Madan Bhandari Memorial College

Experiment vi: ALU

Objective:

1. To implement ALU using VHDL

Theory:

The Arithmetic Logic Unit (ALU) is the heart of any CPU. An ALU performs three kinds of operations, i.e.

- Arithmetic operations such as Addition/Subtraction,
- Logical operations such as AND, OR, etc. and
- Data movement operations such as Load and Store

ALU derives its name because it performs arithmetic and logical operations. A simple ALU design is constructed with Combinational circuits. ALUs that perform multiplication and division are designed around the circuits developed for these operations while implementing the desired algorithm. It is basically the actual data processing element within the central processing unit (CPU) in a computing system. It performs all the arithmetic and logical operations and forms the backbone of modern computer technology.

Architecture:

```
--design for alu
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.NUMERIC_STD.all;
entity ALU is
 generic ( constant N: natural:=1); -- number of shifted or rotated bits
 Port (
 A, B: in STD_LOGIC_VECTOR (7 downto 0); -- 2 inputs 8-bit
 ALU_Sel: in STD_LOGIC_VECTOR (3 downto 0); -- 1 input 4-bit for selecting function
 ALU_Out: out STD_LOGIC_VECTOR (7 downto 0); -- 1 output 8-bit
 Carryout: out std_logic; -- Carryout flag
 );
end ALU;
architecture Behavioral of ALU is
signal ALU_Result: std_logic_vector (7 downto 0);
signal tmp: std_logic_vector (8 downto 0);
begin
 process(A,B,ALU_Sel)
 begin
 case(ALU_Sel) is
 when "0000" => -- Addition
 ALU_Result <= A + B ;
 when "0001" => -- Subtraction
 ALU_Result <= A - B ;
 when "0010" => -- Multiplication
 ALU_Result<=std_logic_vector(to_unsigned((to_integer(unsigned(A)) *
to_integer(unsigned(B))),8)) ;
 when "0011" => -- Division
 ALU_Result <= std_logic_vector(to_unsigned(to_integer(unsigned(A)) /
```

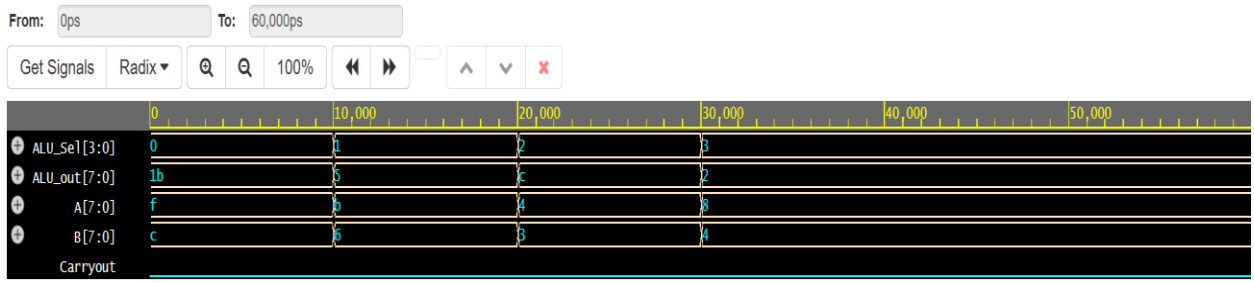
```
to_integer(unsigned(B)),8)) ;
 when "0100" => -- Logical shift left
 ALU_Result <= std_logic_vector(unsigned(A) sll N);
 when "0101" => -- Logical shift right
 ALU_Result <= std_logic_vector(unsigned(A) srl N);
 when "0110" => -- Rotate left
 ALU_Result <= std_logic_vector(unsigned(A) rol N);
 when "0111" => -- Rotate right
 ALU_Result <= std_logic_vector(unsigned(A) ror N);
 when "1000" => -- Logical and
 ALU_Result <= A and B;
 when "1001" => -- Logical or
 ALU_Result <= A or B;
 when "1010" => -- Logical xor
 ALU_Result <= A xor B;
 when "1011" => -- Logical nor
 ALU_Result <= A nor B;
 when "1100" => -- Logical nand
 ALU_Result <= A nand B;
 when "1101" => -- Logical xnor
 ALU_Result <= A xnor B;
 when "1110" => -- Greater comparison
 if(A>B) then
 ALU_Result <= x"01" ;
 else
 ALU_Result <= x"00" ;
 end if;when "1111" => -- Equal comparison
 if(A=B) then
 ALU_Result <= x"01" ;
 else
 ALU_Result <= x"00" ;
 end if;
 when others => ALU_Result <= A + B ;
 end case;
end process;
ALU_Out <= ALU_Result; -- ALU out
tmp <= ('0' & A) + ('0' & B);
Carryout <= tmp(8); -- Carryout flag
end Behavioral;
```

Source code:

```
 -- Testbench for alu adder
library IEEE;
use IEEE.std_logic_1164.all;

entity testbench is
  -- empty
end testbench;

architecture tb of testbench is

  -- DUT component
  component ALU is
 Port (
 A, B: in STD_LOGIC_VECTOR (7 downto 0); -- 2 inputs 8-bit
 ALU_Sel: in STD_LOGIC_VECTOR (3 downto 0); -- 1 input 4-bit for selecting function
```

```
 ALU_Out: out STD_LOGIC_VECTOR (7 downto 0); -- 1 output 8-bit
 Carryout: out std_logic; -- Carryout flag
 );

  end component;

  signal A, B: std_logic_vector(7 downto 0);
  signal ALU_Sel: std_logic_vector(3 downto 0);
  signal ALU_out: std_logic_vector(7 downto 0);
  signal Carryout: std_logic;

begin

  -- Connect DUT
  DUT: ALU
    port map (
      A => A,
      B => B,
      ALU_Sel => ALU_Sel,
      ALU_out => ALU_out,
      Carryout => Carryout
    );

  -- Test process
  process
  begin
    -- Test cases
   A <= "00001111";
   B <= "00001100";
   ALU_Sel<= "0000";
   wait for 10 ns;  -- Expected output: S_out = "0000", Cout_out = '0'


    A <= "00001011";
    B <= "00000110";
    ALU_Sel <= "0001";
    wait for 10 ns;  -- Expected output: S_out = "0010", Cout_out = '0'

    A <= "00000100";
    B <= "00000011";
    ALU_Sel <= "0010";
    wait for 10 ns;
     A <= "00001000";
    B <= "00000100";
    ALU_Sel <= "0011";
    wait for 10 ns;

    -- End of simulation
    wait for 20 ns;
    assert false report "End of simulation" severity failure;
  end process;

end tb;
```

Output:



Conclusion:

In conclusion, designing an ALU in VHDL enables efficient implementation of arithmetic and logic operations in digital systems. VHDL's flexibility and modularity allow for easy testing, simulation, and refinement, ensuring reliable performance. It is a powerful tool for creating high-performance, cost-effective hardware designs.

Experiment vii: 3-Segment Pipeline

Objective:
1. To implement 3-Segment Pipeline using VHDL

Theory:

A 3-segment instruction pipeline in computer architecture divides instruction processing into three stages: fetch, decode, and execute. To perform tasks such as fetching, decoding and execution of instructions, most digital computers with complicated instructions would require an instruction pipeline.

In general, each and every instruction must be processed by the computer in the following order:

1. Fetching the instruction from memory

2. Decoding the obtained instruction

3. Calculating the effective address

4. Fetching the operands from the given memory

5. Execution of the instruction

6. Storing the result in a proper place

Architecture:

```
--design for pipeline
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity pipeline is
    Port (
        a : in integer;
        b : in integer;
        c : in integer;
        clk : in STD_LOGIC;
        y : out integer
    );
end pipeline;

architecture Behavioral of pipeline is
    signal r1, r2, r3, r4, r5 : integer := 0;
begin
    y <= r5;

    process(clk)
    begin
        if rising_edge(clk) then
            -- Pipeline stage 1
            r1 <= a;
            r2 <= b;

            -- Pipeline stage 2 (registered)
            r4 <= r1 + r2;
            r3 <= c;
```

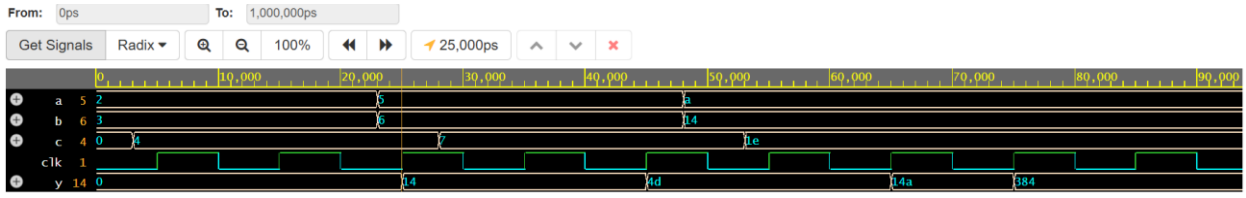
```
            -- Pipeline stage 3 (registered)
            r5 <= r4 * r3;
        end if;
    end process;
end Behavioral;
```

Source code:

```
--testbench for pipeline
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity pipeline_tb is
end pipeline_tb;

architecture Behavioral of pipeline_tb is
    component pipeline
        Port (
            a : in integer;
            b : in integer;
            c : in integer;
            clk : in STD_LOGIC;
            y : out integer
        );
    end component;

    signal a, b, c : integer := 0;
    signal y : integer;
    signal clk : STD_LOGIC := '0';

    constant clk_period : time := 10 ns;

begin
    uut: pipeline port map (
        a => a,
        b => b,
        c => c,
        clk => clk,
        y => y
    );

    -- Clock generation
    clk_process: process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin


        -- Test case 1
        a <= 2;
```

```
        b <= 3;
        wait for 3 ns;
        c <= 4;
        wait for clk_period*2;

        -- Test case 2
        a <= 5;
        b <= 6;
        wait for 5 ns;
        c <= 7;
        wait for clk_period*2;

        -- Test case 3
        a <= 10;
        b <= 20;
        wait for 5 ns;
        c <= 30;
        wait for clk_period*2;

        -- End simulation
        wait;
    end process;
end Behavioral;
```

Output:



Conclusion:

In conclusion, implementing a 3-segment pipeline in VHDL enhances system performance by allowing parallel processing of data in stages. This approach increases throughput and reduces latency by efficiently organizing tasks across different pipeline segments. VHDL provides the necessary tools to model, simulate, and optimize the pipeline, ensuring smooth operation and scalability in complex digital systems.