Project Title: 4-Bit ALU

Student Name1: Ahmed Abdelmonem Mohamed

Student Name2: Youssef Alaa Elbarbary

Group: S25-B2-FPGA-G2-E

Abstract

This project presents the design and simulation of a basic 8-bit microprocessor built using digital logic components.

The processor is capable of executing a small set of instructions including load, store, add, and jump.

The architecture includes a Program Counter, Instruction Memory, 4×8-bit Registers, an Arithmetic Logic Unit (ALU), and a Control Unit, all connected through an 8-bit data bus.

The system was implemented and tested using simulation tools to verify the correct execution of instructions and data flow.

This project provided a hands-on understanding of processor architecture, instruction cycles, and the interaction between hardware modules without the use of high-level programming languages.

Introduction

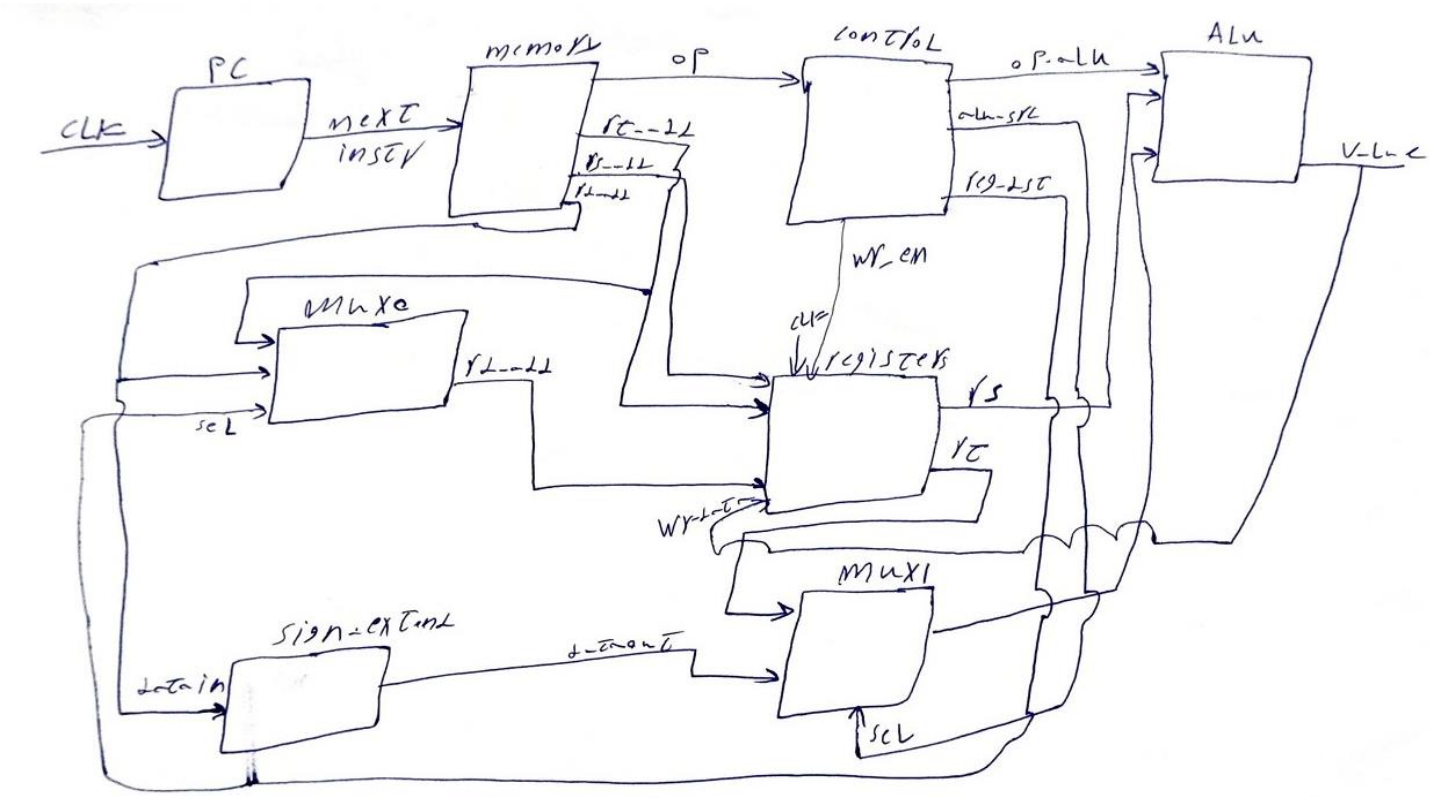
In this project, we designed and simulated a simple 8-bit microprocessor using fundamental digital logic components.

The goal was to build a working processor capable of executing a small set of instructions such as load, store, add, and jump, without relying on any high-level language.

Our microprocessor includes essential components like a Program Counter, Instruction Memory, Registers, ALU, and a Control Unit. The design was implemented and tested through simulation to demonstrate the flow of instruction execution and data processing.

This project helped us understand the core structure of a processor and how control signals manage the interaction between components to perform basic operations.

Design



The microprocessor was designed using a modular approach, where each component was implemented and tested separately before integration. The main components of the design are:

Program Counter (PC): A register that holds the address of the next instruction to be executed. It increments automatically after each instruction or changes based on jump instructions.

Instruction Memory: A ROM that stores the set of instructions. It receives the address from the PC and outputs the corresponding instruction.

Register File: Contains four 8-bit general-purpose registers used for temporary data storage during instruction execution.

ALU (Arithmetic Logic Unit): Performs arithmetic and logic operations (such as addition, subtraction, AND, OR). It receives operands from the registers and outputs the result to a destination register.

Control Unit: Decodes the instruction and generates the necessary control signals to manage data flow between components.

Multiplexers: Used to select between different data sources based on control signals, allowing conditional operations such as branching.

All components are connected via an 8-bit data bus and synchronized using a common clock signal. The control unit coordinates the execution process by activating the appropriate control lines depending on the instruction being executed.

VHDL codes:

PC

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

entity pc_cpu is
  port (
      clk : in std_logic ;
      next_state :out std_logic_vector(2 downto 0)
      );
end entity ;

architecture behavior of pc_cpu is
  signal current_state :std_logic_vector(2 downto 0):="000";
begin
  process ( clk )
  begin
      if falling_edge (clk) then
        current_state <= std_logic_vector(unsigned(current_state)+ 001);
      end if;
  end process;

  next_state <= current_state;

end architecture ;
```

Instraction

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

entity instraction_cpu is
      port(
            instr_adder : in std_logic_vector(2 downto 0);
            op          : out std_logic_vector(1 downto 0);
            rs          : out std_logic_vector(1 downto 0);
            rt          : out std_logic_vector(1 downto 0);
            rd          : out std_logic_vector(1 downto 0)
      );
end entity;

architecture behavior of instraction_cpu is
      type instraction_set is array (0 to 7) of std_logic_vector(7 d
      constant instr : instraction_set := (
            "01000010",  -- Instr 0: op=01(ADD), rs=00, rt=00
            "11010101",  -- Instr 1: op=11(ADDI), rs=01, rt=01
            "11101011",  -- Instr 2: op=11(ADDI), rs=10, rt=10
            "01000111",  -- Instr 3: op=01(ADD), rs=00, rt=00
            "10101100",  -- Instr 4: op=10(SUB), rs=10, rt=11
            "00000110",  -- Instr 5: op=00(AND), rs=01, rt=10
            "00100000",  -- Instr 6: op=00(AND), rs=10, rt=00
            "00000000"   -- Instr 7: op=00(AND), rs=00, rt=00
      );
begin
      op <= instr(to_integer(unsigned(instr_adder)))(7 downto 6);
      rs <= instr(to_integer(unsigned(instr_adder)))(5 downto 4);
      rt <= instr(to_integer(unsigned(instr_adder)))(3 downto 2);
      rd <= instr(to_integer(unsigned(instr_adder)))(1 downto 0);
end architecture;
```

Reguister

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

entity registers_cpu is
    port (
        clk      : in std_logic;
        wr_en    : in std_logic;
        rs_addr  : in std_logic_vector(1 downto 0);
        rd_addr  : in std_logic_vector(1 downto 0);
        rt_addr  : in std_logic_vector(1 downto 0);
        wr_data  : in std_logic_vector(7 downto 0);
        rs       : out std_logic_vector(7 downto 0);
        rt       : out std_logic_vector(7 downto 0)
    );
end entity;

architecture behavior of registers_cpu is
    type registers_set is array (0 to 3) of std_logic_vector(7 downto 0);
    signal regis : registers_set := (
        "01000010",  -- 66
        "11010101",  -- 213
        "11101011",  -- 235
        "01000111"   -- 71
    );
begin
    process(clk)
    begin
        if falling_edge(clk) then
            if wr_en = '1' then
                regis(to_integer(unsigned(rd_addr))) <= wr_data;
            end if;
        end if;
    end process;

    rs <= regis(to_integer(unsigned(rs_addr)));
    rt <= regis(to_integer(unsigned(rt_addr)));
end architecture;
```

ALU

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

entity ALU_cpu is
    port(
        op        : in std_logic_vector(1 downto 0);
        rs        : in std_logic_vector(7 downto 0);
        rt        : in std_logic_vector(7 downto 0);
        rd        : out std_logic_vector(7 downto 0)
    );
end entity;

architecture behavior of ALU_cpu is
    signal result : std_logic_vector(7 downto 0);
begin
    process(op, rs, rt)
    begin
        case op is
            when "00" =>
                result <= rs and rt;
            when "01" =>
                result <= std_logic_vector(unsigned(rs) + unsigned(rt));
            when "10" =>
                result <= std_logic_vector(unsigned(rs) - unsigned(rt));
            when "11" =>
                result <= std_logic_vector(unsigned(rs) + unsigned(rt));
            when others =>
                result <= (others => '0');
        end case;
    end process;

    rd <= result;
end architecture;
```

Multiplexers

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

entity MUX0_cpu is
    port (
        a   : in std_logic_vector(1 downto 0);
        b   : in std_logic_vector(1 downto 0);
        sel : in std_logic;
        y   : out std_logic_vector(1 downto 0)
    );
end entity;

architecture behavioral of MUX0_cpu is
begin
    process (a, b, sel)
    begin
        if sel = '1' then
            y <= a;
        else
            y <= b;
        end if;
    end process;
end architecture;
```

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

entity MUX1_cpu is
    port (
        a   : in std_logic_vector(7 downto 0);
        b   : in std_logic_vector(7 downto 0);
        sel : in std_logic;
        y   : out std_logic_vector(7 downto 0)
    );
end entity;

architecture behavioral of MUX1_cpu is
begin
    process (a, b, sel)
    begin
        if sel = '1' then
            y <= b;
        else
            y <= a;
        end if;
    end process;
end architecture;
```

Sign extend

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

entity sign_extend is
    port(
        input_2bit  : in  std_logic_vector(1 downto 0);
        output_8bit : out std_logic_vector(7 downto 0)
    );
end entity;

architecture behavior of sign_extend is
begin
    output_8bit <= "000000" & input_2bit;
end architecture;

```

Control Unit

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

entity control_cpu is
    port (
        instr   : in std_logic_vector(1 downto 0);
        alu_op  : out std_logic_vector(1 downto 0);
        alu_src : out std_logic;
        reg_dst : out std_logic;
        wr_en   : out std_logic
    );
end entity;

architecture behavior of control_cpu is
begin
    process(instr)
    begin
        if instr = "00" then
            alu_op <= "00";  -- AND
            alu_src <= '0';
            reg_dst <= '0';
        elsif instr = "01" then
            alu_op <= "01";  -- ADD
            alu_src <= '0';
            reg_dst <= '0';
        elsif instr = "10" then
            alu_op <= "10";  -- SUB
            alu_src <= '0';
            reg_dst <= '0';
        else
            alu_op <= "11";  -- ADDI
            alu_src <= '1';
            reg_dst <= '1';
        end if;
    end process;

    wr_en <= '1';

end architecture;
```

UProcessor

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

entity processor_cpu is
    port (
        clk   : in std_logic;
        value : out std_logic_vector(7 downto 0)
    );
end entity;

architecture Behavioral of processor_cpu is

```

```vhdl
architecture Behavioral of processor_cpu is

    -- === Components ===
    component ALU_cpu is
        port(
            op : in std_logic_vector(1 downto 0);
            rs : in std_logic_vector(7 downto 0);
            rt : in std_logic_vector(7 downto 0);
            rd : out std_logic_vector(7 downto 0)
        );
    end component;

    component control_cpu is
        port(
            instr   : in std_logic_vector(1 downto 0);
            alu_op  : out std_logic_vector(1 downto 0);
            alu_src : out std_logic;
            reg_dst : out std_logic;
            wr_en   : out std_logic
        );
    end component;

    component instraction_cpu is
        port(
            instr_adder : in std_logic_vector(2 downto 0);
            op          : out std_logic_vector(1 downto 0);
            rs          : out std_logic_vector(1 downto 0);
            rt          : out std_logic_vector(1 downto 0);
            rd          : out std_logic_vector(1 downto 0)
        );
    end component;

    component MUX0_cpu is
        port (
            a   : in std_logic_vector(1 downto 0);
            b   : in std_logic_vector(1 downto 0);
            sel : in std_logic;
            y   : out std_logic_vector(1 downto 0)
        );
    end component;

    component MUX1_cpu is
        port (
            a   : in std_logic_vector(7 downto 0);
            b   : in std_logic_vector(7 downto 0);
            sel : in std_logic;
            y   : out std_logic_vector(7 downto 0)
        );
    end component;

    component pc_cpu is
        port (
            clk        : in std_logic;
            next_state : out std_logic_vector(2 downto 0)
        );
    end component;

    component registers_cpu is
        port (
            clk      : in std_logic;
            wr_en    : in std_logic;
            rs_addr  : in std_logic_vector(1 downto 0);
            rd_addr  : in std_logic_vector(1 downto 0);
            rt_addr  : in std_logic_vector(1 downto 0);
            wr_data  : in std_logic_vector(7 downto 0);
            rs       : out std_logic_vector(7 downto 0);
            rt       : out std_logic_vector(7 downto 0)
        );
    end component;

    component sign_extend is
        port(
            input_2bit  : in  std_logic_vector(1 downto 0);
            output_8bit : out std_logic_vector(7 downto 0)
        );
    end component;
```

```
    -- === Signals ===
    signal op           : std_logic_vector(1 downto 0);
    signal ctrl_alu_op  : std_logic_vector(1 downto 0);
    signal ctrl_alu_src : std_logic;
    signal ctrl_reg_dst : std_logic;
    signal ctrl_wr_en   : std_logic;

    signal instr_rs_addr : std_logic_vector(1 downto 0);
    signal instr_rt_addr : std_logic_vector(1 downto 0);
    signal instr_rd_addr : std_logic_vector(1 downto 0);

    signal reg_rs      : std_logic_vector(7 downto 0);
    signal reg_rt      : std_logic_vector(7 downto 0);
    signal reg_rt_mux  : std_logic_vector(7 downto 0);
    signal mux0_rd     : std_logic_vector(1 downto 0);
    signal alu_result  : std_logic_vector(7 downto 0);
    signal sign_ext_out: std_logic_vector(7 downto 0);
    signal next_instr  : std_logic_vector(2 downto 0);
    signal final_result: std_logic_vector(7 downto 0);

```

```
begin

    value <= final_result;

    -- === Modules Instantiation ===
    arithmetic_logic_unit: ALU_cpu
        port map (
            op => ctrl_alu_op,
            rs => reg_rs,
            rt => reg_rt_mux,
            rd => alu_result
        );

    control_unit: control_cpu
        port map (
            instr   => op,
            alu_op  => ctrl_alu_op,
            alu_src => ctrl_alu_src,
            reg_dst => ctrl_reg_dst,
            wr_en   => ctrl_wr_en
        );

    instruction_memory: instraction_cpu
        port map (
            instr_adder => next_instr,
            op          => op,
            rs          => instr_rs_addr,
            rt          => instr_rt_addr,
            rd          => instr_rd_addr
        );

    mux_0: MUX0_cpu
        port map (
            a   => instr_rd_addr,
            b   => instr_rt_addr,
            sel => ctrl_reg_dst,
            y   => mux0_rd
        );

    mux_1: MUX1_cpu
        port map (
            a   => reg_rt,
            b   => sign_ext_out,
            sel => ctrl_alu_src,
            y   => reg_rt_mux
        );

    program_counter: pc_cpu
        port map (
            clk        => clk,
            next_state => next_instr
        );

    registers: registers_cpu
        port map (
            clk      => clk,
            wr_en    => ctrl_wr_en,
            rs_addr  => instr_rs_addr,
            rd_addr  => mux0_rd,
            rt_addr  => instr_rt_addr,
            wr_data  => alu_result,
            rs       => reg_rs,
            rt       => reg_rt
        );

    sign_ext_unit: sign_extend
        port map (
            input_2bit  => instr_rd_addr,
            output_8bit => sign_ext_out
        );

    -- === Result Registering ===
    process(clk)
    begin
        if rising_edge(clk) then
            final_result <= alu_result;
        end if;
    end process;

```

Conclusion

In this project, we successfully designed and simulated an 8-bit microprocessor using basic digital logic components.
The processor was able to execute a predefined set of instructions such as load, store, add, and jump.

Through the design and testing process, we gained a deeper understanding of how microprocessors work internally, including instruction flow, control signals, and data path design.

This project strengthened our practical skills in digital systems and gave us a clearer view of how software instructions are translated into hardware operations.

Future improvements may include adding more instructions, implementing pipelining, or expanding memory and register capabilities