

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

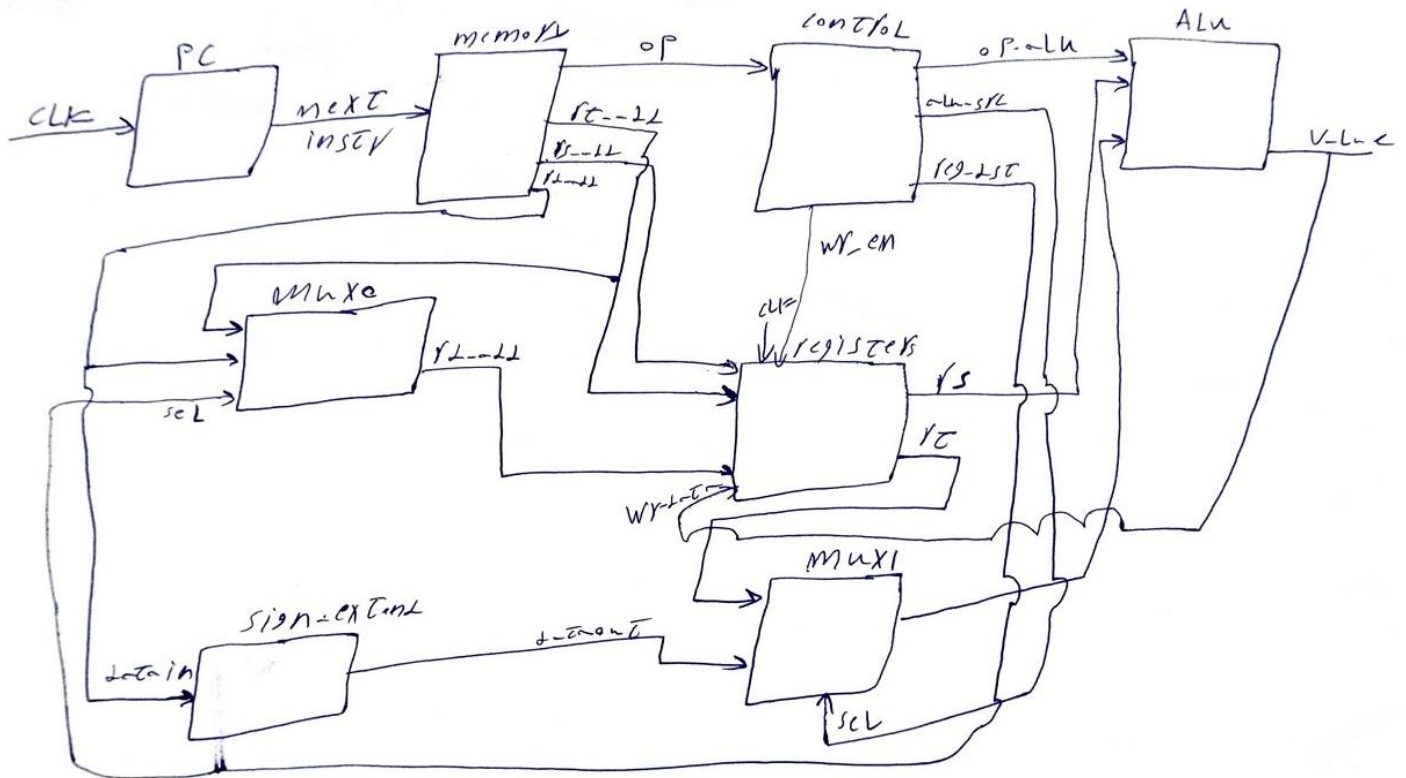
In this project, we designed and simulated a simple 8-bit microprocessor using fundamental digital logic components.

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



```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity pc_cpu is
6      port (
7          clk : in std_logic ;
8          next_state :out std_logic_vector(2 downto 0)
9      );
10 end entity ;
11
12 architecture behavior of pc_cpu is
13     signal current_state :std_logic_vector(2 downto 0):="000";
14 begin
15     process ( clk )
16     begin
17         if falling_edge (clk) then
18             current_state <= std_logic_vector(unsigned(current_state)+ 001);
19         end if;
20     end process;
21
22     next_state <= current_state;
23
24 end architecture ;
```

Instruction

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity instruction_cpu is
6      port(
7          instr_adder : in std_logic_vector(2 downto 0);
8          op          : out std_logic_vector(1 downto 0);
9          rs          : out std_logic_vector(1 downto 0);
10         rt          : out std_logic_vector(1 downto 0);
11         rd          : out std_logic_vector(1 downto 0);
12     );
13 end entity;
14
15 architecture behavior of instruction_cpu is
16     type instruction_set is array (0 to 7) of std_logic_vector(7 downto 0);
17     constant instr : instruction_set := (
18         "01000010", -- Instr 0: op=01(ADD), rs=00, rt=00
19         "11010101", -- Instr 1: op=11(ADDI), rs=01, rt=01
20         "11101011", -- Instr 2: op=11(ADDI), rs=10, rt=10
21         "01000111", -- Instr 3: op=01(ADD), rs=00, rt=00
22         "10101100", -- Instr 4: op=10(SUB), rs=10, rt=11
23         "00000110", -- Instr 5: op=00(AND), rs=01, rt=10
24         "00100000", -- Instr 6: op=00(AND), rs=10, rt=00
25         "00000000"  -- Instr 7: op=00(AND), rs=00, rt=00
26     );
27 begin
28     op <= instr(to_integer(unsigned(instr_adder)))(7 downto 6);
29     rs <= instr(to_integer(unsigned(instr_adder)))(5 downto 4);
30     rt <= instr(to_integer(unsigned(instr_adder)))(3 downto 2);
31     rd <= instr(to_integer(unsigned(instr_adder)))(1 downto 0);
32 end architecture;
```


Reguister

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity registers_cpu is
6      port (
7          clk      : in std_logic;
8          wr_en    : in std_logic;
9          rs_addr   : in std_logic_vector(1 downto 0);
10         rd_addr   : in std_logic_vector(1 downto 0);
11         rt_addr   : in std_logic_vector(1 downto 0);
12         wr_data    : in std_logic_vector(7 downto 0);
13         rs        : out std_logic_vector(7 downto 0);
14         rt        : out std_logic_vector(7 downto 0)
15     );
16 end entity;
17
18 architecture behavior of registers_cpu is
19     type registers_set is array (0 to 3) of std_logic_vector(7 downto 0);
20     signal regis : registers_set := (
21         "01000010", -- 66
22         "11010101", -- 213
23         "11101011", -- 235
24         "01000111"  -- 71
25     );
26 begin
27     process(clk)
28     begin
29         if falling_edge(clk) then
30             if wr_en = '1' then
31                 regis(to_integer(unsigned(rd_addr))) <= wr_data;
32             end if;
33         end if;
34     end process;
35
36     rs <= regis(to_integer(unsigned(rs_addr)));
37     rt <= regis(to_integer(unsigned(rt_addr)));
38 end architecture;
```

ALU

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity ALU_cpu is
6      port(
7          op      : in std_logic_vector(1 downto 0);
8          rs      : in std_logic_vector(7 downto 0);
9          rt      : in std_logic_vector(7 downto 0);
10         rd      : out std_logic_vector(7 downto 0)
11     );
12 end entity;
13
14 architecture behavior of ALU_cpu is
15     signal result : std_logic_vector(7 downto 0);
16 begin
17     process(op, rs, rt)
18     begin
19         case op is
20             when "00" =>
21                 result <= rs and rt;
22             when "01" =>
23                 result <= std_logic_vector(unsigned(rs) + unsigned(rt));
24             when "10" =>
25                 result <= std_logic_vector(unsigned(rs) - unsigned(rt));
26             when "11" =>
27                 result <= std_logic_vector(unsigned(rs) + unsigned(rt));
28             when others =>
29                 result <= (others => '0');
30         end case;
31     end process;
32
33     rd <= result;
34 end architecture;
```

Multiplexers




```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity MUX0_cpu is
6      port (
7          a    : in std_logic_vector(1 downto 0);
8          b    : in std_logic_vector(1 downto 0);
9          sel  : in std_logic;
10         y    : out std_logic_vector(1 downto 0)
11     );
12 end entity;
13
14 architecture behavioral of MUX0_cpu is
15 begin
16     process (a, b, sel)
17     begin
18         if sel = '1' then
19             y <= a;
20         else
21             y <= b;
22         end if;
23     end process;
24 end architecture;
```




```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity MUX1_cpu is
5      port (
6          a    : in std_logic_vector(7 downto 0);
7          b    : in std_logic_vector(7 downto 0);
8          sel  : in std_logic;
9          y    : out std_logic_vector(7 downto 0)
10     );
11 end entity;
12
13 architecture behavioral of MUX1_cpu is
14 begin
15     process (a, b, sel)
16     begin
17         if sel = '1' then
18             y <= b;
19         else
20             y <= a;
21         end if;
22     end process;
23 end architecture;
```

Sign extend



```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity sign_extend is
6      port(
7          input_2bit  : in  std_logic_vector(1 downto 0);
8          output_8bit : out std_logic_vector(7 downto 0)
9      );
10 end entity;
11
12 architecture behavior of sign_extend is
13 begin
14     output_8bit <= "000000" & input_2bit;
15 end architecture;
16
```

Control Unit



```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity control_cpu is
5      port (
6          instr    : in std_logic_vector(1 downto 0);
7          alu_op    : out std_logic_vector(1 downto 0);
8          alu_src   : out std_logic;
9          reg_dst   : out std_logic;
10         wr_en     : out std_logic
11     );
12 end entity;
13
14 architecture behavior of control_cpu is
15 begin
16     process(instr)
17     begin
18         if instr = "00" then
19             alu_op <= "00";  -- AND
20             alu_src <= '0';
21             reg_dst <= '0';
22         elsif instr = "01" then
23             alu_op <= "01";  -- ADD
24             alu_src <= '0';
25             reg_dst <= '0';
26         elsif instr = "10" then
27             alu_op <= "10";  -- SUB
28             alu_src <= '0';
29             reg_dst <= '0';
30         else
31             alu_op <= "11";  -- ADDI
32             alu_src <= '1';
33             reg_dst <= '1';
34         end if;
35     end process;
36
37     wr_en <= '1';
38
39 end architecture;
```

UProcessor



```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity processor_cpu is
6      port (
7          clk      : in std_logic;
8          value : out std_logic_vector(7 downto 0)
9      );
10 end entity;
11
12 architecture Behavioral of processor_cpu is
13
```

```
12 architecture Behavioral of processor_cpu is
13
14     -- === Components ===
15     component ALU_cpu is
16     port(
17         op : in std_logic_vector(1 downto 0);
18         rs : in std_logic_vector(7 downto 0);
19         rt : in std_logic_vector(7 downto 0);
20         rd : out std_logic_vector(7 downto 0)
21     );
22 end component;
23
24     component control_cpu is
25     port(
26         instr      : in std_logic_vector(1 downto 0);
27         alu_op      : out std_logic_vector(1 downto 0);
28         alu_src      : out std_logic;
29         reg_dst      : out std_logic;
30         wr_en       : out std_logic
31     );
32 end component;
33
34     component instruction_cpu is
35     port(
36         instr_adder : in std_logic_vector(2 downto 0);
37         op           : out std_logic_vector(1 downto 0);
38         rs           : out std_logic_vector(1 downto 0);
39         rt           : out std_logic_vector(1 downto 0);
40         rd           : out std_logic_vector(1 downto 0)
41     );
42 end component;
43
44     component MUX0_cpu is
45     port (
46         a : in std_logic_vector(1 downto 0);
47         b : in std_logic_vector(1 downto 0);
48         sel : in std_logic;
49         y : out std_logic_vector(1 downto 0)
50     );
51 end component;
52
53     component MUX1_cpu is
54     port (
55         a : in std_logic_vector(7 downto 0);
56         b : in std_logic_vector(7 downto 0);
57         sel : in std_logic;
58         y : out std_logic_vector(7 downto 0)
59     );
60 end component;
61
62     component pc_cpu is
63     port (
64         clk      : in std_logic;
65         next_state : out std_logic_vector(2 downto 0)
66     );
67 end component;
68
69     component registers_cpu is
70     port (
71         clk      : in std_logic;
72         wr_en     : in std_logic;
73         rs_addr   : in std_logic_vector(1 downto 0);
74         rd_addr   : in std_logic_vector(1 downto 0);
75         rt_addr   : in std_logic_vector(1 downto 0);
76         wr_data   : in std_logic_vector(7 downto 0);
77         rs        : out std_logic_vector(7 downto 0);
78         rt        : out std_logic_vector(7 downto 0)
79     );
80 end component;
81
82     component sign_extend is
83     port(
84         input_2bit : in std_logic_vector(1 downto 0);
85         output_8bit : out std_logic_vector(7 downto 0)
86     );
87 end component;
88
```

```

89  -- === Signals ===
90  signal op      : std_logic_vector(1 downto 0);
91  signal ctrl_alu_op : std_logic_vector(1 downto 0);
92  signal ctrl_alu_src : std_logic;
93  signal ctrl_reg_dst : std_logic;
94  signal ctrl_wr_en  : std_logic;
95
96  signal instr_rs_addr : std_logic_vector(1 downto 0);
97  signal instr_rt_addr : std_logic_vector(1 downto 0);
98  signal instr_rd_addr : std_logic_vector(1 downto 0);
99
100 signal reg_rs      : std_logic_vector(7 downto 0);
101 signal reg_rt      : std_logic_vector(7 downto 0);
102 signal reg_rt_mux  : std_logic_vector(7 downto 0);
103 signal mux0_rd     : std_logic_vector(1 downto 0);
104 signal alu_result  : std_logic_vector(7 downto 0);
105 signal sign_ext_out: std_logic_vector(7 downto 0);
106 signal next_instr  : std_logic_vector(2 downto 0);
107 signal final_result: std_logic_vector(7 downto 0);
108

```

```

109 begin
110
111     value <= final_result;
112
113     -- === Modules Instantiation ===
114     arithmetic_logic_unit: ALU_cpu
115     port map (
116         op => ctrl_alu_op,
117         rs => reg_rs,
118         rt => reg_rt_mux,
119         rd => alu_result
120     );
121
122     control_unit: control_cpu
123     port map (
124         instr  => op,
125         alu_op  => ctrl_alu_op,
126         alu_src => ctrl_alu_src,
127         reg_dst => ctrl_reg_dst,
128         wr_en  => ctrl_wr_en
129     );
130
131     instruction_memory: instruction_cpu
132     port map (
133         instr_adder => next_instr,
134         op          => op,
135         rs          => instr_rs_addr,
136         rt          => instr_rt_addr,
137         rd          => instr_rd_addr
138     );
139
140     mux_0: MUX0_cpu
141     port map (
142         a  => instr_rd_addr,
143         b  => instr_rt_addr,
144         sel => ctrl_reg_dst,
145         y  => mux0_rd
146     );
147
148     mux_1: MUX1_cpu
149     port map (
150         a  => reg_rt,
151         b  => sign_ext_out,
152         sel => ctrl_alu_src,
153         y  => reg_rt_mux
154     );
155
156     program_counter: pc_cpu
157     port map (
158         clk      => clk,
159         next_state => next_instr
160     );
161
162     registers: registers_cpu
163     port map (
164         clk      => clk,
165         wr_en    => ctrl_wr_en,
166         rs_addr  => instr_rs_addr,
167         rd_addr  => mux0_rd,
168         rt_addr  => instr_rt_addr,
169         wr_data  => alu_result,
170         rs       => reg_rs,
171         rt       => reg_rt
172     );
173
174     sign_ext_unit: sign_extend
175     port map (
176         input_2bit  => instr_rd_addr,
177         output_8bit => sign_ext_out
178     );
179
180     -- === Result Registering ===
181     process(clk)
182     begin
183         if rising_edge(clk) then
184             final_result <= alu_result;
185         end if;
186     end process;
187

```

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