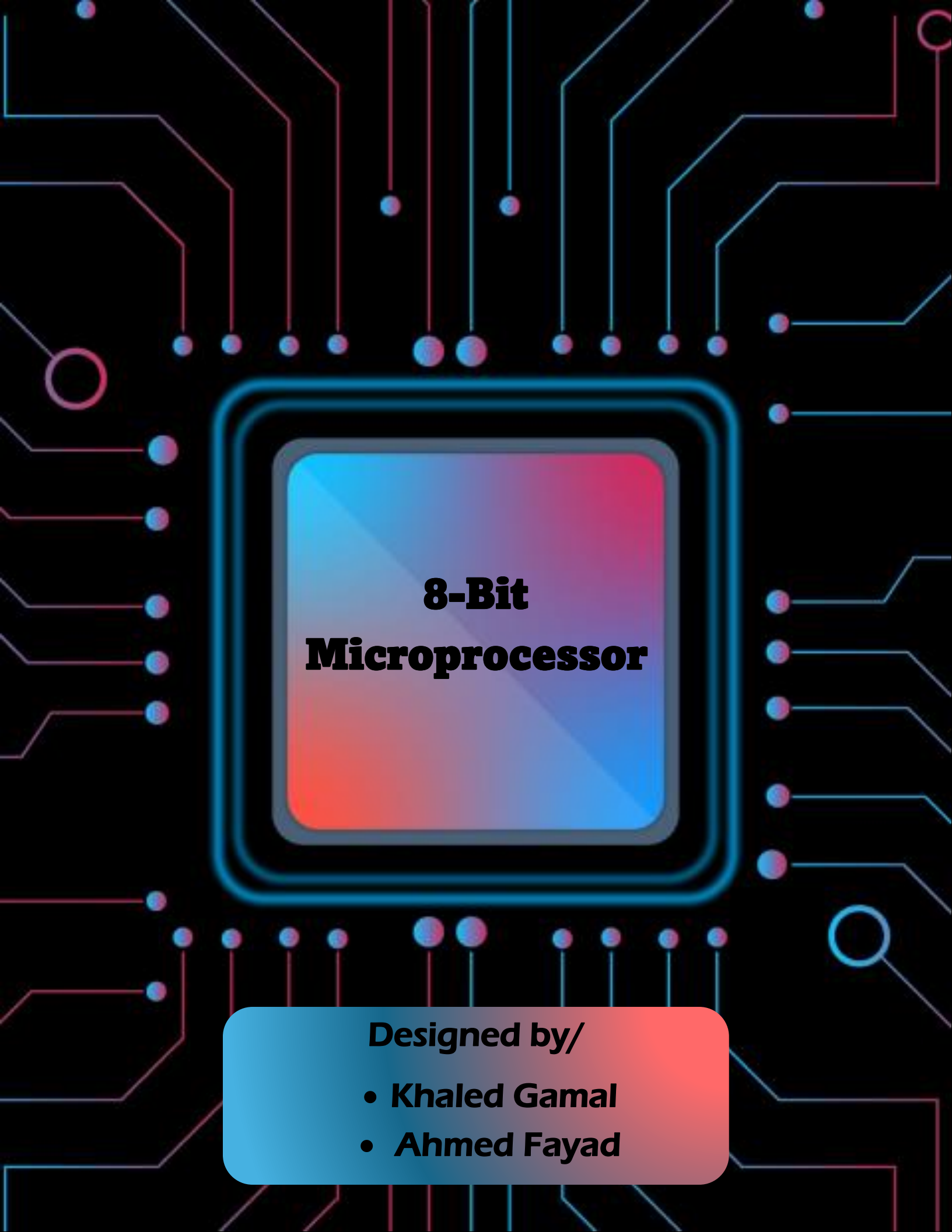


Designed by/

- Khaled Gamal
- Ahmed Fayad

Table of Contents

Overview

This project implements a simplified 8-bit microprocessor using VHDL, designed with a modular architecture suitable for simulation-based verification. The microprocessor supports a basic instruction set architecture (ISA) with operations such as ADD, SUB, AND, and ADDI, and is structured around a classic single-cycle Datapath.

Architecture components

The processor is composed of the following key modules:

1- Program Counter:

The Program Counter is a critical component in the microprocessor design that determines the sequence of instruction execution. It provides the address of the next instruction to be fetched from the instruction memory.

➢ Functionality:

- The PC is implemented as a 3-bit counter (allowing a total of 8 instruction addresses: 0 to 7).
- It increments its value by 1 on every **falling edge** of the clock.
- The current value of the counter is output as next instruction, which is used as the address input to the instruction memory.

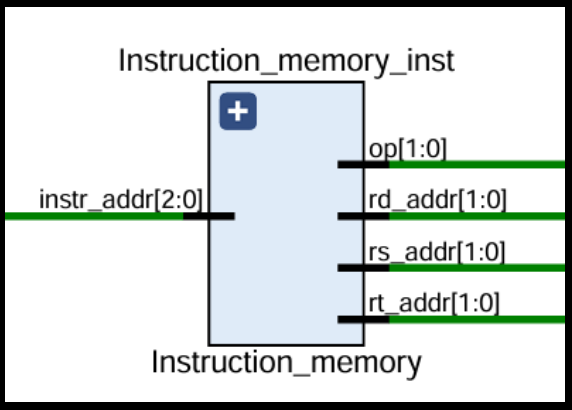
▪ Code

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

entity Program_counter is
    port (
        clk    : in std_logic;
        next_instruction : out STD_LOGIC_VECTOR(2 downto 0)

    );
end entity Program_counter;

architecture rtl of Program_counter is

    --Storing the value to be displayed in the output
    signal current_signal : std_logic_vector(2 downto 0) := "000";

begin
    process (clk)
    begin
        if falling_edge(clk) then
            current_signal <= STD_LOGIC_VECTOR(unsigned(current_signal) +
TO_UNSIGNED(1,3));
        end if;
    end process;

    next_instruction <= current_signal;
end architecture;
```

2-Instruction Memory:

This module simulates a small read-only memory (ROM) that stores 8 predefined 8-bit instructions. Each instruction is decoded into four 2-bit segments:

- **op**: ALU operation code (bit 7 & 6)
- **rs_addr**: source register (A) (bit 5 & 4)
- **rt_addr**: target register (B) (bit 3 & 2)
- **rd_addr**: destination register (bit 1 & 0)



Behavior:

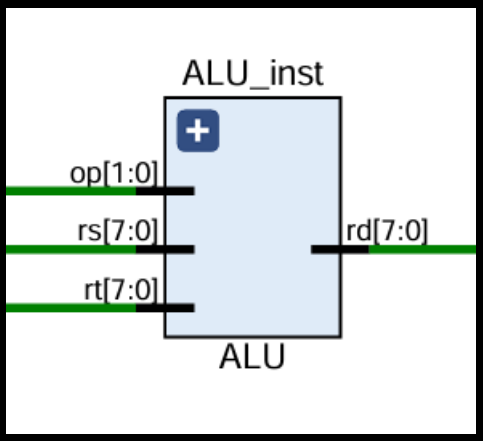
- instr_addr (3-bit input) selects one of the 8 instructions.
- The selected instruction is split into segments and routed to the respective outputs (op, rs_addr, etc.).

Code

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

entity Instruction_memory is
    port (
        instr_addr : in std_logic_vector(2 downto 0);
        op : out std_logic_vector(1 downto 0);
        rs_addr : out std_logic_vector(1 downto 0);       --Address of source register
        rt_addr : out std_logic_vector(1 downto 0);       --Address of target register
        rd_addr : out std_logic_vector(1 downto 0)        --Address of destination register
    );
end entity Instruction_memory;

architecture rtl of Instruction_memory is
    --Construction of the memory which is 8-bit long and each row is 8-bit wide
    type instruction_set is array (0 to 7) of std_logic_vector(7 downto 0);
        constant instr : instruction_set := (
            "01000010",
            "11010101",
            "11101011",
            "01000111",
            "10101100",
            "00000000",
            "00000000",
            "00000000"
        );
begin
    op <= instr (to_integer(unsigned(instr_addr))) (7 downto 6);
    rs_addr <= instr (to_integer(unsigned(instr_addr))) (5 downto 4);
    rt_addr <= instr (to_integer(unsigned(instr_addr))) (3 downto 2);
    rd_addr <= instr (to_integer(unsigned(instr_addr))) (1 downto 0);
end architecture;
```

3- ALU (Arithmetic Logic Unit

Performs basic arithmetic/logic operations between two 8-bit inputs (rs and rt) based on the 2-bit op code.



Supported Operations:

- ➔ "00": AND
- ➔ "01": Addition
- ➔ "10": Subtraction
- ➔ "11": Addition with special behavior (used in addi-like instruction)

Output:

- Result is assigned to rd.

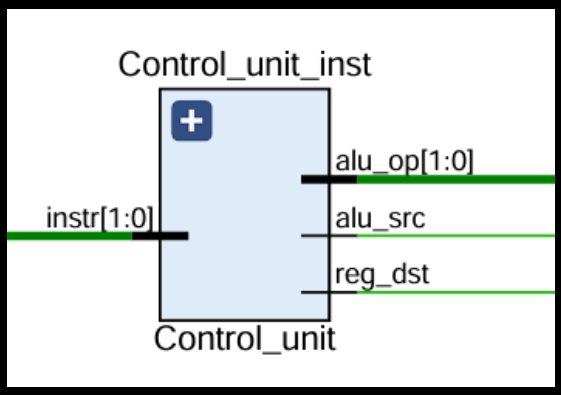
Code

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;        --unsigned variables

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

architecture rtl of ALU is
    signal result : std_logic_vector(7 downto 0);
begin

process (op , rs , rt)
    begin
        if (op = "00") then      --AND operation
            result <= rs and rt;
        elsif (op = "01") then   --Addition operation
            result <= rs + rt ;
        elsif (op = "10") then   --Subtraction operation
            result <= rs - rt ;
```

```
        elsif (op = "11") then   --Addition operation 2 and this operation is done using the
control unit
            result <= rs + rt ;      -- Adding A with the address of the destination register
(8-bit + 2-bit) and the result will be saved in register2
        end if;
    end process;

    --Assigning the result to the output
    rd <= result;

end architecture;
```

4- Control Unit:

Controls the datapath behavior based on the opcode of the current instruction.



Inputs:

- instr: 2-bit ALU opcode.

Outputs:

- alu_op: opcode passed to the ALU.
- alu_src: select signal for mux1 (selects ALU input operand: reg2 address vs destination register address).
- reg_dst: select signal for mux0 (selects destination register: rd vs register2 value).

Behavior:

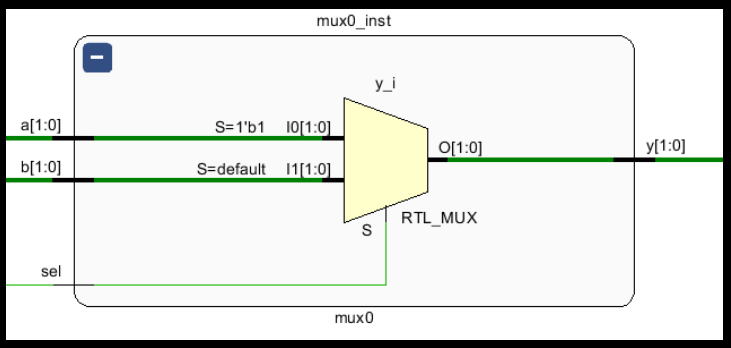
- For normal ops ("00", "01", "10"), both muxes are set to default.
- For special op "11" (addi), muxes are reconfigured.

- Code

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

entity Control_unit is
    port (
        instr : in STD_LOGIC_VECTOR (1 downto 0);      --input to choose which operation
        alu_op : out STD_LOGIC_VECTOR (1 downto 0);    --same opcode will be output here but
the difference is in the muxs
        alu_src : out std_logic;                       --selection line of mux1
        reg_dst : out std_logic                        --selection line of mux0
    );
end entity Control_unit;

architecture rtl of Control_unit is

begin
    --assinging the output to be the same as the input which will be the to ALU opcode
    with instr select
        alu_op  <=  "00" when "00",       --AND
                    "01" when "01",       --Add
                    "10" when "10",       --Sub
                    "11" when others;     --Addi


    --Controlling the mux0 selection line
    with instr select
        reg_dst <=  '1' when "11",       --Addi
                    '0' when others;


    --Controlling the mux1 selection line
    with instr select
        alu_src <=  '1' when "11",       --Addi
                    '0' when others;

end architecture rtl;
```

5- Mux0:

Selects the destination register address.



Inputs:

- a: address of register2
- b: default rd register address
- sel: selector (1 for a, 0 for b)

Output:

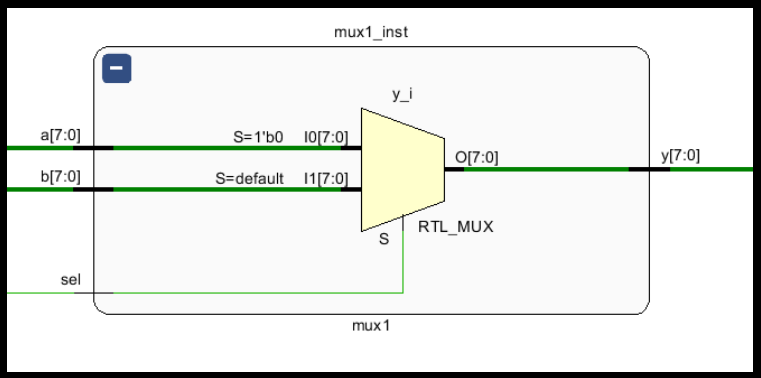
- y: selected register address and pass it to the registers file

Code

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

--This mux chooses the address of which register the data will be saved inside
entity mux0 is
    port (
        a : in STD_LOGIC_VECTOR (1 downto 0);        --address of reg2
        b : in STD_LOGIC_VECTOR (1 downto 0);        --default destination address
        sel : in std_logic;
        y : out STD_LOGIC_VECTOR (1 downto 0)        --destination address
    );
end entity mux0;


architecture behavioral of mux0 is

begin
    process (sel , a , b)
    begin
        if (sel = '1') then    --This operation is done in case of opcode = 11
            y <= a;
        else
            y <= b;          -- default operations will be done while opcode = 00 or 01 or 10
        end if;
    end process;
end architecture behavioral;
```

6- Mux1:

Chooses the second operand to send to the ALU.



Inputs:

- a: value from register2
- b: sign-extended constant (from rd_addr)
- sel: selector (1 for b, 0 for a)

Output:

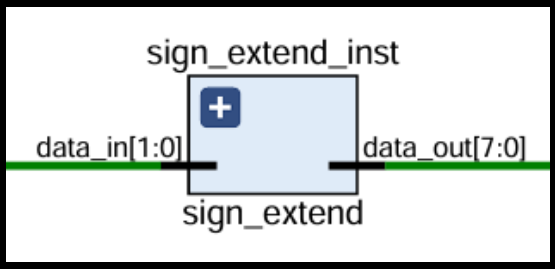
- y: ALU operand

Code

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

--This mux will choose the value of the data to be added together
entity mux1 is
    port (
        a : in STD_LOGIC_VECTOR (7 downto 0);        --default data
        b : in STD_LOGIC_VECTOR (7 downto 0);        --constant data
        sel : in std_logic;
        y : out STD_LOGIC_VECTOR (7 downto 0)        --output data
    );
end entity mux1;
architecture behavioral of mux1 is
begin
    process (sel , a , b)
    begin
        if (sel = '0') then    --default operations (opcode = 00 or 01 or 10)
            y <= a;
        else
            y <= b;            --when opcode = 11
        end if;
    end process;
end architecture behavioral;
```

7- Sign extender:

Extends a 2-bit value (rd_addr) to 8 bits by padding 6 leading zeros.



Use Case:

Used to convert destination register address into a value that can be added to a register (for immediate-like instructions such as addi).

- Code

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

entity sign_extend is
    port (
        data_in : in STD_LOGIC_VECTOR (1 downto 0);
        data_out : out STD_LOGIC_VECTOR (7 downto 0)
    );
end entity sign_extend;

architecture rtl of sign_extend is
begin
    data_out <= "000000" & data_in;
end architecture rtl;
```

Top Module

- Code

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

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

architecture rtl of CPU_top is

--Components declaration
    component Program_counter is
        port (
            clk   : in std_logic;
            next_instruction : out STD_LOGIC_VECTOR(2 downto 0)

        );
    end component Program_counter;


    component Instruction_memory is
        port (
            instr_addr : in std_logic_vector(2 downto 0);
            op : out std_logic_vector(1 downto 0);
            rs_addr : out std_logic_vector(1 downto 0);
            rt_addr : out std_logic_vector(1 downto 0);
            rd_addr : out std_logic_vector(1 downto 0)
        );
    end component Instruction_memory;


    component Registers_file is
        port (
            clk   : in std_logic;
            rs_addr : in std_logic_vector(1 downto 0);
            rt_addr : in std_logic_vector(1 downto 0);
            rd_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 Registers_file;
```

```
    component ALU is
        port (
            op : in std_logic_vector(1 downto 0);
            rs , rt : in std_logic_vector(7 downto 0);
            rd : out std_logic_vector(7 downto 0)
        );
    end component ALU;


    component Control_unit 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
        );
    end component Control_unit;


    component mux0 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 mux0;


    component mux1 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 mux1;


    component sign_extend is
        port (
            data_in : in STD_LOGIC_VECTOR (1 downto 0);
            data_out : out STD_LOGIC_VECTOR (7 downto 0)
        );
    end component sign_extend;


--Signals declaration
    signal opcode_sig : std_logic_vector(1 downto 0);
    signal rs_sig , rt_sig : std_logic_vector(7 downto 0);
    signal instr_sig : std_logic_vector(1 downto 0);
```

```
    signal alu_dst_sig, alu_src_sig : STD_LOGIC;
    signal rt_addr_sig , rd_addr_sig : std_logic_vector(1 downto 0);
    signal rt_val , sign_ext_out: std_logic_vector(7 downto 0);
    signal mux0_out , rs_addr_sig : std_logic_vector(1 downto 0);
    signal instr_addr_sig : std_logic_vector(2 downto 0);
    signal out_val : std_logic_vector(7 downto 0);

begin

    value <= out_val;

--Components instantiation
    ALU_inst:
    ALU port map(
        op => opcode_sig,
        rs => rs_sig,
        rt => rt_sig,
        rd => out_val
    );


    Control_unit_inst: Control_unit
    port map(
        instr => instr_sig,
        alu_op => opcode_sig,
        alu_src => alu_src_sig,
        reg_dst => alu_dst_sig
    );

    Instruction_memory_inst: Instruction_memory
    port map(
        instr_addr => instr_addr_sig,
        op => instr_sig,
        rs_addr => rs_addr_sig,
        rt_addr => rt_addr_sig,
        rd_addr => rd_addr_sig
    );

    mux0_inst: mux0
    port map(
        a => rt_addr_sig,
        b => rd_addr_sig,
        sel => alu_dst_sig,
        y => mux0_out
    );

    mux1_inst: mux1
    port map(
        a => rt_val,
        b => sign_ext_out,
```

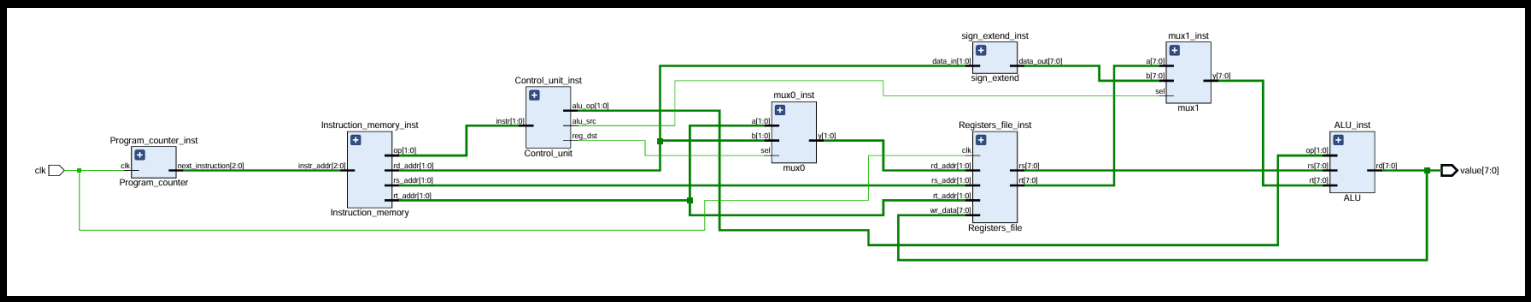
```
        sel => alu_src_sig,
        y => rt_sig
    );

    Program_counter_inst: Program_counter
    port map(
        clk => clk,
        next_instruction => instr_addr_sig
    );

    Registers_file_inst: Registers_file
    port map(
        clk => clk,
        rs_addr => rs_addr_sig,
        rt_addr => rt_addr_sig,
        rd_addr => mux0_out,
        wr_data => out_val,
        rs => rs_sig,
        rt => rt_val
    );

    sign_extend_inst: sign_extend
    port map(
        data_in => rd_addr_sig,
        data_out => sign_ext_out
    );
end architecture;
```

Testbench

- Code

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


entity CPU_tb is
end entity CPU_tb;

architecture rtl of CPU_tb is
    component CPU_top is
        port (
            clk : in std_logic;
            value : out std_logic_vector(7 downto 0)
        );
    end component CPU_top;

    signal clk_tb : STD_LOGIC := '0';
    signal value_tb : std_logic_vector(7 downto 0) := "00000000";

begin

    dut:CPU_top
    port map(
        clk => clk_tb,
        value => value_tb
    );

    clk_generation: process
    begin
        while true loop
            wait for 5 ns;
            clk_tb <= not clk_tb;
        end loop;
    end process clk_generation;


    stim_proc: process
    begin
        wait for 200 ns;
        wait;
    end process stim_proc;
end architecture;
```

Schematic



Simulation output