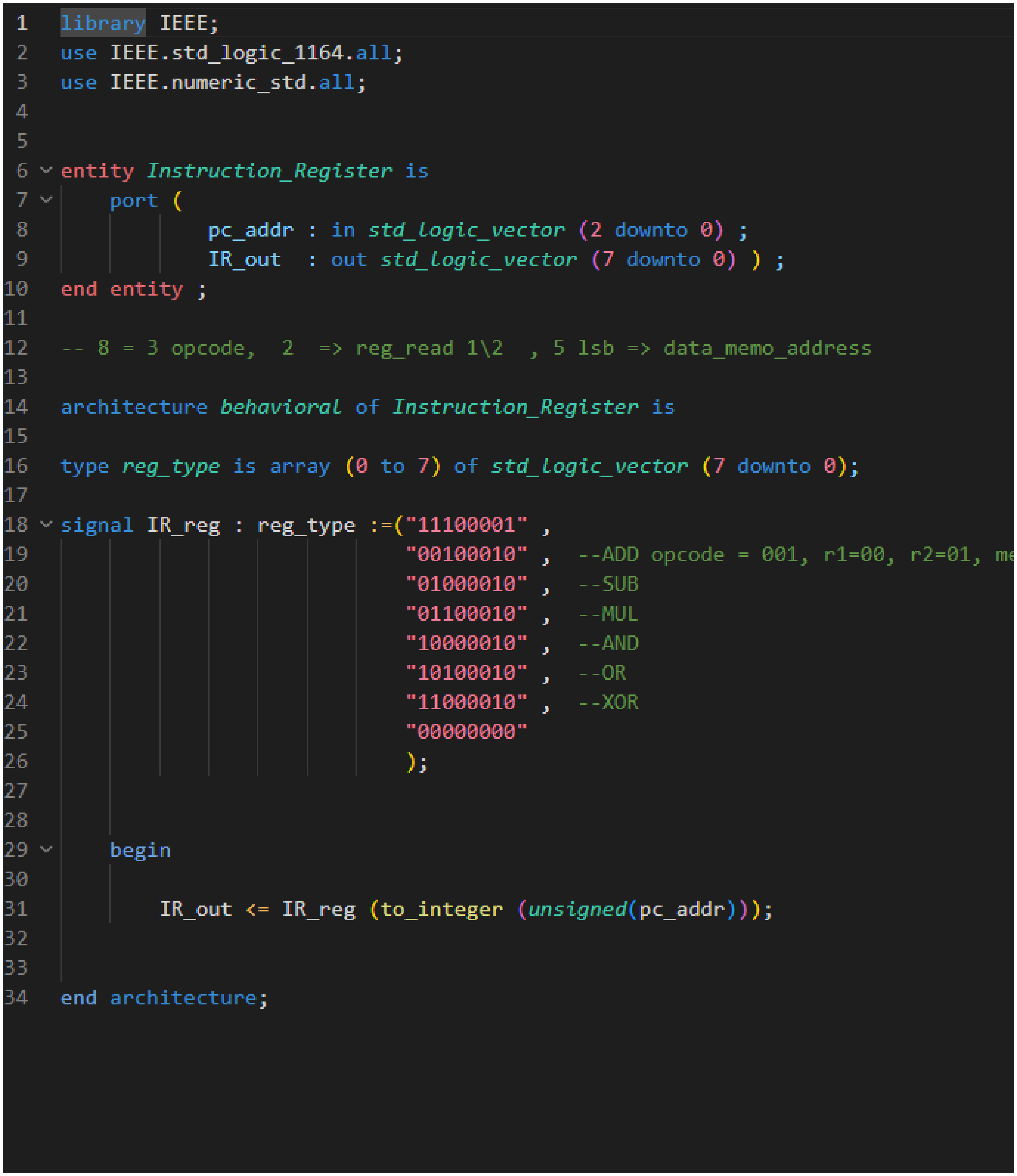
Code snippets



```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Reg_File is
    port(
        write_data : in std_logic_vector (7 downto 0);
        reg_write  : in std_logic_vector (1 downto 0);
        reg_read1  : in std_logic_vector (1 downto 0);
        reg_read2  : in std_logic_vector (1 downto 0);
        write_en   : in std_logic;
        CLK        : in std_logic;

        read_data1 : out std_logic_vector (7 downto 0);
        read_data2 : out std_logic_vector (7 downto 0)
    );
end entity;

architecture behavioral of Reg_File is
    -- 4 registers of 8 bits each
    type reg_file_type is array (0 to 3) of std_logic_vector(7 downto 0);

    -- Initialize: reg0=3, reg1=2, others zero
    signal reg_comb : reg_file_type := (

        0 => (others => '0'),  -- 3
        1 => "00000011",   -- 2
        2 => "00000010",
        3 => (others => '0')
        );
begin
    -- Combinational read
    read_data1 <= reg_comb(to_integer(unsigned(reg_read1)));
    read_data2 <= reg_comb(to_integer(unsigned(reg_read2)));

    -- Write on rising edge if enabled
    process (CLK)
    begin
        if rising_edge(CLK) then
            if (write_en = '1') then
                reg_comb(to_integer(unsigned(reg_write))) <= write_data;
            end if;
        end if;
    end process;
end architecture;

```

4.4 Instruction Register

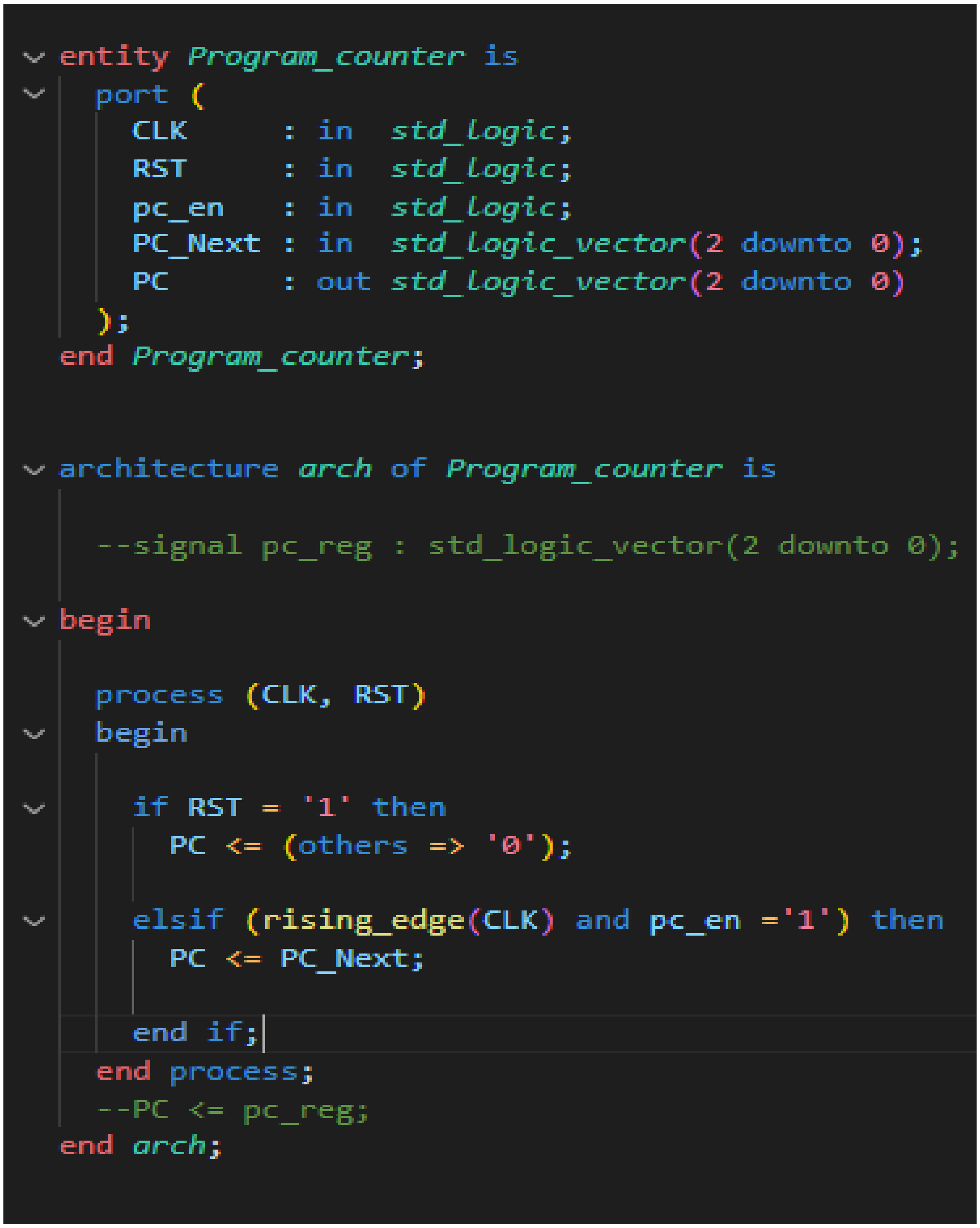
Code snippets



```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;


entity Instruction_Register is
    port (
            pc_addr : in std_logic_vector (2 downto 0) ;
            IR_out  : out std_logic_vector (7 downto 0) ) ;
end entity ;

-- 8 = 3 opcode,  2  => reg_read 1\2  , 5 lsb => data_memo_address

architecture behavioral of Instruction_Register is

type reg_type is array (0 to 7) of std_logic_vector (7 downto 0);

signal IR_reg : reg_type :=("11100001" ,
                            "00100010" ,  --ADD opcode = 001, r1=00, r2=01, me
                            "01000010" ,  --SUB
                            "01100010" ,  --MUL
                            "10000010" ,  --AND
                            "10100010" ,  --OR
                            "11000010" ,  --XOR
                            "00000000"
                            );


    begin

        IR_out <= IR_reg (to_integer (unsigned(pc_addr)));


end architecture;
```

4.5 Program Counter

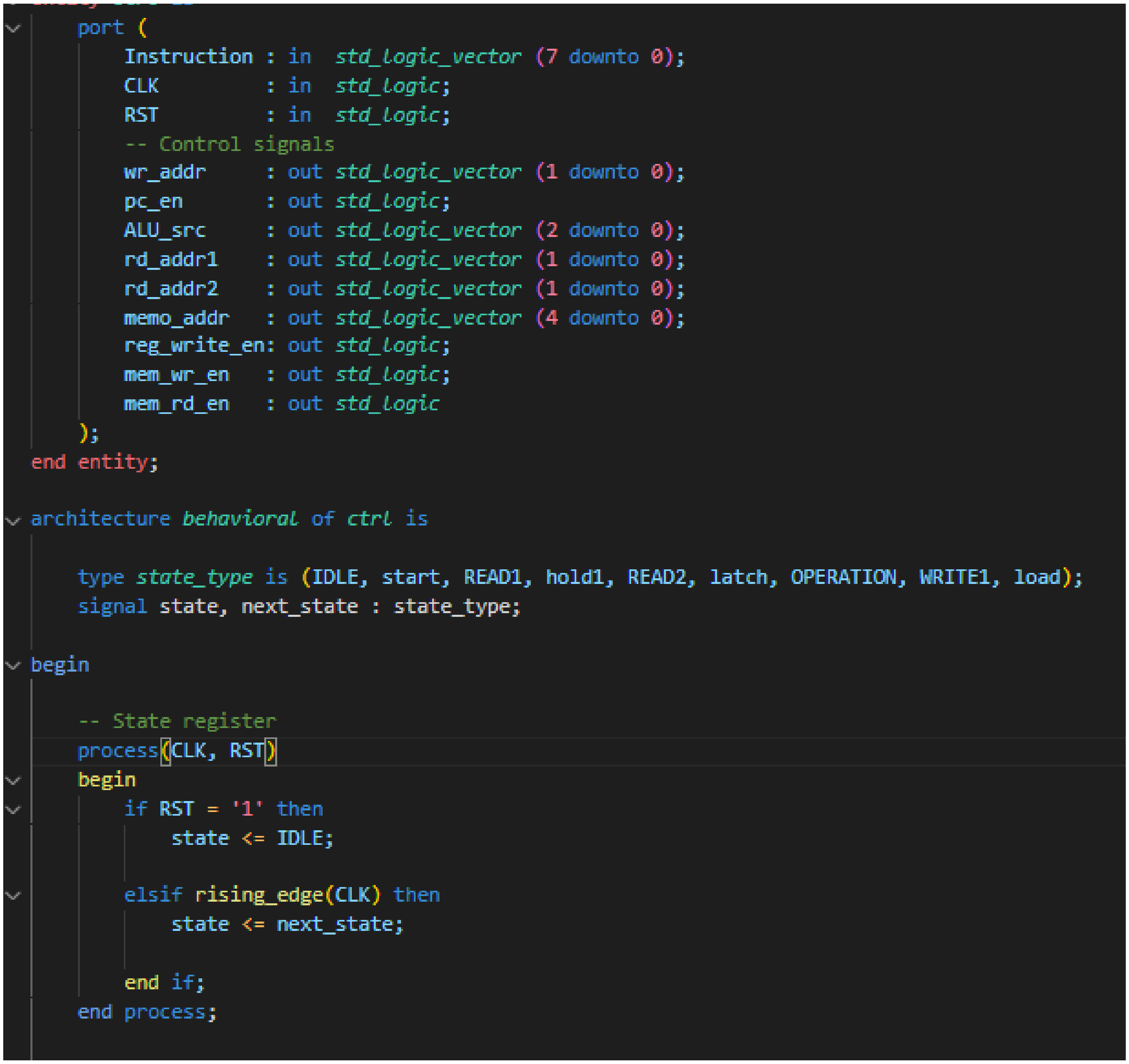
Code snippets



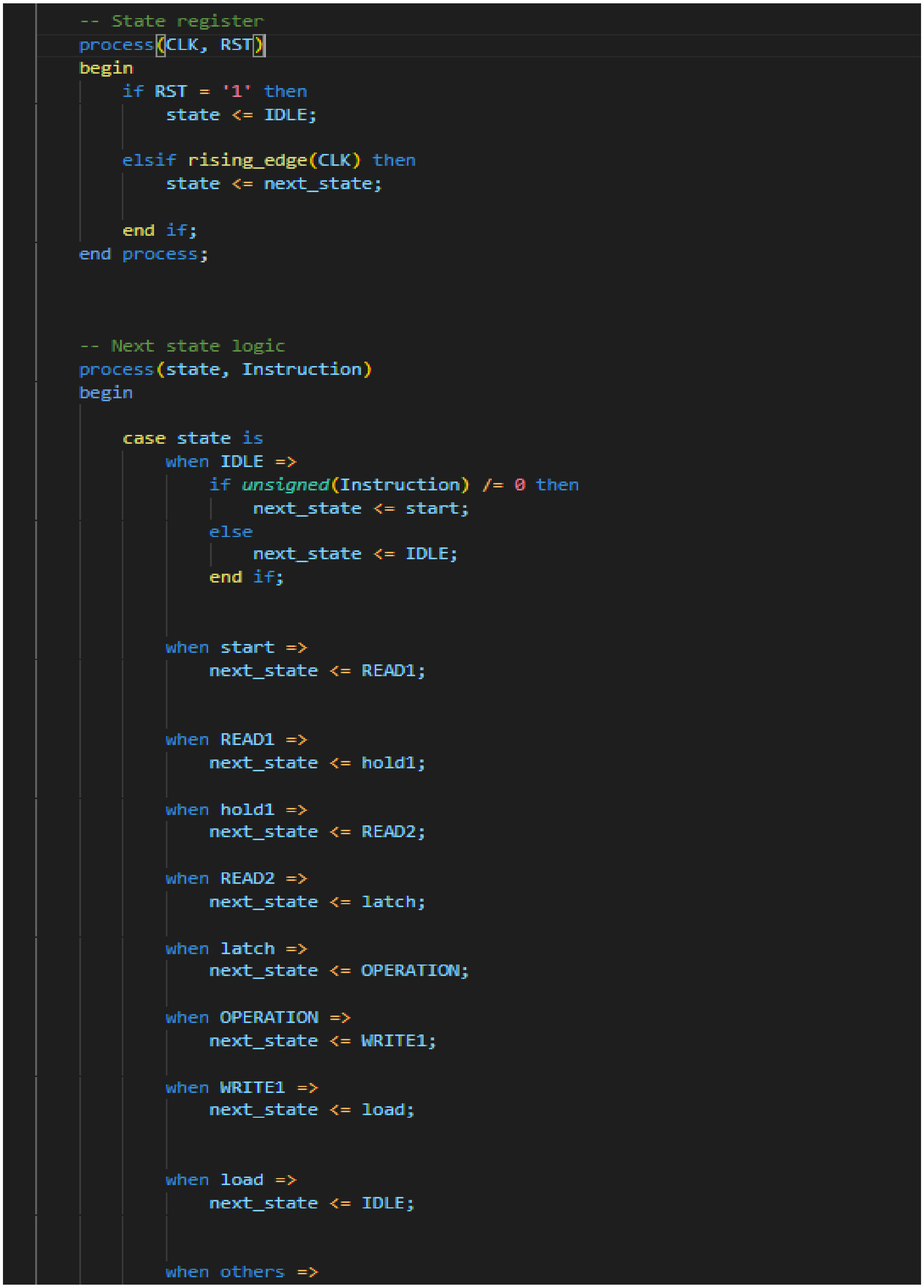
```
entity Program_counter is
  port (
    CLK     : in  std_logic;
    RST     : in  std_logic;
    pc_en   : in  std_logic;
    PC_Next : in  std_logic_vector(2 downto 0);
    PC      : out std_logic_vector(2 downto 0)
  );
end Program_counter;


architecture arch of Program_counter is

  --signal pc_reg : std_logic_vector(2 downto 0);

begin

  process (CLK, RST)
  begin

    if RST = '1' then
      PC <= (others => '0');

    elsif (rising_edge(CLK) and pc_en ='1') then
      PC <= PC_Next;

    end if;
  end process;
  --PC <= pc_reg;
end arch;
```

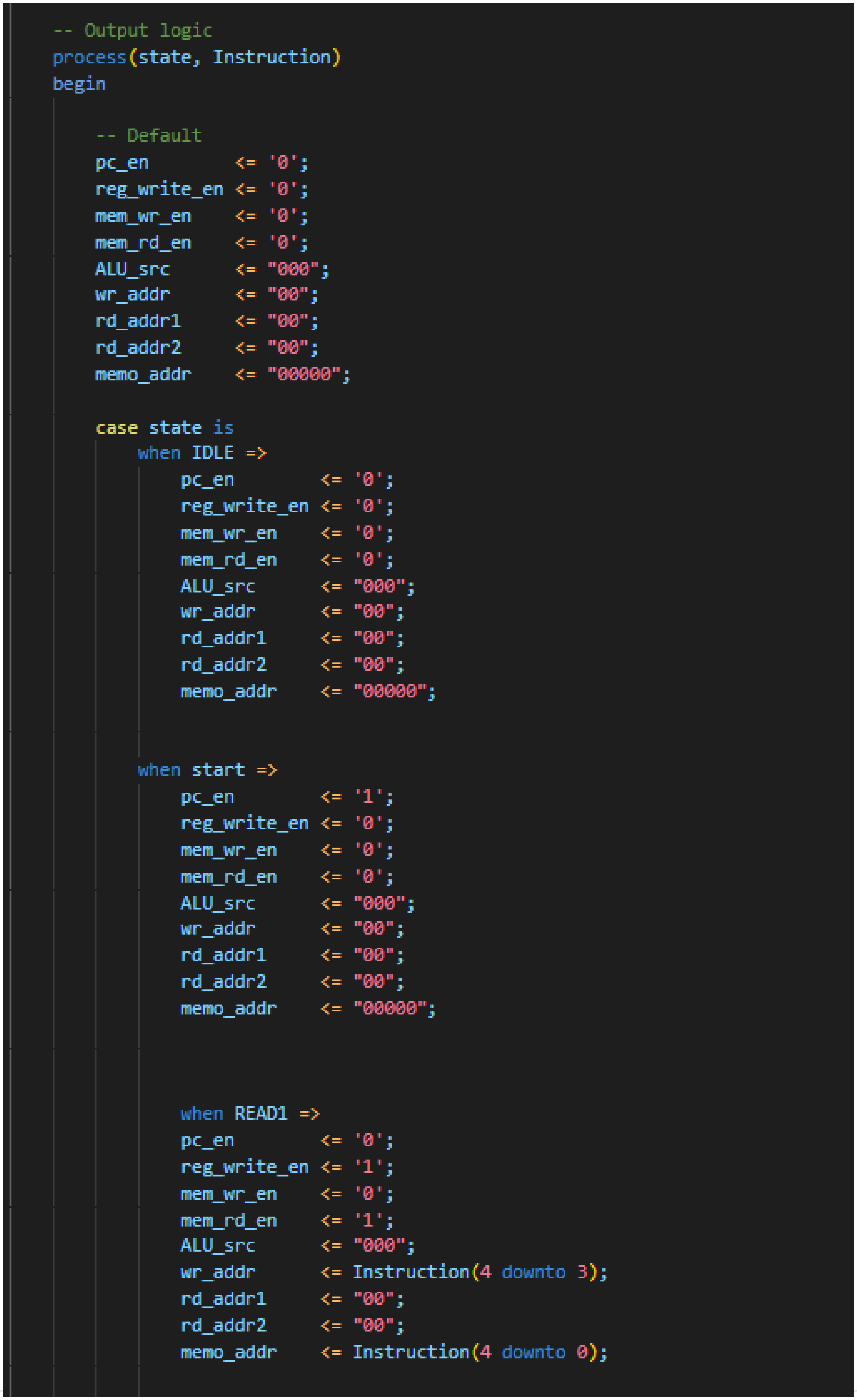
4.6 Controller

Code snippets

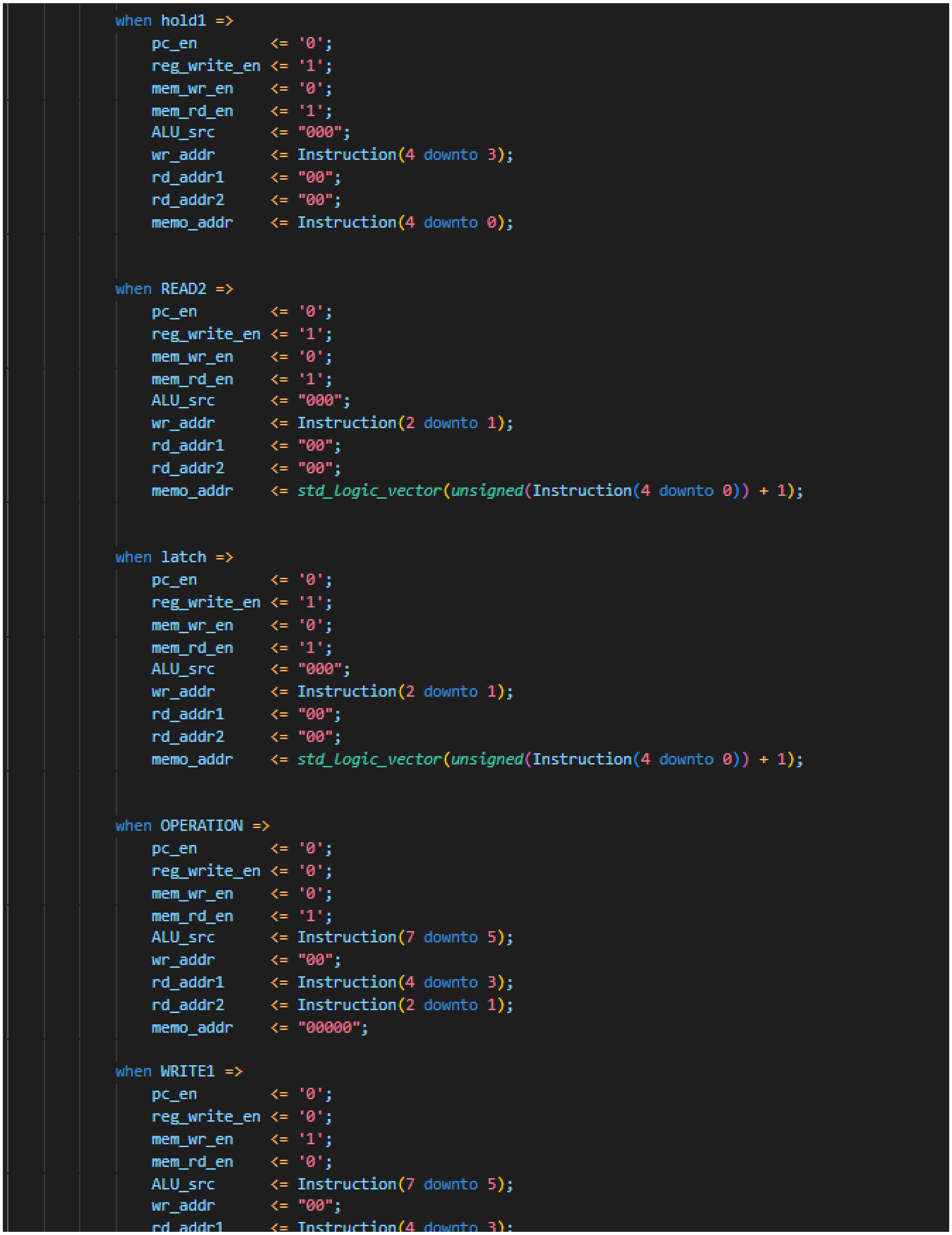


```
    port (
        Instruction : in  std_logic_vector (7 downto 0);
        CLK         : in  std_logic;
        RST         : in  std_logic;
        -- Control signals
        wr_addr     : out std_logic_vector (1 downto 0);
        pc_en       : out std_logic;
        ALU_src     : out std_logic_vector (2 downto 0);
        rd_addr1    : out std_logic_vector (1 downto 0);
        rd_addr2    : out std_logic_vector (1 downto 0);
        memo_addr   : out std_logic_vector (4 downto 0);
        reg_write_en: out std_logic;
        mem_wr_en   : out std_logic;
        mem_rd_en   : out std_logic
    );
end entity;

architecture behavioral of ctrl is

    type state_type is (IDLE, start, READ1, hold1, READ2, latch, OPERATION, WRITE1, load);
    signal state, next_state : state_type;

begin

    -- State register
    process(CLK, RST)
    begin
        if RST = '1' then
            state <= IDLE;

        elsif rising_edge(CLK) then
            state <= next_state;

        end if;
    end process;
```

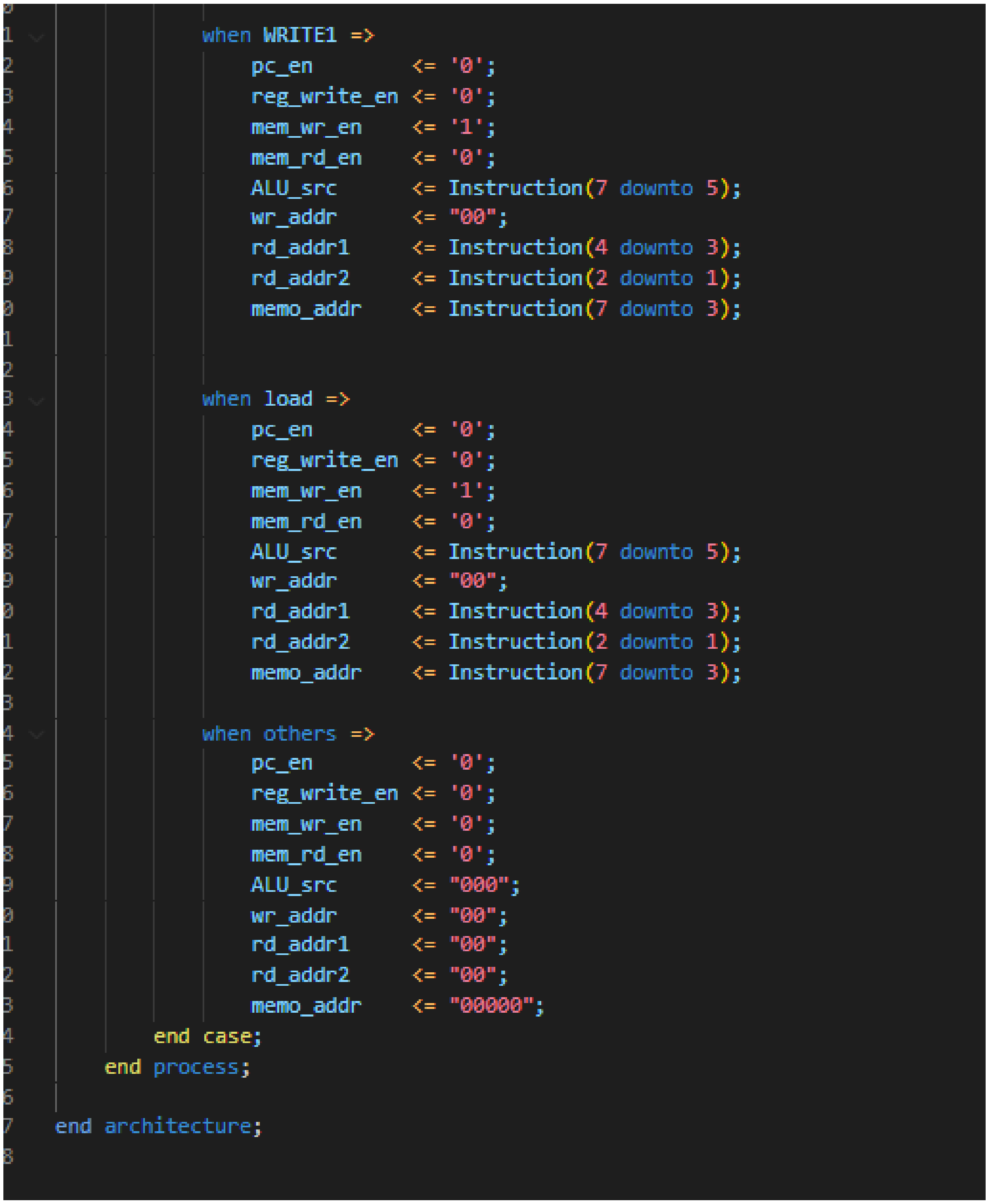

```vhdl
    -- State register
    process(CLK, RST)
    begin
        if RST = '1' then
            state <= IDLE;

        elsif rising_edge(CLK) then
            state <= next_state;

        end if;
    end process;



    -- Next state logic
    process(state, Instruction)
    begin

        case state is
            when IDLE =>
                if unsigned(Instruction) /= 0 then
                    next_state <= start;
                else
                    next_state <= IDLE;
                end if;


            when start =>
                next_state <= READ1;


            when READ1 =>
                next_state <= hold1;

            when hold1 =>
                next_state <= READ2;

            when READ2 =>
                next_state <= latch;

            when latch =>
                next_state <= OPERATION;

            when OPERATION =>
                next_state <= WRITE1;

            when WRITE1 =>
                next_state <= load;


            when load =>
                next_state <= IDLE;


            when others =>
```



```
-- Output logic
process(state, Instruction)
begin

    -- Default
    pc_en        <= '0';
    reg_write_en <= '0';
    mem_wr_en    <= '0';
    mem_rd_en    <= '0';
    ALU_src      <= "000";
    wr_addr      <= "00";
    rd_addr1     <= "00";
    rd_addr2     <= "00";
    memo_addr    <= "00000";

    case state is
        when IDLE =>
            pc_en        <= '0';
            reg_write_en <= '0';
            mem_wr_en    <= '0';
            mem_rd_en    <= '0';
            ALU_src      <= "000";
            wr_addr      <= "00";
            rd_addr1     <= "00";
            rd_addr2     <= "00";
            memo_addr    <= "00000";


        when start =>
            pc_en        <= '1';
            reg_write_en <= '0';
            mem_wr_en    <= '0';
            mem_rd_en    <= '0';
            ALU_src      <= "000";
            wr_addr      <= "00";
            rd_addr1     <= "00";
            rd_addr2     <= "00";
            memo_addr    <= "00000";



            when READ1 =>
            pc_en        <= '0';
            reg_write_en <= '1';
            mem_wr_en    <= '0';
            mem_rd_en    <= '1';
            ALU_src      <= "000";
            wr_addr      <= Instruction(4 downto 3);
            rd_addr1     <= "00";
            rd_addr2     <= "00";
            memo_addr    <= Instruction(4 downto 0);
```



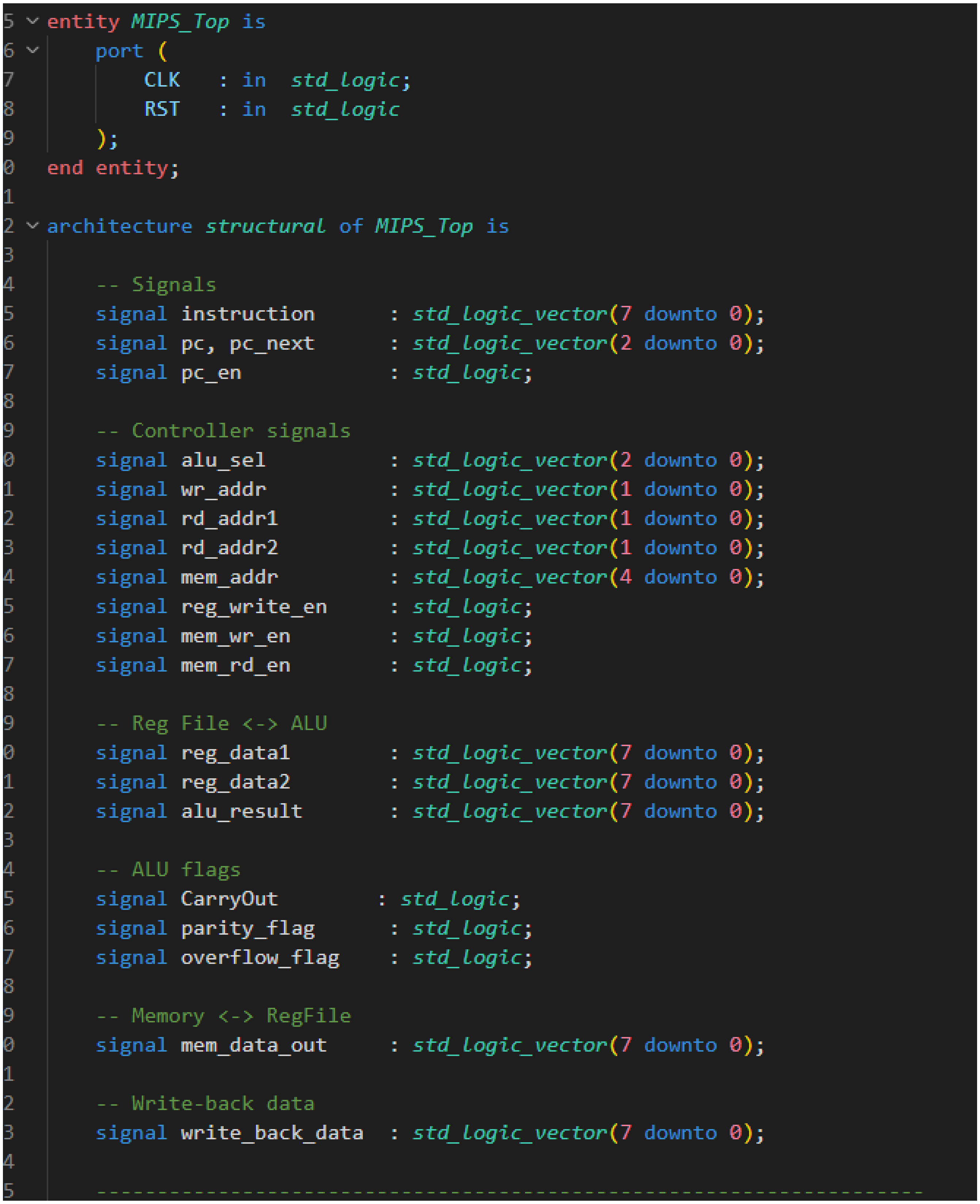
```
            when hold1 =>
                pc_en        <= '0';
                reg_write_en <= '1';
                mem_wr_en    <= '0';
                mem_rd_en    <= '1';
                ALU_src      <= "000";
                wr_addr      <= Instruction(4 downto 3);
                rd_addr1     <= "00";
                rd_addr2     <= "00";
                memo_addr    <= Instruction(4 downto 0);


            when READ2 =>
                pc_en        <= '0';
                reg_write_en <= '1';
                mem_wr_en    <= '0';
                mem_rd_en    <= '1';
                ALU_src      <= "000";
                wr_addr      <= Instruction(2 downto 1);
                rd_addr1     <= "00";
                rd_addr2     <= "00";
                memo_addr    <= std_logic_vector(unsigned(Instruction(4 downto 0)) + 1);


            when latch =>
                pc_en        <= '0';
                reg_write_en <= '1';
                mem_wr_en    <= '0';
                mem_rd_en    <= '1';
                ALU_src      <= "000";
                wr_addr      <= Instruction(2 downto 1);
                rd_addr1     <= "00";
                rd_addr2     <= "00";
                memo_addr    <= std_logic_vector(unsigned(Instruction(4 downto 0)) + 1);


            when OPERATION =>
                pc_en        <= '0';
                reg_write_en <= '0';
                mem_wr_en    <= '0';
                mem_rd_en    <= '1';
                ALU_src      <= Instruction(7 downto 5);
                wr_addr      <= "00";
                rd_addr1     <= Instruction(4 downto 3);
                rd_addr2     <= Instruction(2 downto 1);
                memo_addr    <= "00000";

            when WRITE1 =>
                pc_en        <= '0';
                reg_write_en <= '0';
                mem_wr_en    <= '1';
                mem_rd_en    <= '0';
                ALU_src      <= Instruction(7 downto 5);
                wr_addr      <= "00";
                rd_addr1     <= Instruction(4 downto 3);
```



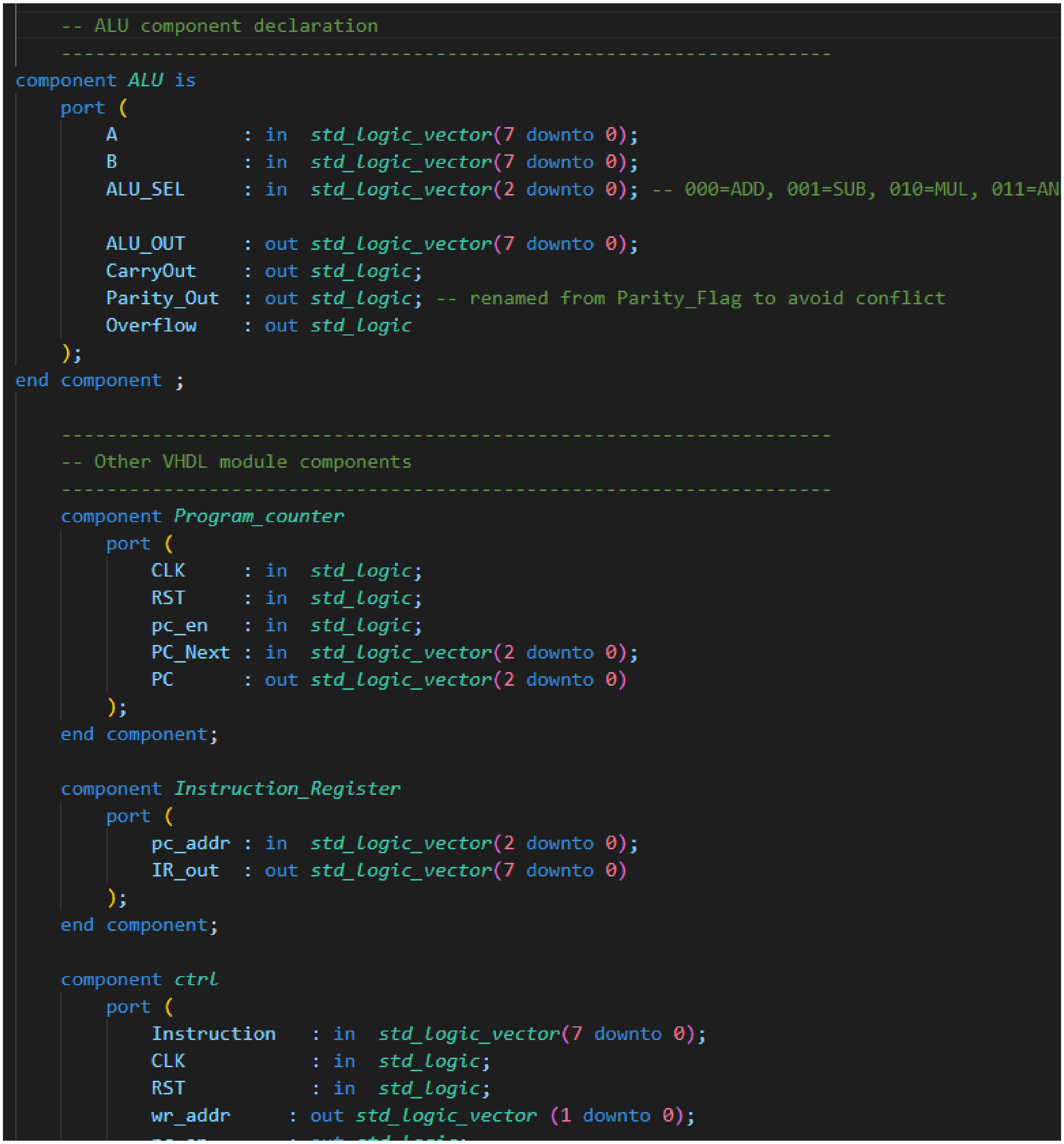
```vhdl
            when WRITE1 =>
                pc_en        <= '0';
                reg_write_en <= '0';
                mem_wr_en    <= '1';
                mem_rd_en    <= '0';
                ALU_src      <= Instruction(7 downto 5);
                wr_addr      <= "00";
                rd_addr1     <= Instruction(4 downto 3);
                rd_addr2     <= Instruction(2 downto 1);
                memo_addr    <= Instruction(7 downto 3);


            when load =>
                pc_en        <= '0';
                reg_write_en <= '0';
                mem_wr_en    <= '1';
                mem_rd_en    <= '0';
                ALU_src      <= Instruction(7 downto 5);
                wr_addr      <= "00";
                rd_addr1     <= Instruction(4 downto 3);
                rd_addr2     <= Instruction(2 downto 1);
                memo_addr    <= Instruction(7 downto 3);

            when others =>
                pc_en        <= '0';
                reg_write_en <= '0';
                mem_wr_en    <= '0';
                mem_rd_en    <= '0';
                ALU_src      <= "000";
                wr_addr      <= "00";
                rd_addr1     <= "00";
                rd_addr2     <= "00";
                memo_addr    <= "00000";
        end case;
    end process;

end architecture;
```

4.7 Top Module

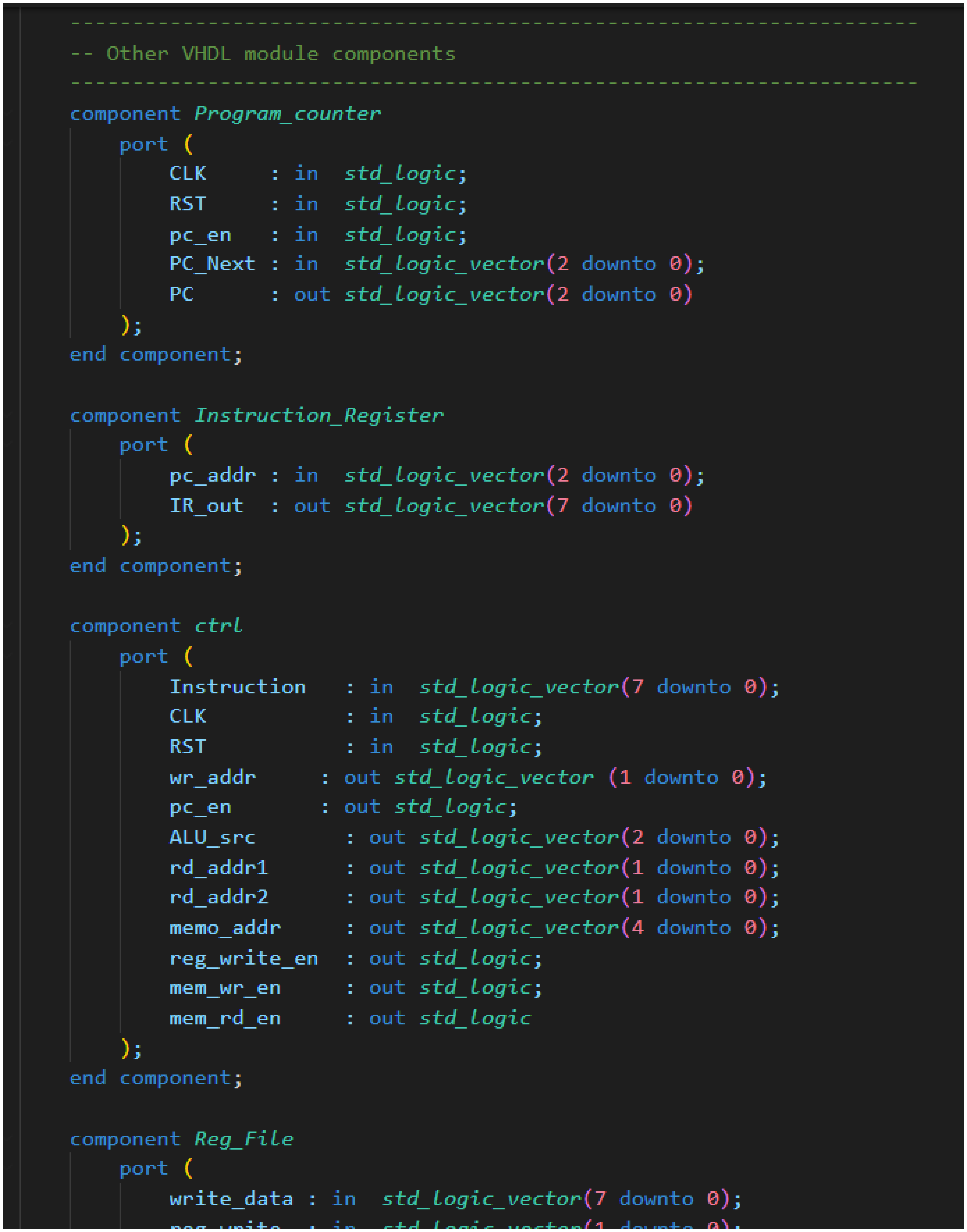
Code snippets



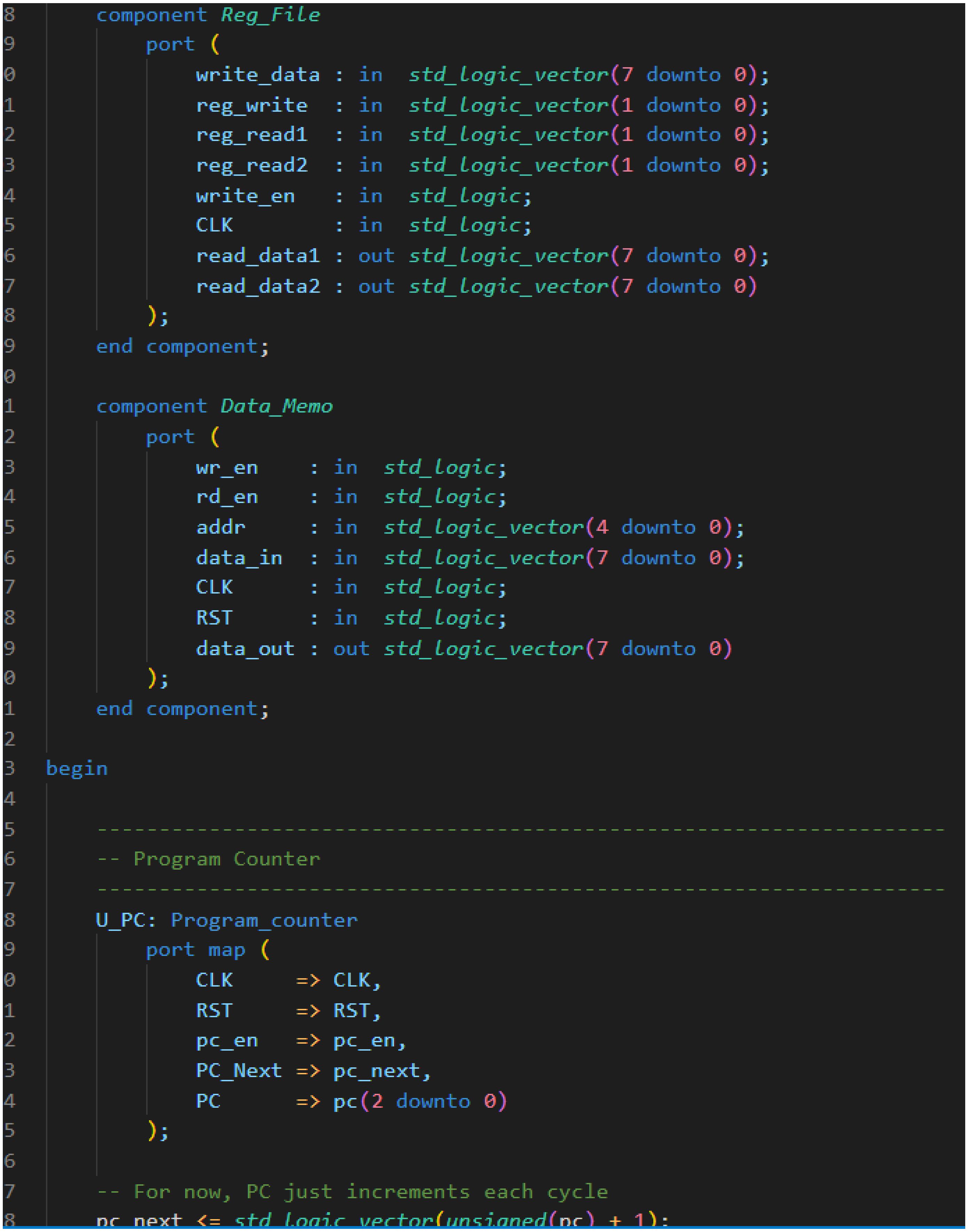
```vhdl
entity MIPS_Top is
    port (
        CLK    : in  std_logic;
        RST    : in  std_logic
    );
end entity;

architecture structural of MIPS_Top is

    -- Signals
    signal instruction      : std_logic_vector(7 downto 0);
    signal pc, pc_next      : std_logic_vector(2 downto 0);
    signal pc_en            : std_logic;

    -- Controller signals
    signal alu_sel          : std_logic_vector(2 downto 0);
    signal wr_addr          : std_logic_vector(1 downto 0);
    signal rd_addr1         : std_logic_vector(1 downto 0);
    signal rd_addr2         : std_logic_vector(1 downto 0);
    signal mem_addr         : std_logic_vector(4 downto 0);
    signal reg_write_en     : std_logic;
    signal mem_wr_en        : std_logic;
    signal mem_rd_en        : std_logic;

    -- Reg File <-> ALU
    signal reg_data1        : std_logic_vector(7 downto 0);
    signal reg_data2        : std_logic_vector(7 downto 0);
    signal alu_result       : std_logic_vector(7 downto 0);

    -- ALU flags
    signal CarryOut        : std_logic;
    signal parity_flag      : std_logic;
    signal overflow_flag    : std_logic;

    -- Memory <-> RegFile
    signal mem_data_out     : std_logic_vector(7 downto 0);

    -- Write-back data
    signal write_back_data  : std_logic_vector(7 downto 0);

    ---------------------------------------------------------------------
```



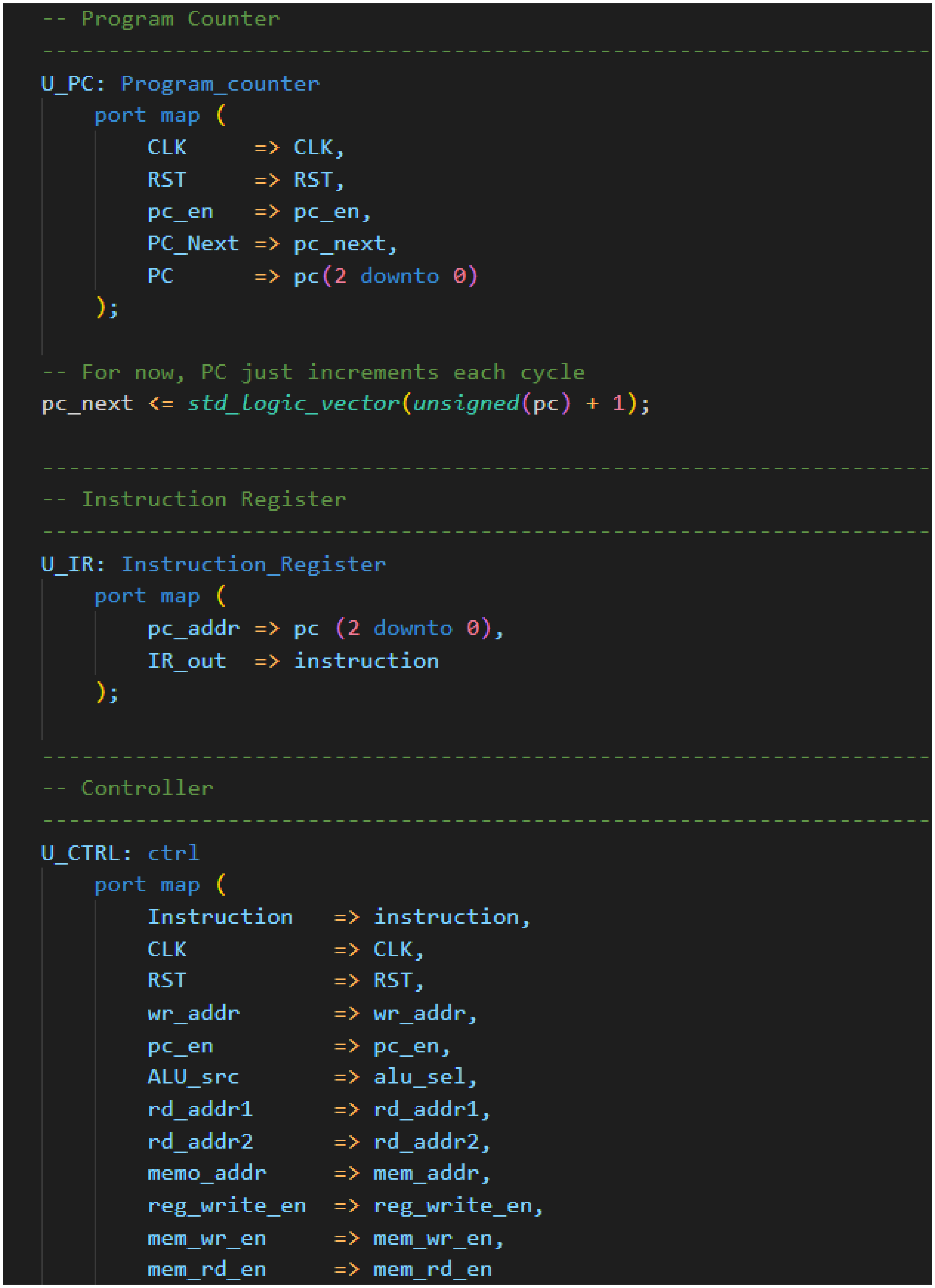
```vhdl
    -- ALU component declaration
    --------------------------------------------------------------------
component ALU is
    port (
        A           : in  std_logic_vector(7 downto 0);
        B           : in  std_logic_vector(7 downto 0);
        ALU_SEL     : in  std_logic_vector(2 downto 0); -- 000=ADD, 001=SUB, 010=MUL, 011=AN

        ALU_OUT     : out std_logic_vector(7 downto 0);
        CarryOut    : out std_logic;
        Parity_Out  : out std_logic; -- renamed from Parity_Flag to avoid conflict
        Overflow    : out std_logic
    );
end component ;

    --------------------------------------------------------------------
    -- Other VHDL module components
    --------------------------------------------------------------------
    component Program_counter
        port (
            CLK     : in  std_logic;
            RST     : in  std_logic;
            pc_en   : in  std_logic;
            PC_Next : in  std_logic_vector(2 downto 0);
            PC      : out std_logic_vector(2 downto 0)
        );
    end component;

    component Instruction_Register
        port (
            pc_addr : in  std_logic_vector(2 downto 0);
            IR_out  : out std_logic_vector(7 downto 0)
        );
    end component;

    component ctrl
        port (
            Instruction   : in  std_logic_vector(7 downto 0);
            CLK           : in  std_logic;
            RST           : in  std_logic;
            wr_addr     : out std_logic_vector (1 downto 0);
            pc_en       : out std_logic;
```



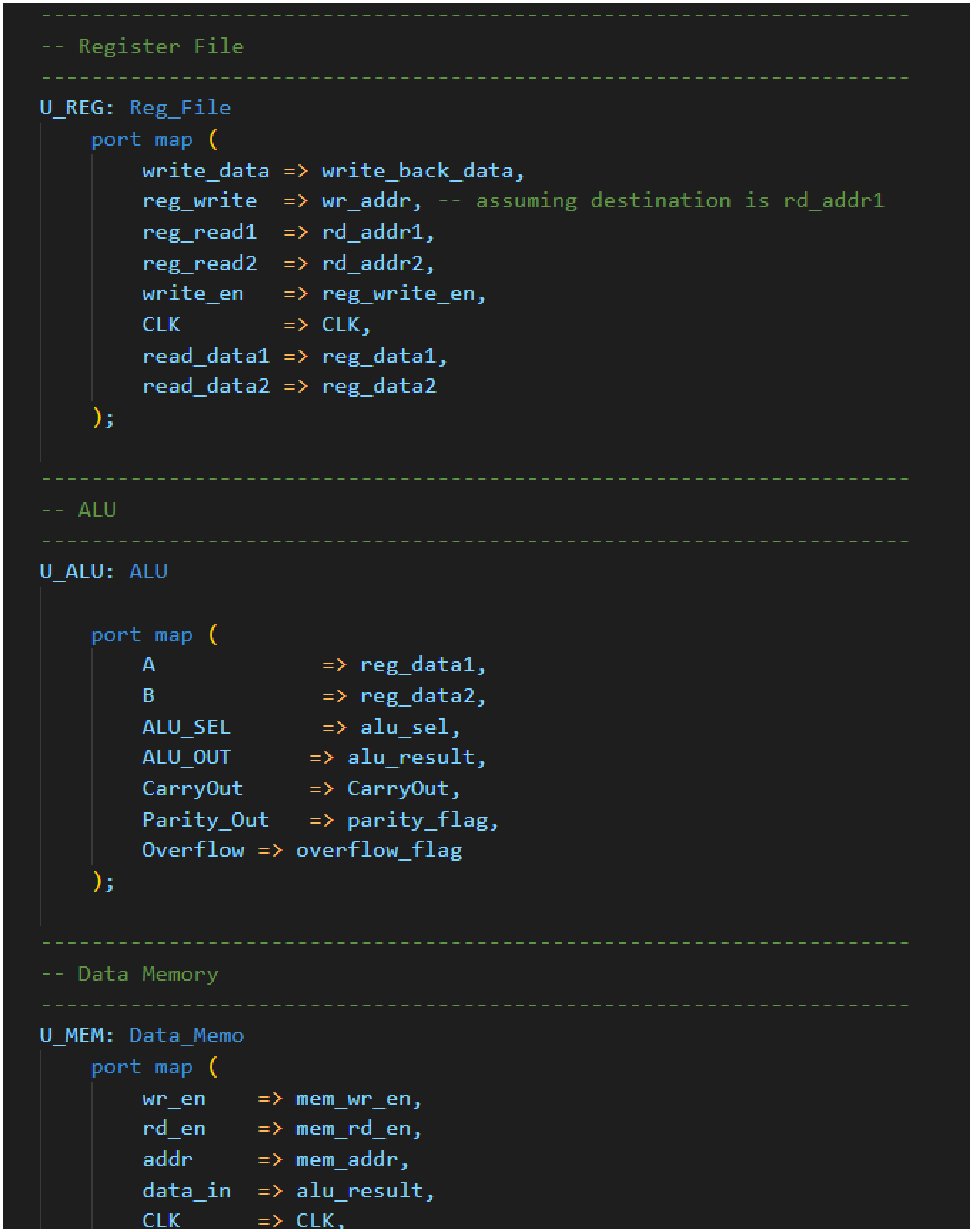
```
    ---------------------------------------------------------------------
    -- Other VHDL module components
    ---------------------------------------------------------------------
    component Program_counter
        port (
            CLK     : in  std_logic;
            RST     : in  std_logic;
            pc_en   : in  std_logic;
            PC_Next : in  std_logic_vector(2 downto 0);
            PC      : out std_logic_vector(2 downto 0)
        );
    end component;

    component Instruction_Register
        port (
            pc_addr : in  std_logic_vector(2 downto 0);
            IR_out  : out std_logic_vector(7 downto 0)
        );
    end component;

    component ctrl
        port (
            Instruction   : in  std_logic_vector(7 downto 0);
            CLK           : in  std_logic;
            RST           : in  std_logic;
            wr_addr     : out std_logic_vector (1 downto 0);
            pc_en       : out std_logic;
            ALU_src       : out std_logic_vector(2 downto 0);
            rd_addr1      : out std_logic_vector(1 downto 0);
            rd_addr2      : out std_logic_vector(1 downto 0);
            memo_addr     : out std_logic_vector(4 downto 0);
            reg_write_en  : out std_logic;
            mem_wr_en     : out std_logic;
            mem_rd_en     : out std_logic
        );
    end component;

    component Reg_File
        port (
            write_data : in  std_logic_vector(7 downto 0);
            reg_write  : in  std_logic_vector(1 downto 0);
```



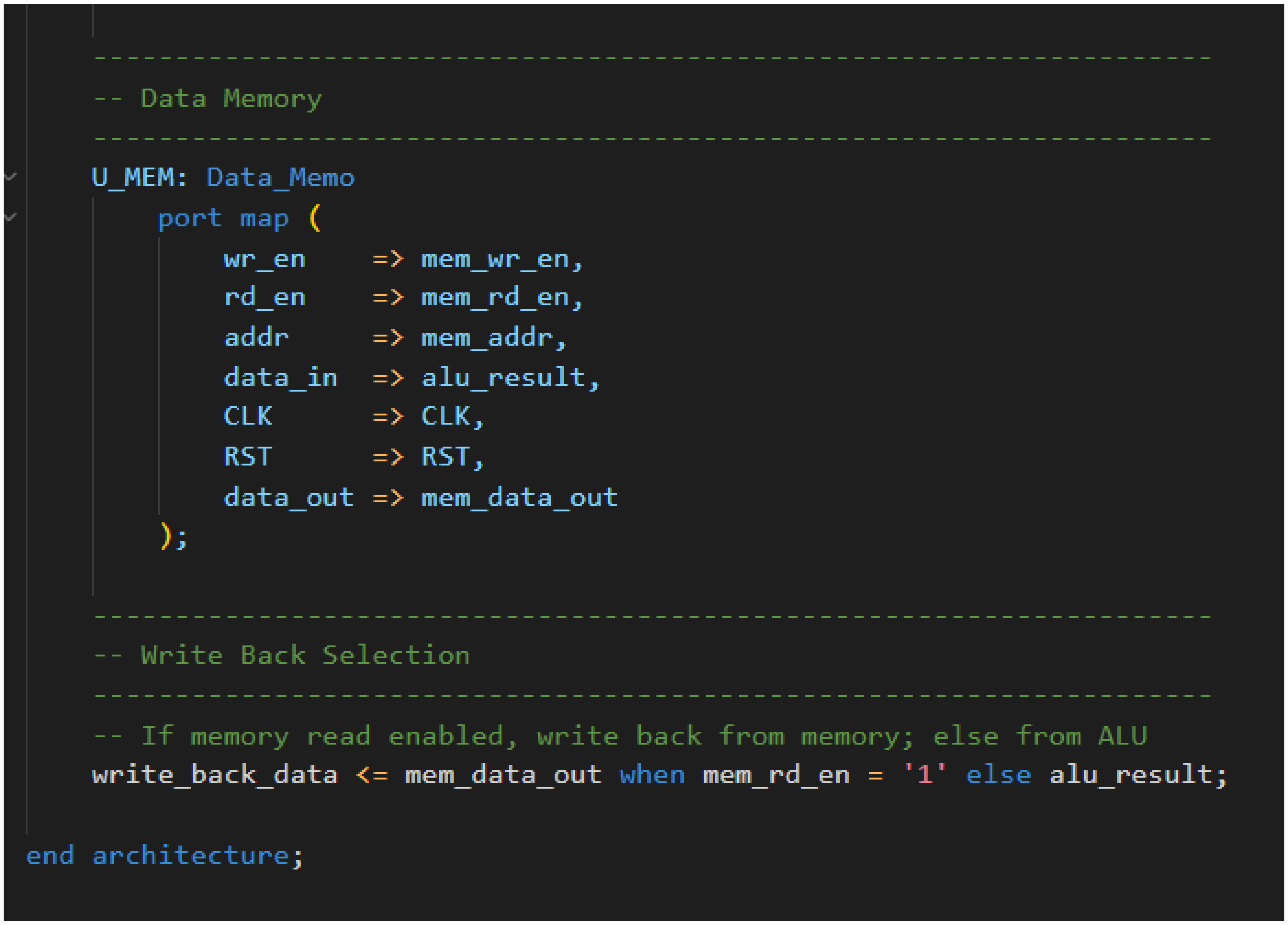
```vhdl
    component Reg_File
        port (
            write_data : in  std_logic_vector(7 downto 0);
            reg_write  : in  std_logic_vector(1 downto 0);
            reg_read1  : in  std_logic_vector(1 downto 0);
            reg_read2  : in  std_logic_vector(1 downto 0);
            write_en   : in  std_logic;
            CLK        : in  std_logic;
            read_data1 : out std_logic_vector(7 downto 0);
            read_data2 : out std_logic_vector(7 downto 0)
        );
    end component;

    component Data_Memo
        port (
            wr_en    : in  std_logic;
            rd_en    : in  std_logic;
            addr     : in  std_logic_vector(4 downto 0);
            data_in  : in  std_logic_vector(7 downto 0);
            CLK      : in  std_logic;
            RST      : in  std_logic;
            data_out : out std_logic_vector(7 downto 0)
        );
    end component;

begin

    --------------------------------------------------------------------
    -- Program Counter
    --------------------------------------------------------------------
    U_PC: Program_counter
        port map (
            CLK     => CLK,
            RST     => RST,
            pc_en   => pc_en,
            PC_Next => pc_next,
            PC      => pc(2 downto 0)
        );

    -- For now, PC just increments each cycle
    pc_next <= std_logic_vector(unsigned(pc) + 1);
```



```vhdl
-- Program Counter
-------------------------------------------------------------------
U_PC: Program_counter
    port map (
        CLK     => CLK,
        RST     => RST,
        pc_en   => pc_en,
        PC_Next => pc_next,
        PC      => pc(2 downto 0)
    );

-- For now, PC just increments each cycle
pc_next <= std_logic_vector(unsigned(pc) + 1);

-------------------------------------------------------------------
-- Instruction Register
-------------------------------------------------------------------
U_IR: Instruction_Register
    port map (
        pc_addr => pc (2 downto 0),
        IR_out  => instruction
    );

-------------------------------------------------------------------
-- Controller
-------------------------------------------------------------------
U_CTRL: ctrl
    port map (
        Instruction   => instruction,
        CLK           => CLK,
        RST           => RST,
        wr_addr       => wr_addr,
        pc_en         => pc_en,
        ALU_src       => alu_sel,
        rd_addr1      => rd_addr1,
        rd_addr2      => rd_addr2,
        memo_addr     => mem_addr,
        reg_write_en  => reg_write_en,
        mem_wr_en     => mem_wr_en,
        mem_rd_en     => mem_rd_en
```

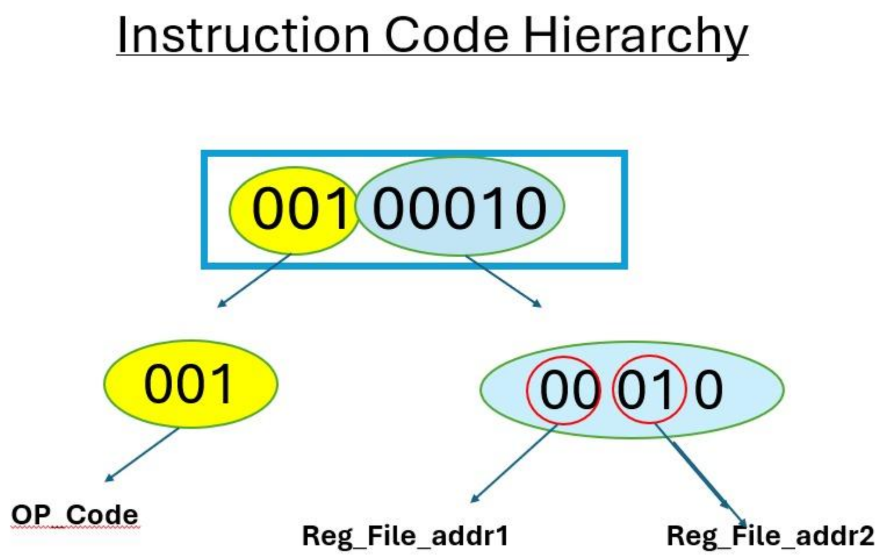


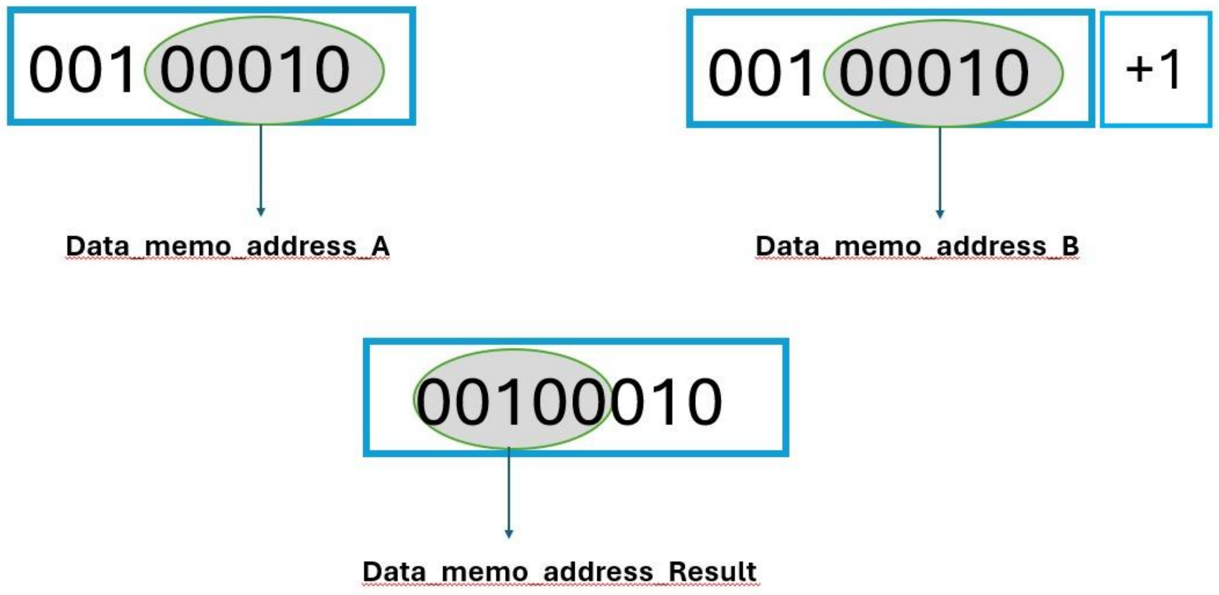
```
---------------------------------------------------------------------
-- Register File
---------------------------------------------------------------------
U_REG: Reg_File
    port map (
        write_data => write_back_data,
        reg_write  => wr_addr, -- assuming destination is rd_addr1
        reg_read1  => rd_addr1,
        reg_read2  => rd_addr2,
        write_en   => reg_write_en,
        CLK        => CLK,
        read_data1 => reg_data1,
        read_data2 => reg_data2
    );

---------------------------------------------------------------------
-- ALU
---------------------------------------------------------------------
U_ALU: ALU

    port map (
        A             => reg_data1,
        B             => reg_data2,
        ALU_SEL       => alu_sel,
        ALU_OUT      => alu_result,
        CarryOut     => CarryOut,
        Parity_Out   => parity_flag,
        Overflow => overflow_flag
    );

---------------------------------------------------------------------
-- Data Memory
---------------------------------------------------------------------
U_MEM: Data_Memo
    port map (
        wr_en    => mem_wr_en,
        rd_en    => mem_rd_en,
        addr     => mem_addr,
        data_in  => alu_result,
        CLK      => CLK,
```



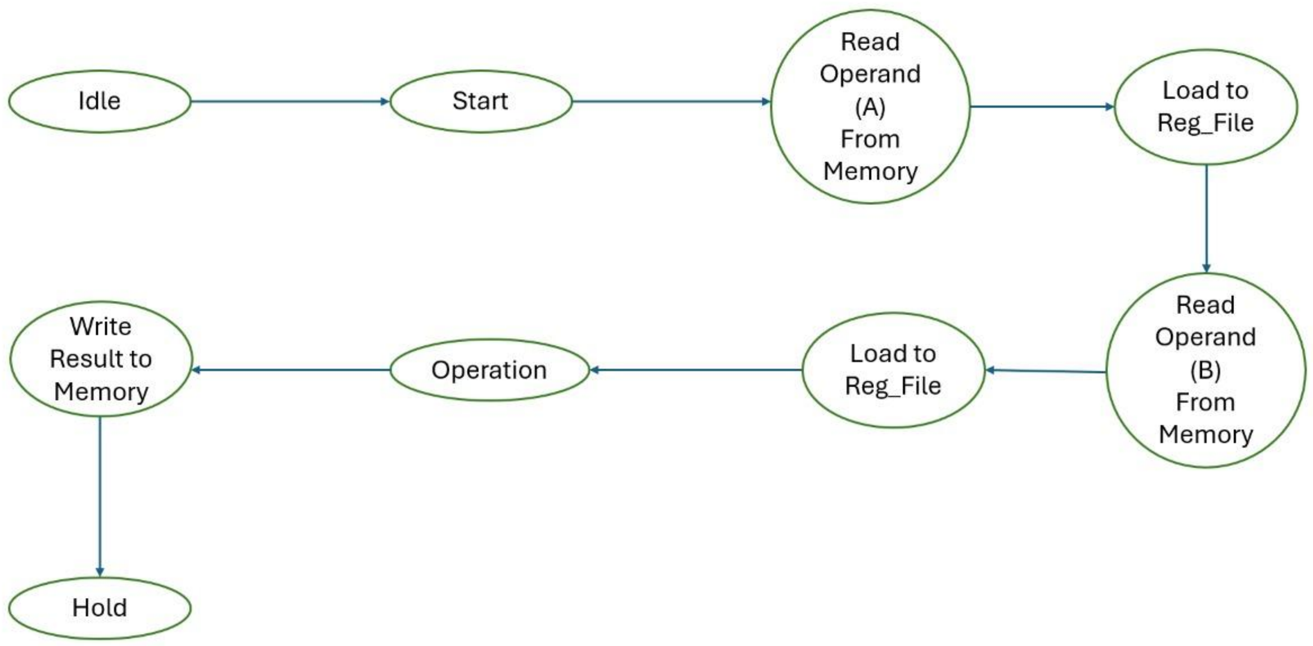
```vhdl

    --------------------------------------------------------------------
    -- Data Memory
    --------------------------------------------------------------------
    U_MEM: Data_Memo
        port map (
            wr_en    => mem_wr_en,
            rd_en    => mem_rd_en,
            addr     => mem_addr,
            data_in  => alu_result,
            CLK      => CLK,
            RST      => RST,
            data_out => mem_data_out
        );

    --------------------------------------------------------------------
    -- Write Back Selection
    --------------------------------------------------------------------
    -- If memory read enabled, write back from memory; else from ALU
    write_back_data <= mem_data_out when mem_rd_en = '1' else alu_result;

end architecture;
```

5. 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.

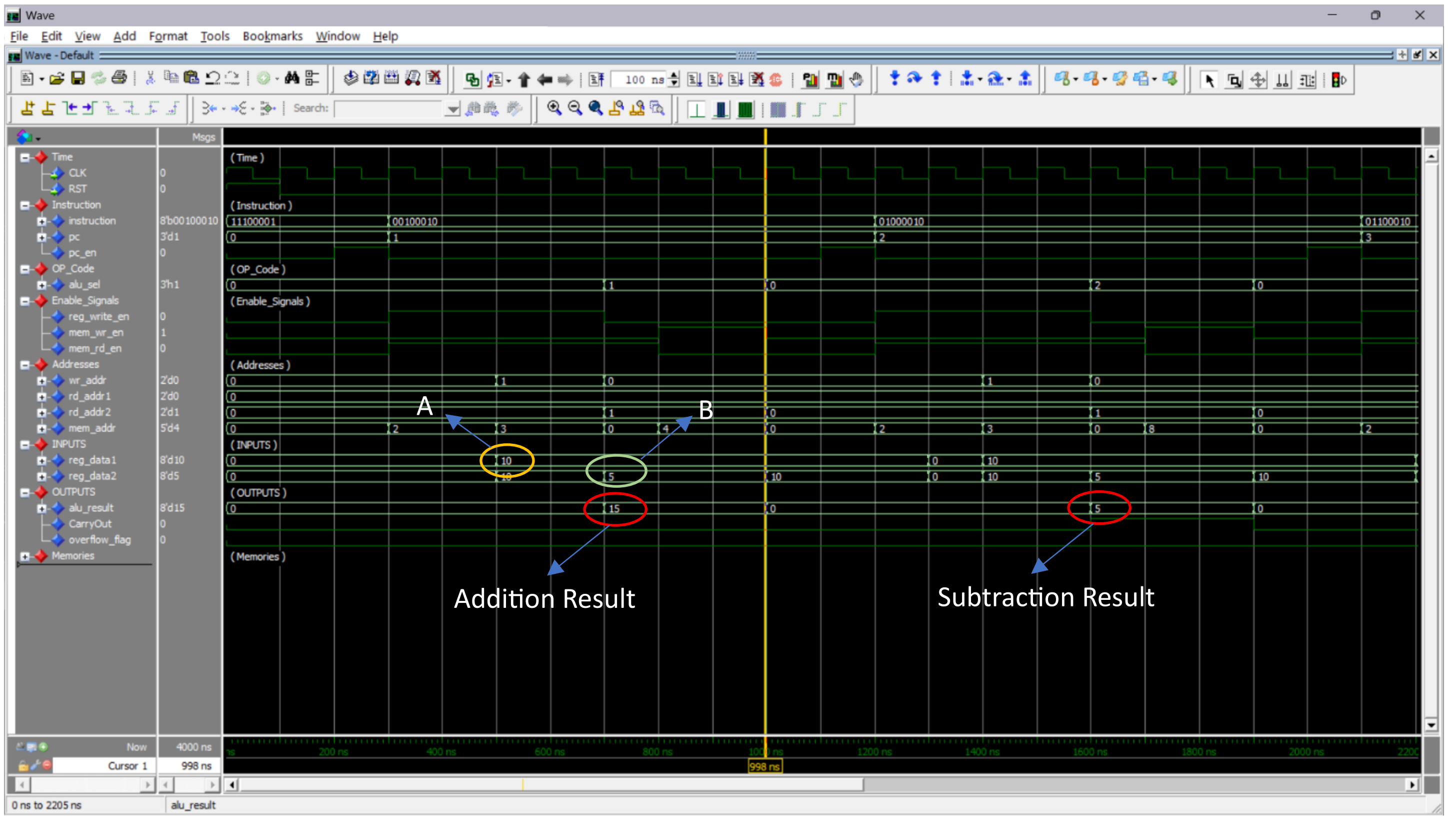
5. **Read Operand (B) from Memory**

- The processor reads the second operand (B) from memory.
- Operand B is also required for the ALU computation.

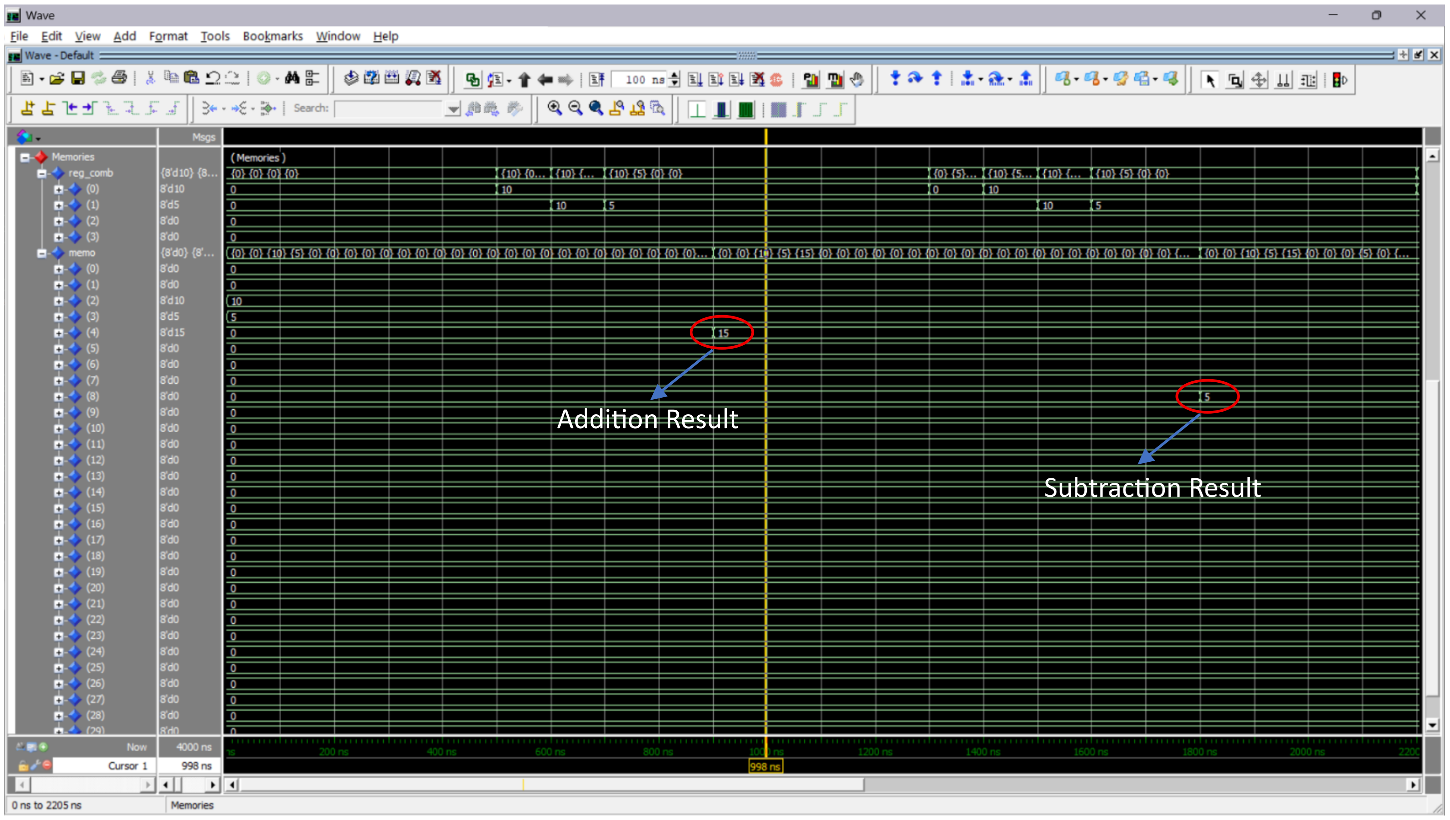
6. Load Operand (B) to Register File
    - Operand B is stored in the register file, ready to be used in the operation.
7. Operation
    - The ALU performs the selected operation (ADD, SUB, MUL, AND, OR, XOR, etc.) on operands A and B.
    - Status flags such as Carry, Overflow, and Parity are updated 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.

6.0 Simulation Results

➢ Addition , Subtraction operation and Storing to the Memory

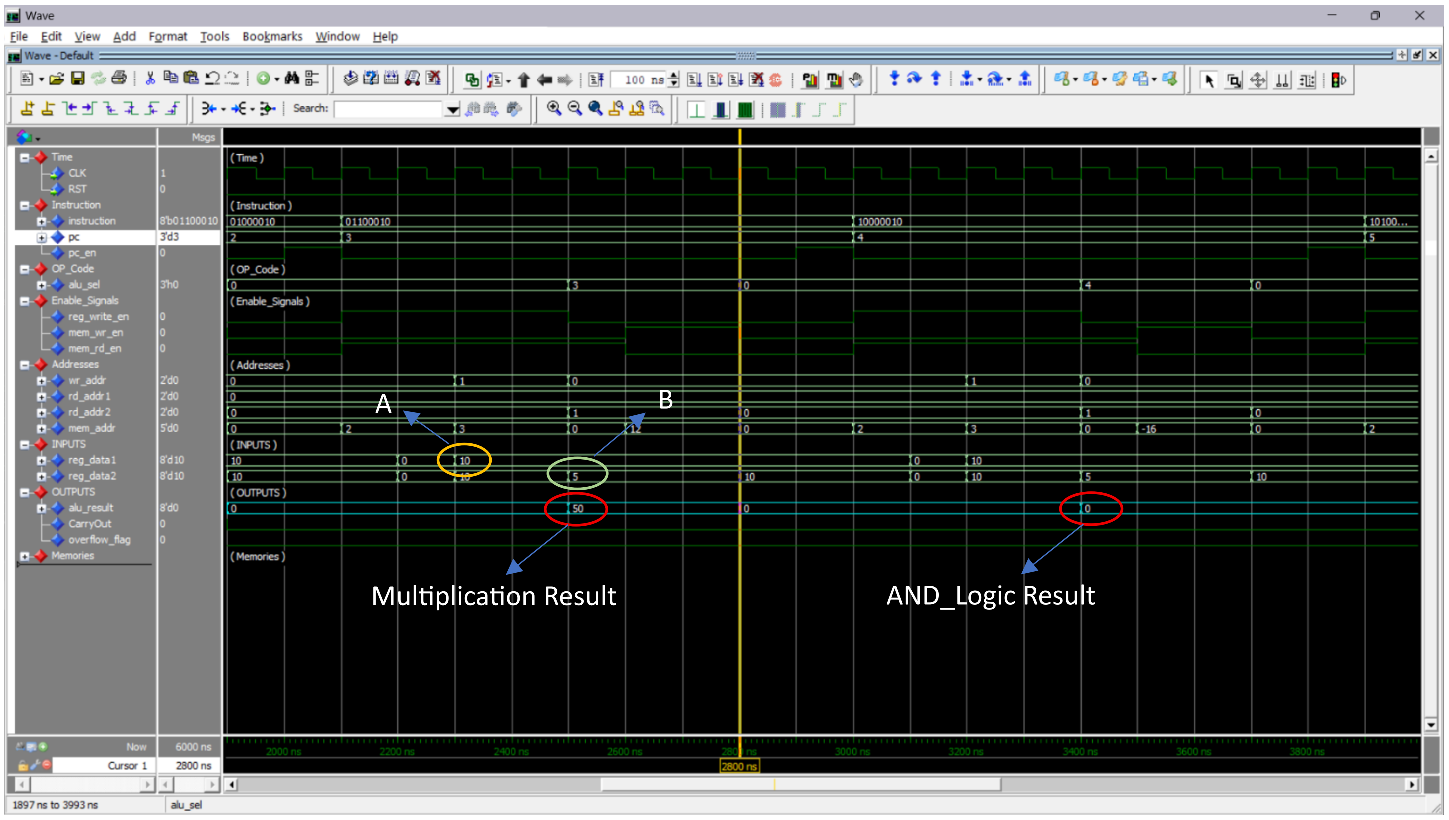


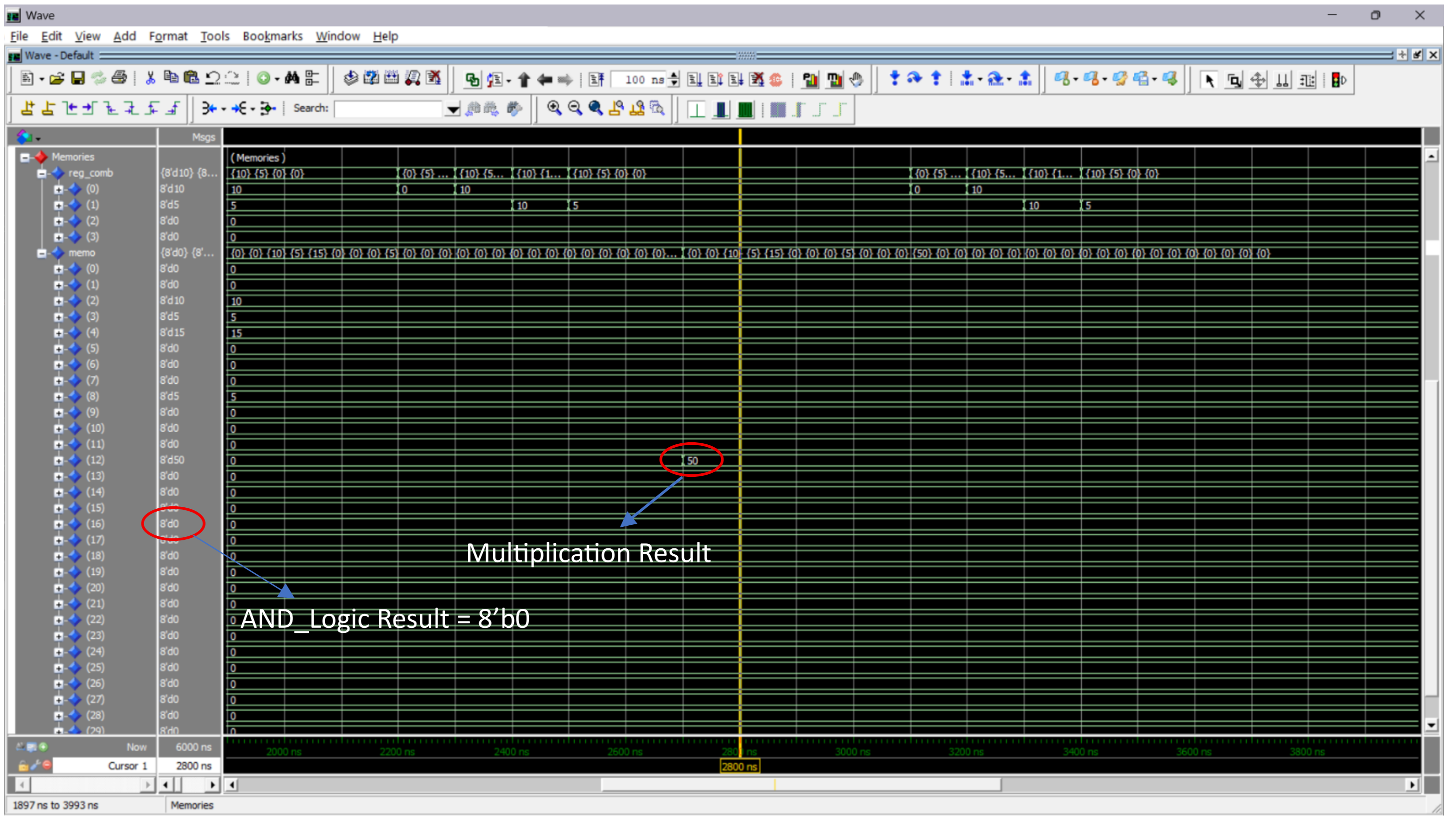
```
Time
  CLK              0
  RST              0
Instruction
  instruction      8'b00100010
  pc               3'd1
  pc_en            0
OP_Code
  alu_sel          3'h1
Enable_Signals
  reg_write_en     0
  mem_wr_en        1
  mem_rd_en        0
Addresses
  wr_addr          2'd0
  rd_addr1         2'd0
  rd_addr2         2'd1
  mem_addr         5'd4
INPUTS
  reg_data1        8'd10
  reg_data2        8'd5
OUTPUTS
  alu_result       8'd15
  CarryOut         0
  overflow_flag    0
Memories

Now      4000 ns
Cursor 1  998 ns
0 ns to 2205 ns    alu_result
```



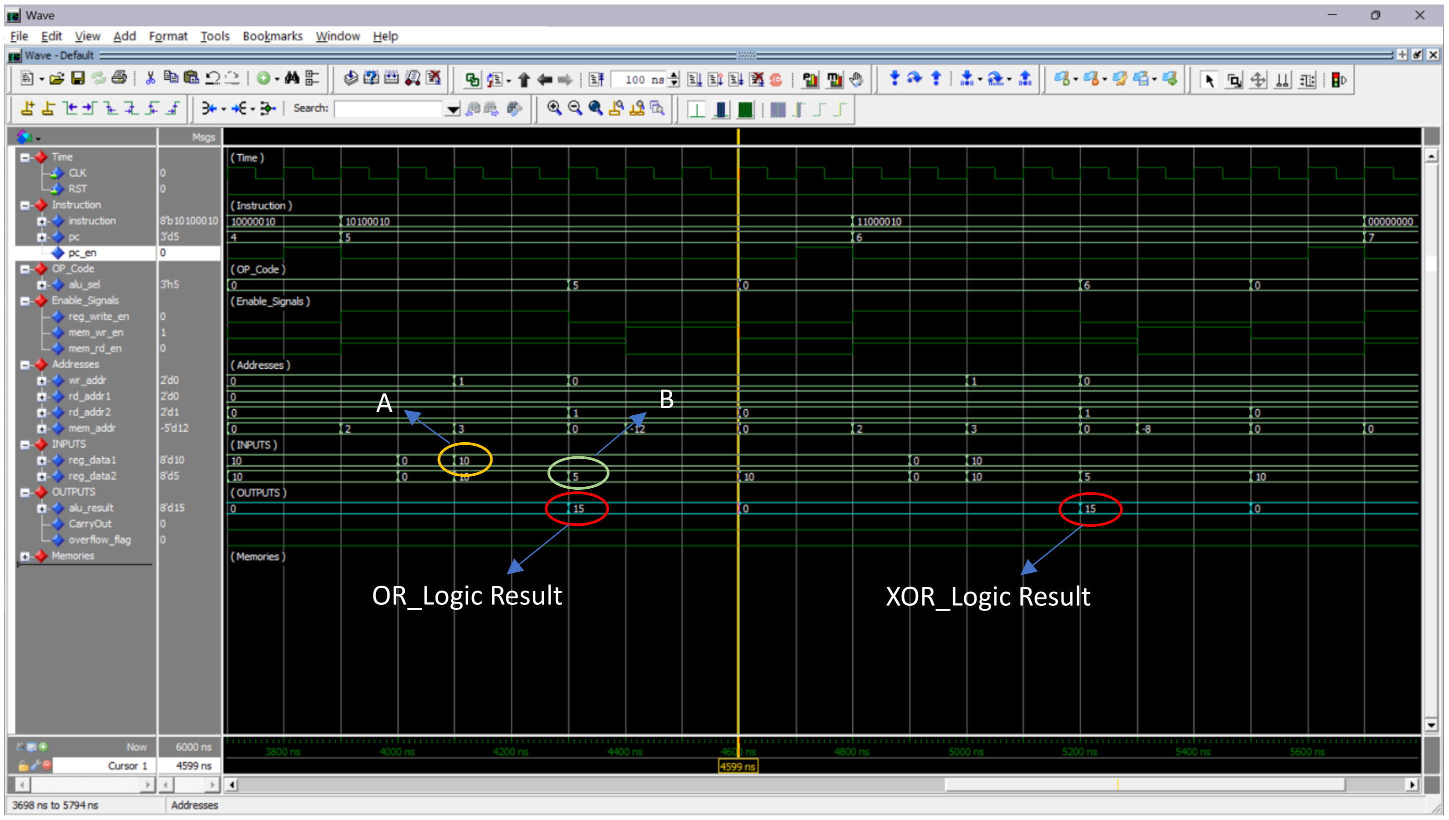
```
Memories
  reg_comb         {8'd10} {8'...
    (0)            8'd10
    (1)            8'd5
    (2)            8'd0
    (3)            8'd0
  memo             {8'd0} {8'...
    (0)            8'd0
    (1)            8'd0
    (2)            8'd10
    (3)            8'd5
    (4)            8'd15
    (5)            8'd0
    (6)            8'd0
    (7)            8'd0
    (8)            8'd0
    (9)            8'd0
    (10)           8'd0
    (11)           8'd0
    (12)           8'd0
    (13)           8'd0
    (14)           8'd0
    (15)           8'd0
    (16)           8'd0
    (17)           8'd0
    (18)           8'd0
    (19)           8'd0
    (20)           8'd0
    (21)           8'd0
    (22)           8'd0
    (23)           8'd0
    (24)           8'd0
    (25)           8'd0
    (26)           8'd0
    (27)           8'd0
    (28)           8'd0

Now      4000 ns
Cursor 1  998 ns
0 ns to 2205 ns    Memories
```

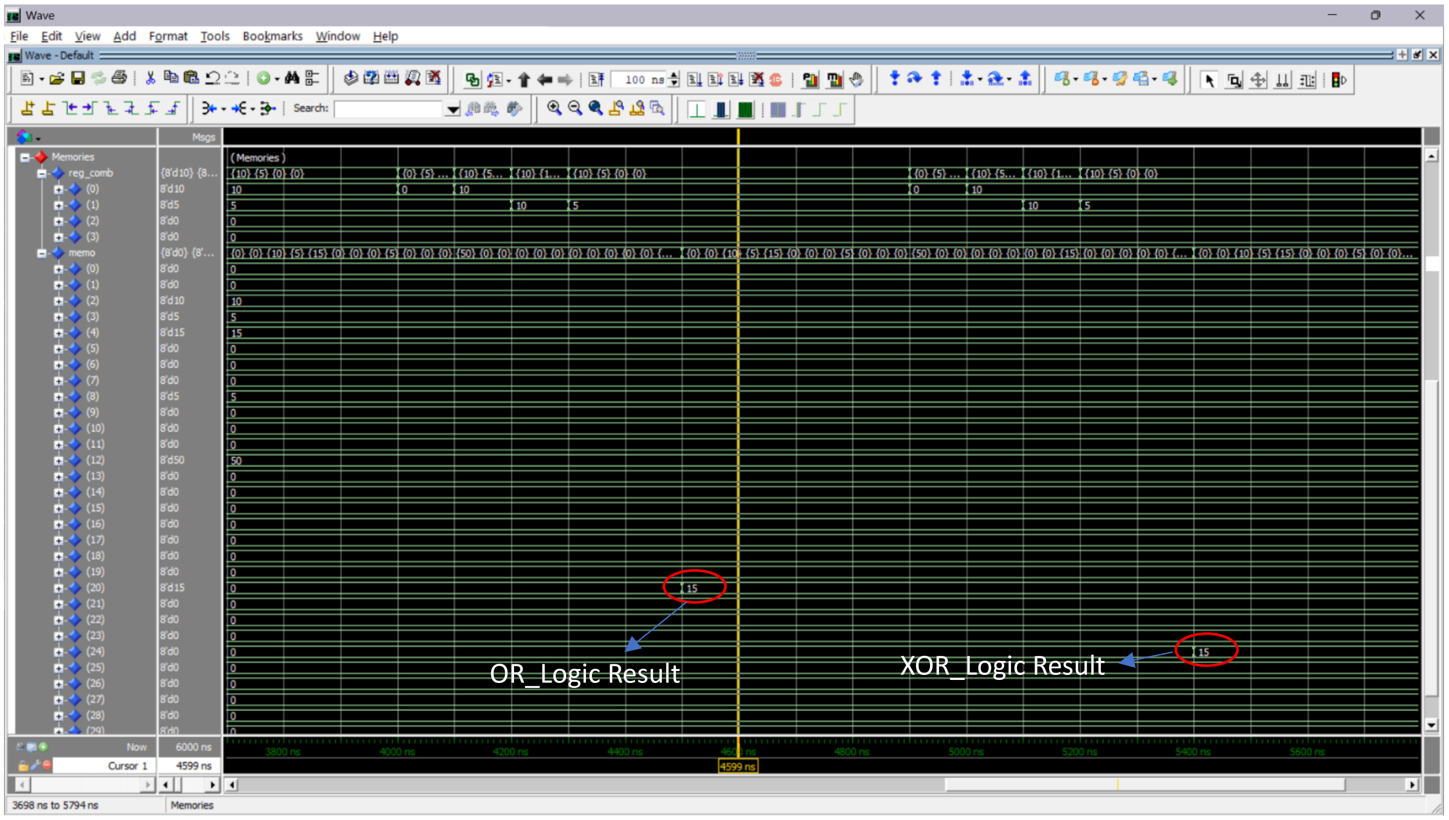
➢ Multiplication, And_Logic operation and Storing to the Memory





➢ OR, XOR Logic operations and storing to the memory





7.0 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.