Alexandria University
Faculty of Engineering
Computer Architecture
Spring 2025

Register File Implementation Report

Waleed Emad Yahyaa 20012197
Mohamed Atef Yousef 20011630

1. Introduction:

In this lab, we designed and implemented a 32-register file using VHDL. The register file allows read and write operations based on a control signal and a clock cycle mechanism. The design ensures that data is written in the first half of the clock cycle and read in the second half.

2. Design Specifications:

2.1. Inputs

- Three Register Numbers: Two for reading and one for writing, each consisting of 5 bits.
- Write Data: A 32-bit input value to be stored in the selected register.
- RegWrite Control Line: Enables the write operation when set to '1'.
- Clock Signal: Controls read and write operations, where writing occurs in the first half cycle and reading in the second half.

2.2 Outputs

- Two Read Data Outputs: Corresponding to the values stored in the registers indexed by the two read addresses.

3. VHDL Implementation:

The register file was implemented using an array of 32 registers, each 32 bits wide. The design adheres to the following logic:

- Writing occurs when RegWrite = '1' and the clock is in its falling edge.
- Reading occurs asynchronously and outputs the stored values based on the given read addresses.

VHDL Code

The complete VHDL implementation is included in the Appendix.

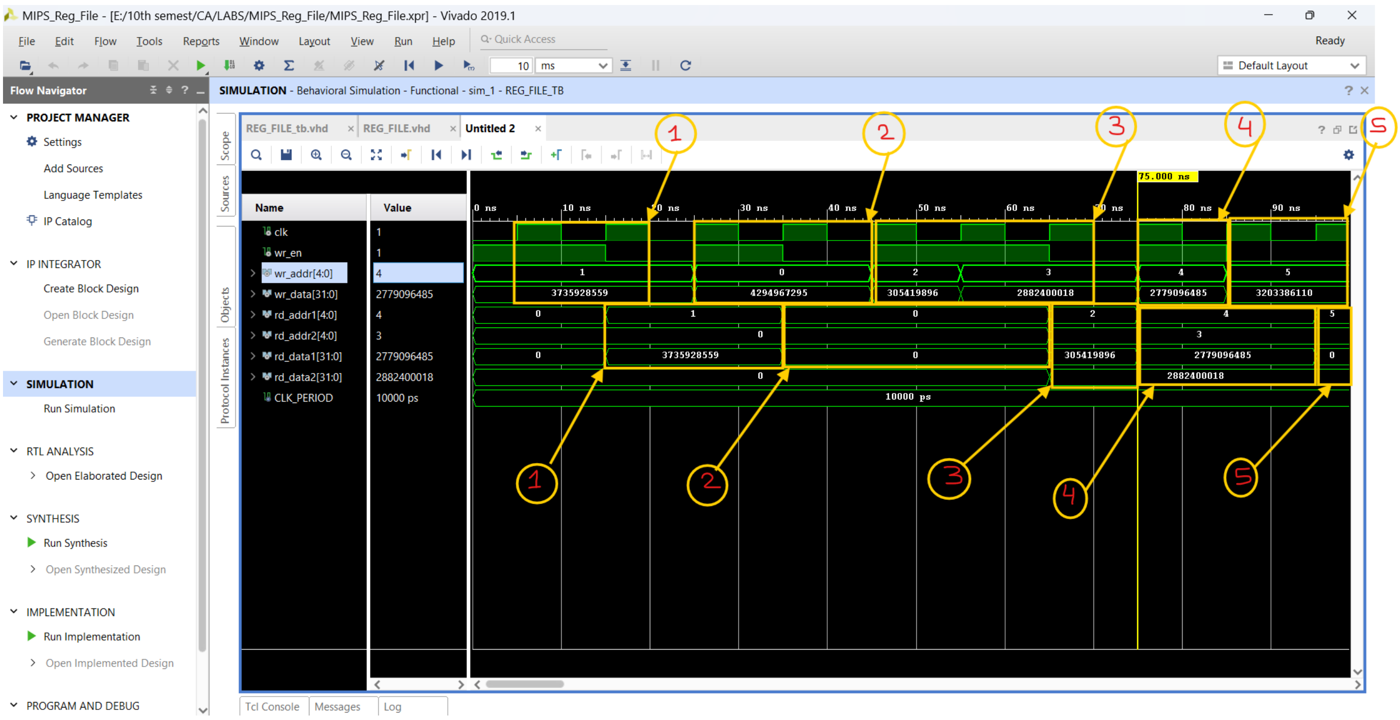
4. Testbench and Simulation:

To verify the correctness of our design, a testbench was developed covering multiple test scenarios, including:

- Writing and reading from registers.
- Writing to register zero to confirm that it remains zero.
- Performing a read and write operation on the same register in the same clock cycle.
- Disabling the write operation and ensuring no unintended changes occur.

Simulation Results

Below are screenshots showing the behavior of the register file during multiple clock cycles. The results confirm that the implementation meets the expected functionality.



5. Tools and Environment:

Vivado was used for compilation and simulation.

6. Submission Details:

The report includes:

- The full VHDL implementation of the register file.
- A detailed testbench covering multiple scenarios.
- Screenshots demonstrating simulation results.
- A link to the complete project repository: https://github.com/moateff/Computer-Architecture-labs

7. Conclusion:

The register file was successfully implemented and tested. The design meets all requirements, ensuring correct read/write operations and handling edge cases effectively.

Appendix: VHDL Code

(Include VHDL code for the register file and testbench here.)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity REG_FILE is
    Port (
        clk       : in  STD_LOGIC;                       -- Clock signal
        wr_en     : in  STD_LOGIC;                       -- Write enable signal
        wr_addr   : in  STD_LOGIC_VECTOR (4 downto 0);  -- Write address (5-bit)
        wr_data   : in  STD_LOGIC_VECTOR (31 downto 0); -- Write data (32-bit)
        rd_addr1  : in  STD_LOGIC_VECTOR (4 downto 0);  -- Read address 1 (5-bit)
        rd_addr2  : in  STD_LOGIC_VECTOR (4 downto 0);  -- Read address 2 (5-bit)
        rd_data1  : out STD_LOGIC_VECTOR (31 downto 0); -- Read data 1 (32-bit)
        rd_data2  : out STD_LOGIC_VECTOR (31 downto 0)  -- Read data 2 (32-bit)
    );
end REG_FILE;

architecture Behavioral of REG_FILE is
    type reg_array is array (0 to 31) of STD_LOGIC_VECTOR(31 downto 0);
    signal reg_memory : reg_array := (others => (others => '0')); -- Initialize to
zeros

begin

    -- Read logic: Outputs data from register memory
    rd_data1 <= reg_memory(to_integer(unsigned(rd_addr1))) when rd_addr1 /= "00000"
else (others => '0');
    rd_data2 <= reg_memory(to_integer(unsigned(rd_addr2))) when rd_addr2 /= "00000"
else (others => '0');

    -- Write logic: Updates register memory on rising edge of clk
    process (clk)
    begin
        if rising_edge(clk) then
            if wr_en = '1' then
                reg_memory(to_integer(unsigned(wr_addr))) <= wr_data;
            end if;
        end if;
    end process;

end Behavioral;
```

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use std.textio.all;


entity REG_FILE_TB is
end REG_FILE_TB;


architecture testbench of REG_FILE_TB is

    -- DUT signals
    signal clk      : std_logic := '0';
    signal wr_en    : std_logic := '0';
    signal wr_addr  : std_logic_vector(4 downto 0) := (others => '0');
    signal wr_data  : std_logic_vector(31 downto 0) := (others => '0');
    signal rd_addr1 : std_logic_vector(4 downto 0) := (others => '0');
    signal rd_addr2 : std_logic_vector(4 downto 0) := (others => '0');
    signal rd_data1 : std_logic_vector(31 downto 0);
    signal rd_data2 : std_logic_vector(31 downto 0);

    constant CLK_PERIOD : time := 10 ns;

begin

    -- Instantiate the Register File DUT
    DUT: entity work.REG_FILE
        port map (
            clk      => clk,
            wr_en    => wr_en,
            wr_addr  => wr_addr,
            wr_data  => wr_data,
            rd_addr1 => rd_addr1,
            rd_addr2 => rd_addr2,
            rd_data1 => rd_data1,
            rd_data2 => rd_data2
        );

    -- Clock process
    clk_process: process
    begin
        while now < 500 ns loop  -- Limit simulation time
            clk <= '0';
            wait for CLK_PERIOD / 2;
            clk <= '1';
            wait for CLK_PERIOD / 2;
        end loop;
        wait;
    end process;
```

```vhdl
-- Stimulus process
stimulus_process: process
begin
    report "Starting REG_FILE Testbench...";

    -- Test Case 1: Write and Read from Register
    wr_en   <= '1';
    wr_addr <= "00001";
    wr_data <= x"DEADBEEF";
    wait for CLK_PERIOD / 2;
    wait for CLK_PERIOD;
    wr_en   <= '0';

    -- Read back the value
    rd_addr1 <= "00001";
    wait for CLK_PERIOD;
    assert rd_data1 = x"DEADBEEF"
        report "Error: Incorrect data read from register!" severity error;

    -- Test Case 2: Writing to Register 0 (Should Remain Zero)
    wr_en   <= '1';
    wr_addr <= "00000";
    wr_data <= x"FFFFFFFF";
    wait for CLK_PERIOD;
    wr_en   <= '0';

    rd_addr1 <= "00000";
    wait for CLK_PERIOD;
    assert rd_data1 = x"00000000"
        report "Error: Register 0 should always be zero!" severity error;

    -- Test Case 3: Multiple Writes and Reads
    wr_en   <= '1';
    wr_addr <= "00010";
    wr_data <= x"12345678";
    wait for CLK_PERIOD;

    wr_addr <= "00011";
    wr_data <= x"ABCDEF12";
    wait for CLK_PERIOD;
    wr_en   <= '0';

    rd_addr1 <= "00010";
    rd_addr2 <= "00011";
    wait for CLK_PERIOD;

    assert rd_data1 = x"12345678"
        report "Error: Incorrect data read from register 2!" severity error;
    assert rd_data2 = x"ABCDEF12"
        report "Error: Incorrect data read from register 3!" severity error;
```

```
        -- Test Case 4: Write and Read in the Same Clock Cycle
        wr_en   <= '1';
        wr_addr <= "00100";
        wr_data <= x"A5A5A5A5";
        rd_addr1 <= "00100";
        wait for CLK_PERIOD;
        wr_en   <= '0';

        assert rd_data1 = x"A5A5A5A5"
            report "Error: Failed to read and write in the same cycle!" severity
error;

        -- Test Case 5: RegWrite Disabled (wr_en = 0)
        wr_en   <= '0';
        wr_addr <= "00101";
        wr_data <= x"BEEFCAFE";
        wait for CLK_PERIOD;

        rd_addr1 <= "00101";
        wait for CLK_PERIOD;
        assert rd_data1 = x"00000000"
            report "Error: Data should not be written when wr_en is '0'!" severity
error;

        report "Testbench Completed Successfully." severity note;
        wait;
    end process;

end testbench;
```