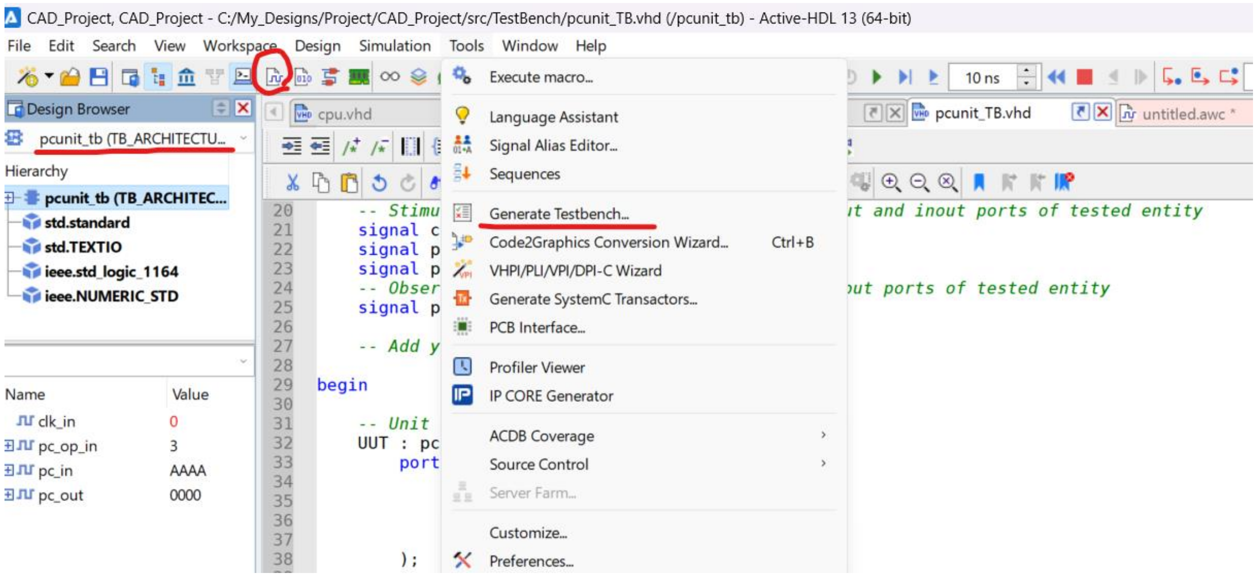
سانیا مسعودی/فاطمه علیزاده بهار 1403

CAD PROJECT VHDL

```
-- Libraries must import
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
-- Create entity
entity EntityName is
  port ( InputPortName : in STD_LOGIC; -- STD_LOGIC used for 1 bit
      InputPortName: in STD_LOGIC_VECTOR (1 downto 0); -- STD_LOGIC_VECTOR(n downto 0) used for n bits
      OutputPortName: out STD_LOGIC_VECTOR (15 downto 0));
end EntityName;
-- Create architecture to manage behavior of our entity
architecture Behavioral of EntityName  is
        -- Create type for your entity
        --  (for example for ram 256 address * 16 bit):
         type TypeName is array(0 to 7) of STD_LOGIC_VECTOR (15 downto 0);
        -- Create signal for your type/or for yourself to manage better process
        signal SignalName: STD_LOGIC_VECTOR (15 downto 0) := x"0000";
begin
 -- Create process
  process (clk_in)
  begin
    if rising_edge(clk_in) then
      case inputName is
        when "00" => SignalName <= x"0000";
        when "01" => SignalName  <= STD_LOGIC_VECTOR(unsigned(SignalName) + 1);
        when others =>
      end case;
   end if;
end process;
  OutputName <= SignalName;
end Behavioral;
```

Test bench

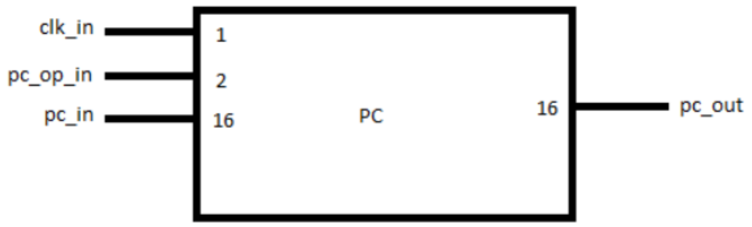


```
library ieee;
use ieee.NUMERIC_STD.all;
use ieee.std_logic_1164.all;
        -- Add your library and packages declaration here ...
entity pcunit_tb is
end pcunit_tb;
architecture TB_ARCHITECTURE of pcunit_tb is
        -- Component declaration of the tested unit
        component pcunit
        port(
                clk_in : in STD_LOGIC;
                pc_op_in : in STD_LOGIC_VECTOR(1 downto 0);
                pc_in : in STD_LOGIC_VECTOR(15 downto 0);
                pc_out : out STD_LOGIC_VECTOR(15 downto 0) );
        end component;
        -- Stimulus signals - signals mapped to the input and inout ports of tested entity
        signal clk_in : STD_LOGIC;
        signal pc_op_in : STD_LOGIC_VECTOR(1 downto 0);
        signal pc_in : STD_LOGIC_VECTOR(15 downto 0);
        -- Observed signals - signals mapped to the output ports of tested entity
        signal pc_out : STD_LOGIC_VECTOR(15 downto 0);
        -- Add your code here ...
```

```
begin
        -- Unit Under Test port map
        UUT : pcunit
                port map (
                        clk_in => clk_in,
                        pc_op_in => pc_op_in,
                        pc_in => pc_in,
                        pc_out => pc_out
                );

        -- Add your stimulus here ...
  clk_process : process
  begin
    clk_in <= '0';
    wait for 10 ns;
    clk_in <= '1';
    wait for 10 ns;
  end process;

  main_process: process
  begin
    pc_op_in <= "00";
    pc_in <= (others => '0');
    wait for 100 ns;
    pc_op_in <= "01";
    wait for 20 ns;
    pc_op_in <= "10";
    pc_in <= x"AAAA";
    wait for 20 ns;
    pc_op_in <= "00";
    wait for 20 ns;
    pc_op_in <= "11";
    wait for 20 ns;
    wait;
  end process;
end TB_ARCHITECTURE;
configuration TESTBENCH_FOR_pcunit of pcunit_tb is
        for TB_ARCHITECTURE
                for UUT : pcunit
                        use entity work.pcunit(behavioral);
                end for;
        end for;
end TESTBENCH_FOR_pcunit;
```

Pc Unit:

```
when "00" => -- reset
when "01" => -- increment
when "10" => -- branch
when "11" => -- NOP
```



```
case pc_op_in is
    when "00" =>
     pc_out_signal <= x"0000";
    when "01" =>
        pc_out_signal <= STD_LOGIC_VECTOR(unsigned(pc_out_signal) + 1);
    when "10" =>
        pc_out_signal <= pc_in;
    when "11" =>
    when others =>
end case;
```

Control Unit:

when "000001" => -- Fetch

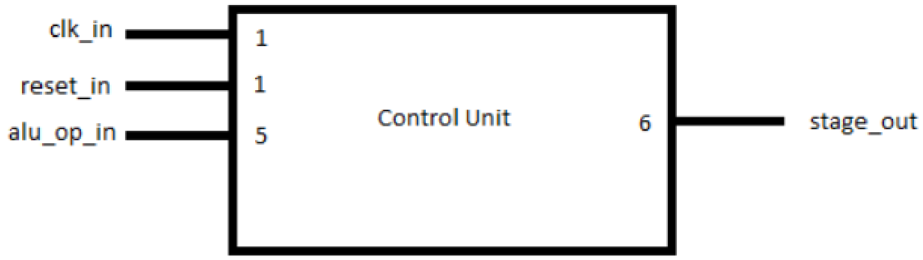
when "000010" => -- Decode

when "000100" => -- Reg read

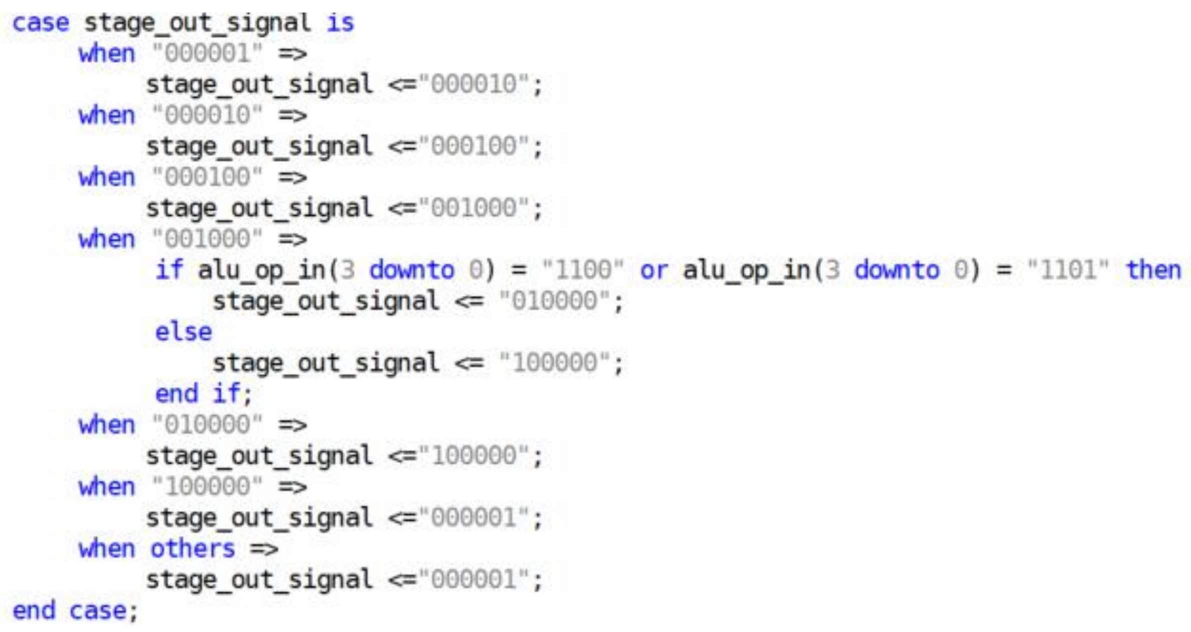
when "001000" => -- Execute

when "010000" => -- Memory

when "100000" => -- Reg write

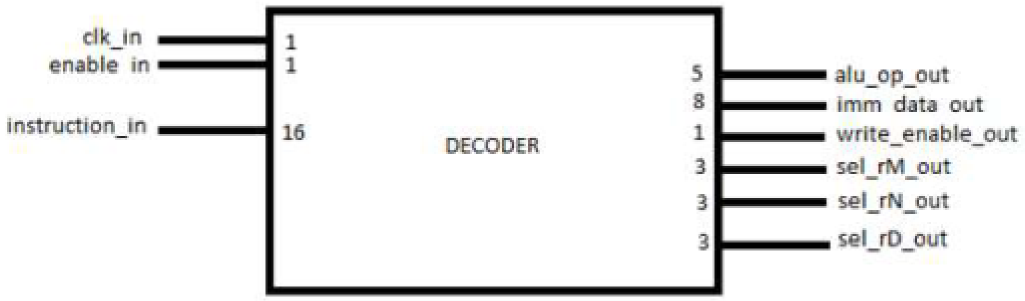


Instruction	Form	Implementation	Condition bit	OPCODE
ST	URR	memory(rM) = rN	c: N/A	1101
LD	RRU	rD = memory(rM)	c: N/A	1100



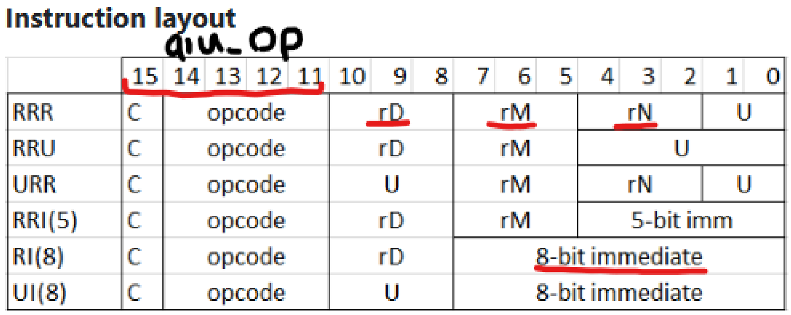
```
case stage_out_signal is
     when "000001" =>
          stage_out_signal <="000010";
     when "000010" =>
          stage_out_signal <="000100";
     when "000100" =>
          stage_out_signal <="001000";
     when "001000" =>
           if alu_op_in(3 downto 0) = "1100" or alu_op_in(3 downto 0) = "1101" then
               stage_out_signal <= "010000";
           else
               stage_out_signal <= "100000";
           end if;
     when "010000" =>
          stage_out_signal <="100000";
     when "100000" =>
          stage_out_signal <="000001";
     when others =>
          stage_out_signal <="000001";
end case;
```

Decoder:



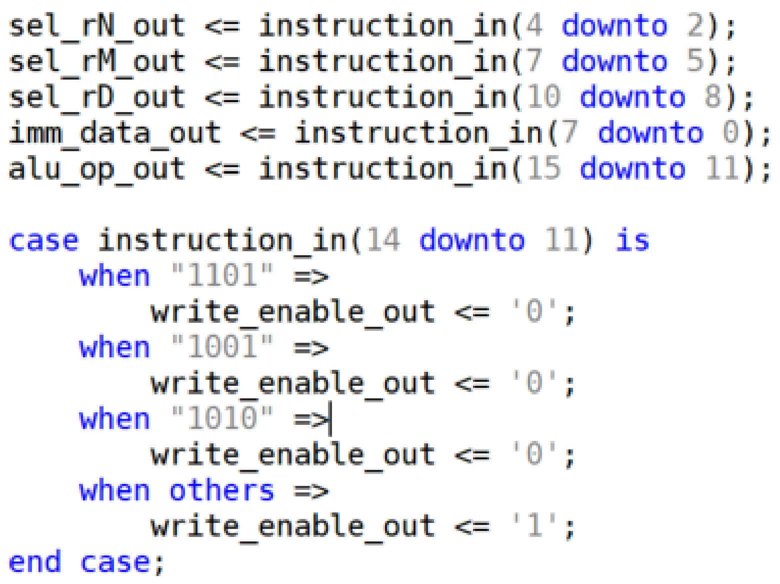
• Instruction	Form	Implementation	Condition bit	OPCODE
ADD	RRR	rD = rM + rN	c: 1/0 = signed/unsigned	0000
SUB	RRR	rD = rM - rN	c: 1/0 = signed/unsigned	0001
NOT	RRU	rD = not rN	c: N/A	0010
AND	RRR	rD = rM and rN	c: N/A	0011
OR	RRR	rD = rM or rN	c: N/A	0100
XOR	RRR	rD = rM xor rN	c: N/A	0101
LSL	RRI(5)	rD = rM << rN	c: N/A	0110
LSR	RRI(5)	rD = rM >> rN	c: N/A	0111
CMP	RRR	rD = cmp(rM, rN)	c: 1/0 = signed/unsigned	1000
B	UI(8)	PC = rM or 8-bit immediate	c: 1/0 = rM/8-bit immediate	1001
BEQ	URR	PC = rM conditional on rN	c: N/A	1010
IMMEDIATE	RI(8)	rD = 8-bit immediate	c: 1/0 = upper/lower 8-bits	1011
LD	RRU	rD = memory(rM)	c: N/A	1100
ST	URR	memory(rM) = rN	c: N/A	1101

Decoder:



Instruction layout

	15	14 13 12 11	10 9 8	7 6 5	4 3 2	1 0
RRR	C	opcode	rD	rM	rN	U
RRU	C	opcode	rD	rM	U	
URR	C	opcode	U	rM	rN	U
RRI(5)	C	opcode	rD	rM	5-bit imm	
RI(8)	C	opcode	rD	8-bit immediate		
UI(8)	C	opcode	U	8-bit immediate		



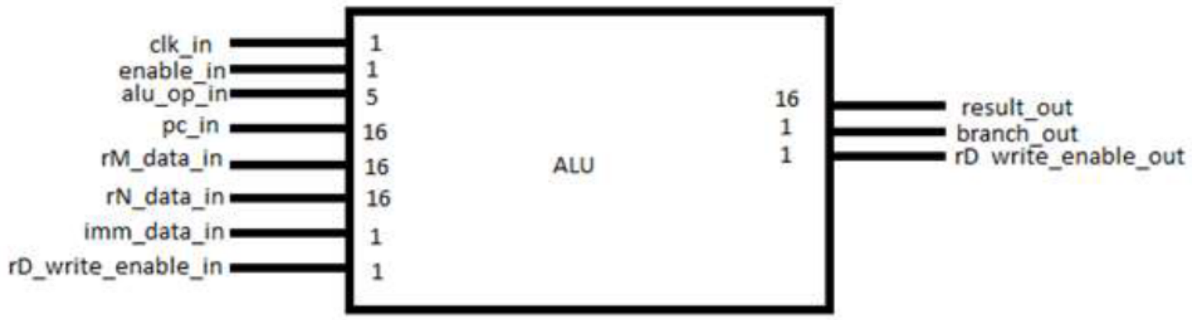
```
sel_rN_out <= instruction_in(4 downto 2);
sel_rM_out <= instruction_in(7 downto 5);
sel_rD_out <= instruction_in(10 downto 8);
imm_data_out <= instruction_in(7 downto 0);
alu_op_out <= instruction_in(15 downto 11);

case instruction_in(14 downto 11) is
    when "1101" =>
        write_enable_out <= '0';
    when "1001" =>
        write_enable_out <= '0';
    when "1010" =>
        write_enable_out <= '0';
    when others =>
        write_enable_out <= '1';
end case;
```

ST(1101): This usually means “store”. A st instruction typically writes data from a register to memory

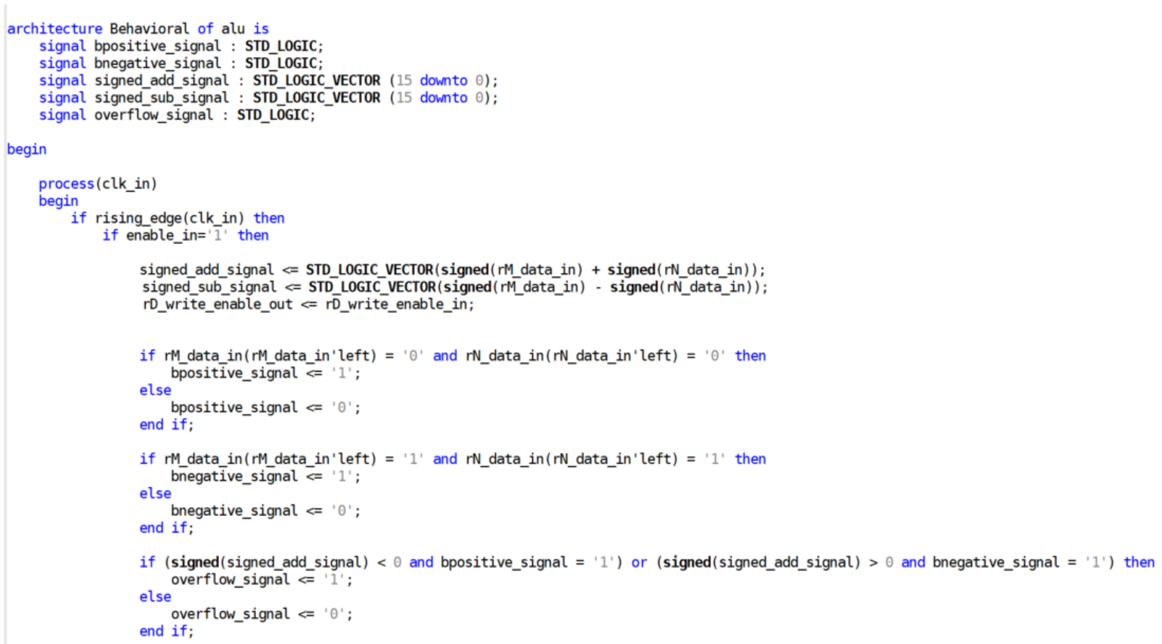
B(1001): This is often shorthand for “branch”. A b instruction generally causes the program to jump to a different address unconditionally, altering the sequential flow of control.

BEQ(1010): This stands for “branch if equal”. In many instruction sets, a beq instruction is used to change the flow of execution to a different part of the program if the previous conditions or comparisons evaluated to equal (often set by a flag in the status register).

ALU:

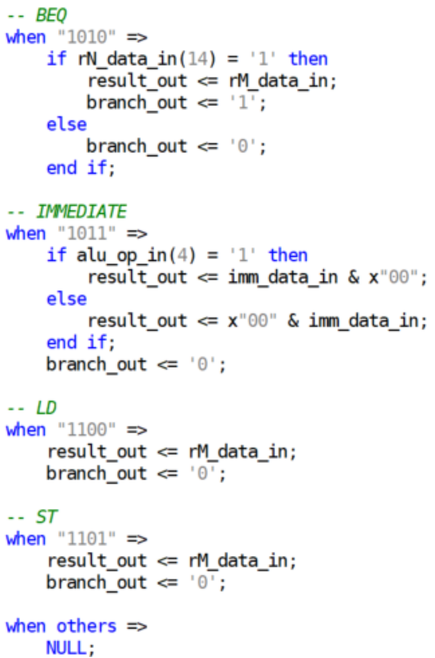


1) Define signal (bothPositive,bothNegative,signedAdd,signedSub,overflow)

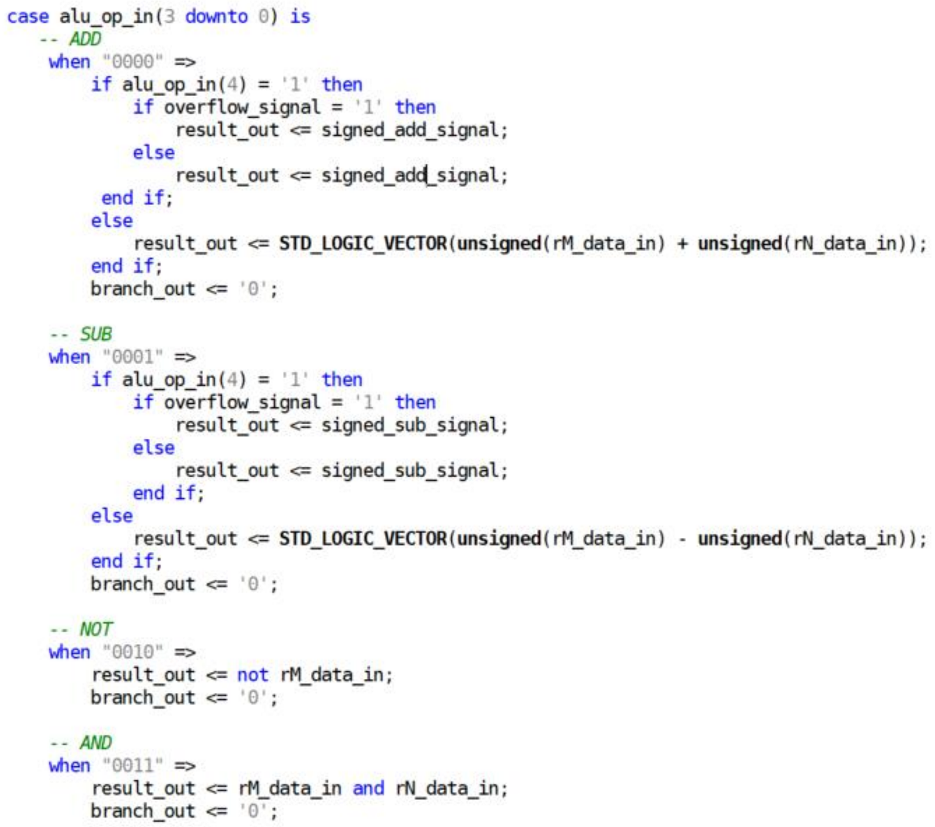


```
architecture Behavioral of alu is
    signal bpositive_signal : STD_LOGIC;
    signal bnegative_signal : STD_LOGIC;
    signal signed_add_signal : STD_LOGIC_VECTOR (15 downto 0);
    signal signed_sub_signal : STD_LOGIC_VECTOR (15 downto 0);
    signal overflow_signal : STD_LOGIC;

begin

    process(clk_in)
    begin
        if rising_edge(clk_in) then
            if enable_in='1' then

                signed_add_signal <= STD_LOGIC_VECTOR(signed(rM_data_in) + signed(rN_data_in));
                signed_sub_signal <= STD_LOGIC_VECTOR(signed(rM_data_in) - signed(rN_data_in));
                rD_write_enable_out <= rD_write_enable_in;


                if rM_data_in(rM_data_in'left) = '0' and rN_data_in(rN_data_in'left) = '0' then
                    bpositive_signal <= '1';
                else
                    bpositive_signal <= '0';
                end if;

                if rM_data_in(rM_data_in'left) = '1' and rN_data_in(rN_data_in'left) = '1' then
                    bnegative_signal <= '1';
                else
                    bnegative_signal <= '0';
                end if;

                if (signed(signed_add_signal) < 0 and bpositive_signal = '1') or (signed(signed_add_signal) > 0 and bnegative_signal = '1') then
                    overflow_signal <= '1';
                else
                    overflow_signal <= '0';
                end if;
```

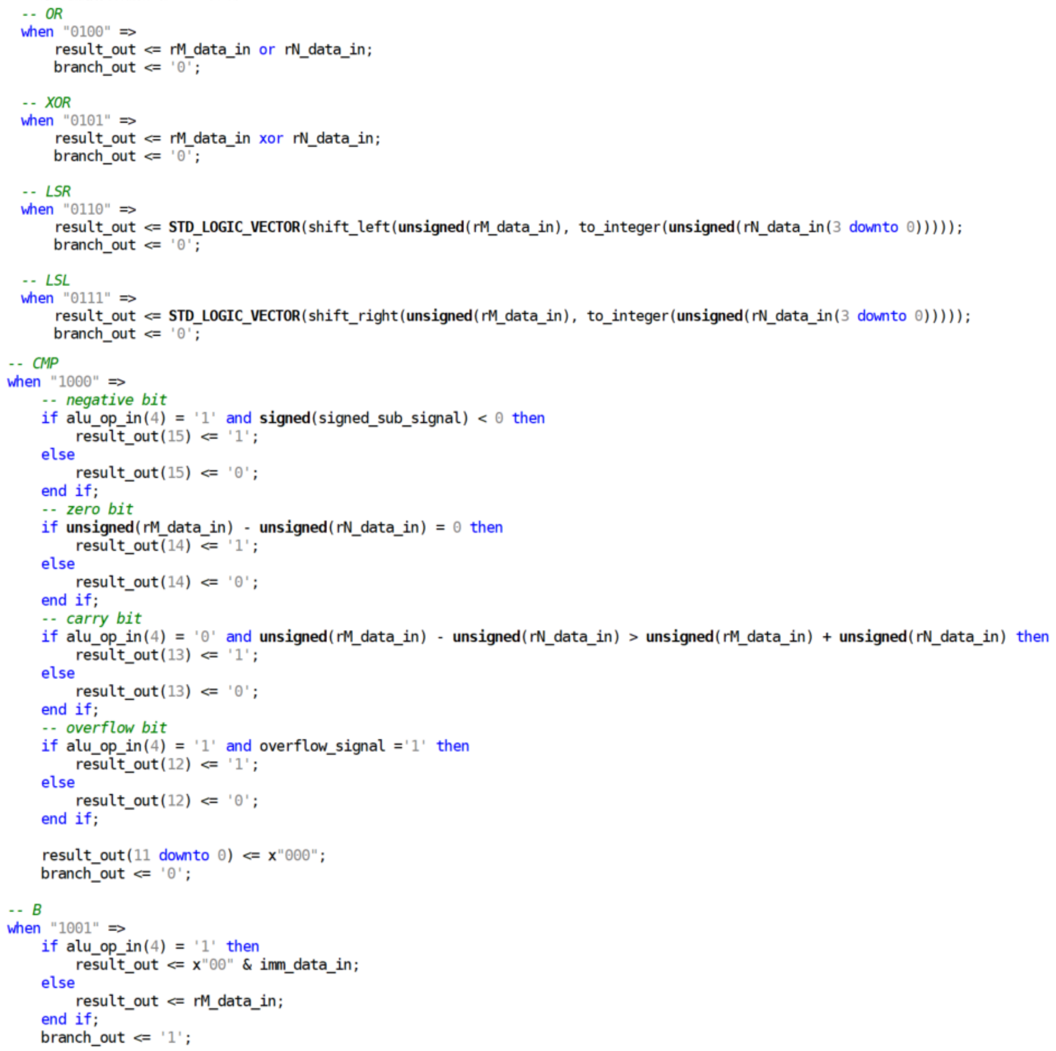
2) Define instructions based attention to signals



```
-- BEQ
when "1010" =>
    if rN_data_in(14) = '1' then
        result_out <= rM_data_in;
        branch_out <= '1';
    else
        branch_out <= '0';
    end if;

-- IMMEDIATE
when "1011" =>
    if alu_op_in(4) = '1' then
        result_out <= imm_data_in & x"00";
    else
        result_out <= x"00" & imm_data_in;
    end if;
    branch_out <= '0';

-- LD
when "1100" =>
    result_out <= rM_data_in;
    branch_out <= '0';

-- ST
when "1101" =>
    result_out <= rM_data_in;
    branch_out <= '0';

when others =>
    NULL;
```



```
case alu_op_in(3 downto 0) is
   -- ADD
    when "0000" =>
        if alu_op_in(4) = '1' then
            if overflow_signal = '1' then
                result_out <= signed_add_signal;
            else
                result_out <= signed_add_signal;
         end if;
        else
            result_out <= STD_LOGIC_VECTOR(unsigned(rM_data_in) + unsigned(rN_data_in));
        end if;
        branch_out <= '0';

    -- SUB
    when "0001" =>
        if alu_op_in(4) = '1' then
            if overflow_signal = '1' then
                result_out <= signed_sub_signal;
            else
                result_out <= signed_sub_signal;
            end if;
        else
            result_out <= STD_LOGIC_VECTOR(unsigned(rM_data_in) - unsigned(rN_data_in));
        end if;
        branch_out <= '0';

    -- NOT
    when "0010" =>
        result_out <= not rM_data_in;
        branch_out <= '0';

    -- AND
    when "0011" =>
        result_out <= rM_data_in and rN_data_in;
        branch_out <= '0';
```



```
    -- OR
    when "0100" =>
        result_out <= rM_data_in or rN_data_in;
        branch_out <= '0';

    -- XOR
    when "0101" =>
        result_out <= rM_data_in xor rN_data_in;
        branch_out <= '0';

    -- LSR
    when "0110" =>
        result_out <= STD_LOGIC_VECTOR(shift_left(unsigned(rM_data_in), to_integer(unsigned(rN_data_in(3 downto 0)))));
        branch_out <= '0';

    -- LSL
    when "0111" =>
        result_out <= STD_LOGIC_VECTOR(shift_right(unsigned(rM_data_in), to_integer(unsigned(rN_data_in(3 downto 0)))));
        branch_out <= '0';
    -- CMP
    when "1000" =>
        -- negative bit
        if alu_op_in(4) = '1' and signed(signed_sub_signal) < 0 then
            result_out(15) <= '1';
        else
            result_out(15) <= '0';
        end if;
        -- zero bit
        if unsigned(rM_data_in) - unsigned(rN_data_in) = 0 then
            result_out(14) <= '1';
        else
            result_out(14) <= '0';
        end if;
        -- carry bit
        if alu_op_in(4) = '0' and unsigned(rM_data_in) - unsigned(rN_data_in) > unsigned(rM_data_in) + unsigned(rN_data_in) then
            result_out(13) <= '1';
        else
            result_out(13) <= '0';
        end if;
        -- overflow bit
        if alu_op_in(4) = '1' and overflow_signal ='1' then
            result_out(12) <= '1';
        else
            result_out(12) <= '0';
        end if;

        result_out(11 downto 0) <= x"000";
        branch_out <= '0';

    -- B
    when "1001" =>
        if alu_op_in(4) = '1' then
            result_out <= x"00" & imm_data_in;
        else
            result_out <= rM_data_in;
        end if;
        branch_out <= '1';
```