Annexure – E

VHDL code
-- Module name : PID_controller_top Module

This Module is the top module for PID controller where we have three submodules in this main module.

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity PID_controller_top is
  port ( Reset_in : in std_logic; -- Reset input to Line control block
          ADC_CLK_in : in std_logic; -- 10MHz Clock to the module
          ADC_Addr_in : in std_logic_vector(2 downto 0); -- Switch input for ADC Address
          ADC_data_in : in std_logic_vector(7 downto 0); -- ADC data Input
          Speed_select_in : in std_logic_vector(2 downto 0); -- Switches to select the set speed
          Capture_ctrl_in : in std_logic; -- Switch to control the Data capture
          SOC_out : out std_logic; -- Start of conversion pulse and ALE pulse
          CE_out : out std_logic; -- Output Enable for ADC output Latch Buffer
          CE2_out : out std_logic; -- Output enable for seven segment display 2
          CE3_out : out std_logic; -- Output enable for seven segment display 3
          CE4_out : out std_logic; -- Output enable for seven segment display 4
          LED_out : out std_logic; -- LED output to indicate Lock to set speed
          Address_out : out std_logic_vector(2 downto 0); -- Address output to ADC MUX
          Seven_seg_out : out std_logic_vector(7 downto 0); -- Seven Segment display output
          PID_Data_out : out std_logic_vector(7 downto 0)); -- ADC sampled data for Motor Control

end PID_controller_top;

Architecture PID_controller_top_arch of PID_controller_top is

use ieee.numeric_std.all;
use ieee.std_logic_1164.all;

--This Module is the top module for PID controller where we have three submodules in this main module.

--- Component Declaration ---

Component Motor_Control is
  port ( Reset_in : in std_logic; -- Reset input to Line control block
          ADC_CLK_in : in std_logic; -- 10MHz Clock to the module
          ADC_Addr_in : in std_logic_vector(2 downto 0); -- Switch input for ADC Address
          ADC_data_in : in std_logic_vector(7 downto 0); -- ADC data Input
          Current_speed_capture : in std_logic; -- Current speed capture switch
          Speed_Cap_switch : in std_logic_vector(2 downto 0); -- Switches to select the set speed
          ADC_sum_data_in : in std_logic_vector(7 downto 0); -- ADC data Input
          LED_out : out std_logic; -- LED output to indicate Lock to set speed
          PID_Data_out : out std_logic_vector(7 downto 0)); -- PID data to DAC.
end Component Motor_control;

Component ADC_Data_rd is
  port ( Reset_in : in std_logic; -- Reset input to Line control block
          ADC_CLK_in : in std_logic; -- 10MHz Clock to the module
          ADC_Addr_in : in std_logic_vector(2 downto 0); -- Switch input for ADC Address
          ADC_data_in : in std_logic_vector(7 downto 0); -- ADC data Input
          ADC_data_rdy_in : in std_logic; -- Average Data Input from ADC Data Read Module
          ADC_data_in : in std_logic_vector(7 downto 0); -- ADC data Input
          ADC_AVG_Data_out : out std_logic_vector(7 downto 0); -- Avg data out of 200 samples
          ADC_AVG_Data_rdy_out : out std_logic); -- ADC average data ready
end Component ADC_Data_rd;

Component ADC_Interface is
  port ( Reset_in : in std_logic; -- Reset input to Line control block
          ADC_CLK_in : in std_logic; -- 10MHz Clock to the module
          ADC_Addr_in : in std_logic_vector(2 downto 0); -- Switch input for ADC Address
          ADC_data_in : in std_logic_vector(7 downto 0); -- ADC data Input
          DAC_data_in : in std_logic_vector(7 downto 0); -- Data going to DAC after PID eqn
          SOC_out : out std_logic; -- Start of conversion pulse and ALE pulse
          CE_out : out std_logic; -- Output Enable for seven segment display 1
          CE2_out : out std_logic; -- Output enable for seven segment display 2
          CE3_out : out std_logic; -- Output enable for seven segment display 3
          CE4_out : out std_logic; -- Output enable for seven segment display 4
          Address_out : out std_logic_vector(2 downto 0); -- Address output to ADC MUX
          CLK_Div8_out : out std_logic; -- Clock output for other modules
          ADC_Data_ready : out std_logic; -- ADC Sampled Data Ready signal for Motor control
          ADC_Sampled_data_out : out std_logic_vector(7 downto 0)); -- ADC Sampled_data for Motor Control
end component ADC_Interface;

-- Signal Declaration --

Signal ADC_Sum_data : std_logic_vector(7 downto 0);
Signal ADC_AVG_Data : std_logic_vector(7 downto 0);
Signal ADC_Data_rdy : std_logic;
Signal PID_Data_to_DAC : std_logic_vector(7 downto 0);
Signal CLK_Div8 : std_logic;
Signal ADC_Sampled_data : std_logic_vector(7 downto 0);
begin

PID_Data_out <= PID_Data_to_DAC; -- PID data output for DAC to control the motor.

Motor ctrl_Module : Motor_Control port map (Reset_in, ADC_Clk_in, ADC_Data_rdy, Current_speed_capture, Speed_Set_switch, LED_out, ADC_Raw_data_in, PID_Data_out) => Reset_in, Clk_Div8, ADC_AVG_Data_Rdy, Capture_ctrl_in, Speed_select_in, LED_out, ADC_Avg_Data, PID_Data_to_DAC);

ADC Data rd_Module : ADC_Data_rd port map (Reset_in, ADC_Clk_in, ADC_Data_rdy_in, ADC_Data_capture, ADC_AVG_Data_out, ADC_AVG_Data_Rdy_out) => Reset_in, Clk_Div8, ADC_Data_rdy, Capture_ctrl_in, ADC_Avg_Data, ADC_AVG_Data_Rdy);

ADC Interface_Module : ADC_Interface port map (Reset_in, ADC_Clk_in, ADC_Addrs_in, ADC_data_in, DAC_Data_in, SOC_out, OE_out, OE1_Disip, OE2_Disip, OE3_Disip, OE4_Disip, Address_out, ADC_Div8Out, ADC_data_ready, Seven_seg_out, ADC_Sampled_Data_out) => Reset_in, ADC_Clk_in, ADC_Addrs_in, ADC_data_in, DAC_Data_in, SOC_out, OE_out, OE1_Disip, OE2_Disip, OE3_Disip, OE4_Disip, Address_out, ADC_Div8Out, ADC_data_ready, Seven_seg_out, ADC_Sampled_Data_out);

End controller_top_arch;
This Module is for the ADC Interface for AD 0808 this Module will generate the ADC control signals and latches the valid data from the ADC.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ADC_Interface is
    port ( Reset_in : in std_logic; -- Reset input to Line control block
           ADC_CLK_in : in std_logic; -- 10MHz Clock to the module
           ADC_Addr_in : in std_logic_vector(2 downto 0); -- Switch input for ADC Address
           ADC_data_in : in std_logic_vector(7 downto 0); -- ADC data input
           DAC_Data_in : in std_logic_vector(7 downto 0); -- Data going to DAC after PID equation
           SOC_out : out std_logic; -- Start of conversion pulse and ALE pulse
           OE1_Disp : out std_logic; -- Output Enable for ADC output latch Buffer
           OE2_Disp : out std_logic; -- Output Enable for seven segment display 1
           OE3_Disp : out std_logic; -- Output enable for seven segment display 2
           OE4_Disp : out std_logic; -- Output enable for seven segment display 3
           Address_out : out std_logic_vector(2 downto 0); -- Address output to ADC MUX
           Clk__Div8_out : out std_logic; -- Clock output for other modules
           ADC_Sampled_data_out : out std_logic_vector(7 downto 0); -- ADC sampled data for Motor Control
           $aven_aeg"out : out std_logic_vector(7 downto 0); -- Seven Segment display output
           ADC_Sample_data_reg : out std_logic_vector(7 downto 0)); -- ADC sampled data for Motor Control
    end ADC_Interface;

Architecture ADC_Interface_arch of ADC_Interface is

    type ADC_State is (Resat_ADC, State_ALE, State_SOC, State_EOC_Check, State_EOC, State_QEl, State_OE2);
    Signal Next_state_ADC : ADC_State;

    Component Seven_Segment is
        port ( Reset_in : in std_logic; -- Reset input to Line control block
                ADC_CLK_in : in std_logic; -- 1.25MHz Clock to the module
                ADC_data_in : in std_logic_vector(7 downto 0); -- ADC data input
                OE1_out : out std_logic; -- Output Enable for Display output Latch Buffer
                OE2_out : out std_logic; -- Output Enable for Display output Latch Buffer
                Display_Data_out : out std_logic_vector(7 downto 0)); -- Display data for Seven Segment Display
    end Component Seven_Segment;

    begin
        SOC_out <= ADC_SOC;
        OE2_out <= ADC_SOC;
        OE3_Disp <= OE3_Val; -- Output enable for Seven segment Display 3
        begin
            Signal ADC_OE : Std_logic;
            Signal ADC_ALE : Std_logic;
            Signal ADC_SOC : Std_logic;
            Signal ADC_SOC : Std_logic;
            Signal ADC_EOC : Std_logic;
            Signal ADC_EOC : Std_logic;
            Signal ADC_SOC : Std_logic;
            Signal ADC_Raise : Std_logic;
            Signal ADC_CLK_Count : Std_logic_vector(2 downto 0);
            Signal ADC_Sampled_data : Std_logic_vector(7 downto 0);
            Signal ADC_Sampled_data_reg : Std_logic_vector(7 downto 0);
            Signal ADC_Sample_clk : Std_logic;
            constant OE1_Val : STD_LOGIC := '1';
            constant OE2_Val : STD_LOGIC := '1';
            Signal Seven_seg_Check : std_logic_vector(7 downto 0);
            -- The below declaration is for the state machine which fetches the data from the ADC
            type ADC_State is (Reset_ADC, State_ALE, State_SOC, State_EOC_Check, State_EOC, State_QEl, State_OE2);
            Signal Next_state_ADC : ADC_State;

            -- The below declaration is for the signals which are used internally in the design
            Signal ADC_OE : Std_logic;
            Signal ADC_ALE : Std_logic;
            Signal ADC_SOC : Std_logic;
            Signal ADC_EOC : Std_logic;
            Signal ADC_SOC : Std_logic;
            Signal ADC_Raise : Std_logic;
            Signal ADC_CLK_Count : Std_logic_vector(2 downto 0);
            Signal ADC_Sampled_data : Std_logic_vector(7 downto 0);
            Signal ADC_Sampled_data_reg : Std_logic_vector(7 downto 0);
            Signal ADC_Sample_clk : Std_logic;
            constant OE1_Val : STD_LOGIC := '1';
            constant OE2_Val : STD_LOGIC := '1';
            Signal Seven_seg_Check : std_logic_vector(7 downto 0);

            -- The below declaration is for the component for seven segment display on the board. The DAC data which will be sent to DAC after PID equation calculation will be displayed on the seven segment display.
            Component Seven_Segment is
                port ( Reset_in : in std_logic; -- Reset input to Line control block
                        ADC_CLK_in : in std_logic; -- 1.25MHz Clock to the module
                        ADC_data_in : in std_logic_vector(7 downto 0); -- ADC data input
                        OE1_out : out std_logic; -- Output Enable for Display output Latch Buffer
                        OE2_out : out std_logic; -- Output Enable for Display output Latch Buffer
                        Display_Data_out : out std_logic_vector(7 downto 0)); -- Display data for Seven Segment Display
            end Component Seven_Segment;

            -- Beginning of the Architecture for this module
            begin
                SOC_out <= ADC_SOC;
                OE2_out <= ADC_SOC;
                OE3_Disp <= OE3.Val; -- Output enable for Seven segment Display 3
0E4_Disp <= 0E4_Val; -- Output enable for Seven segment Display 4

-- The below statements are port mapping for seven segment display which displays the DAC Data

Display : Seven_Segment port map ( Reset_in, ADC_Sample_clk, ADC_data_in, OEl_out, OE2_out, Display_Data_out => Seven__seg_out);

--- Counter for Clock Division----------------------------------

-- The below statement is the counter for clock division. Input clock is divided by 2, 4 and 8 and can be used in the other modules.

Process(ADC_Clk_in, Reset_in)
begin
if Reset_in = '0' then
  Clk_Count <= "111";
elsif(ADC_Clk_in'event and ADC_Clk_in = '1') then
  Clk_Count <= Clk_Count + "001";
end if;
end process;

--- Address out for ADC channel selection----------------------------------

-- The below statement is for assigning address to ADC for channel selection out of 8 input channels. This is a register which registers the input switch data for ADC channel selection and assigns it to the output for ADC channel selection.

Process(ADC_Sample_clk, Reset_in)
begin
if Reset_in = '0' then
  Address_out <= "000";
elsif (ADC_Sample_clk'event and ADC_Sample_clk = '1') then
  Address_out <= ADC_Addrs_in;
end if;
end process;

-- ADC Capture State Machine-----------------------------

-- This below state machine generates and sends out the ADC control signals to capture the ADC Data

Process(ADC_Sample_clk, Reset_in)
begin
if(Reset_in = '0') then
  ADC_SOC <= '0';
  ADC_Data_Ready <= '0';
  ADC_OE <= '0';
  ADC_Sampled_data <= (others => '0');
  Next_state_ADC <= Reset_ADC;
elsif (ADC_Sample_clk'event and ADC_Sample_clk = '1') then
  case Next_state_ADC IS
  WHEN Reset_ADC =>
    
  WHEN State_SOC =>
    
  WHEN State_ALX =>
    
  WHEN State_SOC =>
    
  WHEN State_ALX =>
    
end case;
end if;
end process;

-- The below state is a State_SOC where SOC and OE for ADC will be asserted

-- Below state is a State_ALX state where it is a Dummy state for Delay

-- Below state is a State_SOC where SOC and OE for ADC will be asserted

-- Below state is a Reset state where all the control signals are initialized

--- Divide by 8 clock for ADC module -------

--- ADC Interface.vhd
Below state is a State_EOC where we wait in this state until ADC asserts EOC

WHEN State_EOC =>
  ADC_SOC <= '0';
  ADC_Data_Ready <= '0';
  ADC_OE <= '0';
  ADC_Sampled_data <= (others => '0');
  if ADC_EOC = '0' then
    Next_state_ADC <= State_EOC;
  else
    Next_state_ADC <= State_EOC_Check;
  end if;

Below state is a State_EOF where we assert OE to ADC for data latching

WHEN State_EOF =>
  ADC_SOC <= '0';
  ADC_Data_Ready <= '0';
  ADC_OE <= '0';
  ADC_Sampled_data <= ADC_data_in;
  Next_state_ADC <= State_EOF;

Below state is a State_OE2 where we assert ADC Data_ready to latch the ADC data

WHEN State_OE2 =>
  ADC_SOC <= '0';
  ADC_Data_Ready <= '1';
  ADC_ALE <= '0';
  ADC_OE <= '1';
  ADC_Sampled_data <= ADC_data_in;
  Next_state_ADC <= State_EOF;

Below state is a others state where it is a default state when no condition is satisfied

WHEN others =>
  ADC_SOC <= '0';
  ADC_Data_Ready <= '0';
  ADC_OE <= '0';
  ADC_Sampled_data <= (others => '0');
  Next_state_ADC <= Reset_ADC;
end if;

End case;
End process;

End of State machine

End ADC_Interface_arch;

End of Module
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ADC_Data_rs is
  port ( 
    Reset_in : in std_logic; -- Reset input to Line control block
    ADC_Clk_in : in std_logic; -- 10MHz Clock to the module
    ADC_Data_rdy_in : in std_logic; -- Average Data Input from ADC Data Read Module
    ADC_Data_capture : in std_logic; -- Switch to control the Data capture
    ADC_data_in : in std_logic_vector(7 downto 0); --ADC data Input
    ADC_AVG_Data_out : out std_logic_vector(7 downto 0); -- Avg data out of 200 samples
    ADC_AVG_Data_Rdy_out : out std_logic); -- ADC Average data ready
end ADC_Data_rs;

Architecture ADC_Data_rs_arch of ADC_Data_rs is
  State Machine Declaration
  
  The below state machine declaration is for ADC data latching from the ADC interface module

  type ADC_RD_State is (ResetJRB, State_ADC"Data_chk, State_ADC_Data_Add, State_ADC_Data_Ltch, 
                        State_Count CHK, State_ADC_AVG_dataJRdy, State_ADC_AVG_data_Rdy2, State_ADC_Nxt_data_wait); 

  Signal Next_state_ADC_RD : ADC_RD_State;
  
  Signal ADC_Read_Data : Std_logic_vector(7 downto 0);
  Signal ADC_Avg_Data_reg : atd_logic_vector(7 downto 0);
  Signal ADC_Avg_Data : std_logic_vector(7 downto 0);
  Signal Data_Count : std_logic_vector(7 downto 0);
  Signal ADC_AVG_Data_Rdy_latch : Std_logic;
  Signal ADC_AVG_Data_Rdy : Std_logic;

  -- Begining of the Architecture for this module

  begin

    -- Signal assignment

    ADC_AVG_Data_Rdy_out <= ADC_AVG_Data_Rdy;
    ADC_AVG_Data_out <= ADC_AVG_Data_reg;

    -- Register assignment

    Process(Reset_in, ADC_Clk_in)
    begin
      if(Reset_in = '0') then
        ADC_Avg_Data_reg <= (others => '0');
      elsif (ADC_Clk_in'event and ADC_Clk_in = '1') then
        if(ADC_AVG_Data_Rdy_latch = '1') then
          ADC_Avg_Data_reg <= ADC_Avg_Data;
        else
          ADC_Avg_Data_reg <= ADC_Avg_Data_reg;
        end if;
      end if;
    end process;

    -- Register assignment

    Process(Reset_in, ADC_Clk_in)
    begin
      if(Reset_in = '0') then
        ADC_Read_Data <= (others => '0');
      elsif (ADC_Clk_in'event and ADC_Clk_in = '1') then
        if(ADC_Data_rdy_in = '1') then
          ADC_Read_Data <= ADC_data_in;
        end if;
      end if;
    end process;

earchitecture ADC_Data_rd_arch of ADC_Data_rd is
  
  -- The below states are Signal declaration for top modules to assign the inputs and outputs of the module.

  Signal ADC_Read_Data : Std_logic_vector(7 downto 0);
  Signal ADC_Avg_Data_reg : atd_logic_vector(7 downto 0);
  Signal ADC_Avg_Data : std_logic_vector(7 downto 0);
  Signal Data_Count : std_logic_vector(7 downto 0);
  Signal ADC_AVG_Data_Rdy_latch : Std_logic;
  Signal ADC_AVG_Data_Rdy : Std_logic;

  -- Begining of the Architecture for this module

  begin

    -- Signal assignment

    ADC_AVG_Data_Rdy_out <= ADC_AVG_Data_Rdy;
    ADC_AVG_Data_out <= ADC_AVG_Data_reg;

    -- Register assignment

    Process(Reset_in, ADC_Clk_in)
    begin
      if(Reset_in = '0') then
        ADC_Avg_Data_reg <= (others => '0');
      elsif (ADC_Clk_in'event and ADC_Clk_in = '1') then
        if(ADC_AVG_Data_Rdy_latch = '1') then
          ADC_Avg_Data_reg <= ADC_Avg_Data;
        else
          ADC_Avg_Data_reg <= ADC_Avg_Data_reg;
        end if;
      end if;
    end process;

    -- The below register assignment for Data which is going to Motor control module for further processing. This data
    -- is registered based on the ADC_AVG_Data_RdyLatch signal from the state machine.

    Process(Reset_in, ADC_Clk_in)
    begin
      if(Reset_in = '0') then
        ADC_Read_Data <= (others => '0');
      elsif (ADC_Clk_in'event and ADC_Clk_in = '1') then
        if(ADC_Data_rdy_in = '1') then
          ADC_Read_Data <= ADC_data_in;
        end if;
      end if;
    end process;

end architecture ADC_Data_rd_arch;
else
ADC_Read_Data <= ADC_Read_Data;
end if;
end if;
end process;

----------      ADC Read State Machine ———--—-------—---------
— The below state machine is for ADC data latching from the ADC interface module ———--—-------—---------

Process(ADC_Clk_in, Reset_in)
begin
if (Reset_in * '0') then
  Data_Count <= (others => '0');
  ADC_Avg_Data <= (others => '1');
  ADC_AVG_Data_Rdy <= '0';
  ADC_AVG_Data_Rdy_latch <= '0';
  Next_state_ADC_RD <= Reset_RD;
elsif (ADC_Clk_in'event and ADC_Clk_in = '0') then
  Data_Count <= (others => '0');
  ADC_Avg_Data <= (others => '1');
  ADC_AVG_Data_Rdy <= '0';
  ADC_AVG_Data_Rdy_latch <= '0';
  Next_state_ADC_RD <= State_ADC_Data_chk;
end if;

-- Below state is a Reset state where all the control signals are initialized
WHEN Reset_RD =>
  Data_Count <= (others => '0');
  ADC_Avg_Data <= (others => '1');
  ADC_AVG_Data_Rdy <= '0';
  ADC_AVG_Data_Rdy_latch <= '0';
  Next_state_ADC_RD <= State_ADC_Data_chk;

-- Below state is State_ADC_Data_chk state where we are checking for ADC_Data_capture control signal from
input toggle switch and proceeds to next state if it is asserted
WHEN State_ADC_Data_chk =>
  Data_Count <= Data_Count;
  ADC_Avg_Data <= ADC_Avg_Data;
  ADC_AVG_Data_Rdy <= '0';
  ADC_AVG_Data_Rdy_latch <= '0';
  if(ADC_Data_capture = '1') then
    Next_state_ADC_RD <= State_ADC_Data_Latch;
  else
    Next_state_ADC_RD <= State_ADC_Data_chk;
  end if;

-- Below state is State_ADC_Data_Latch state where we are checking for ADC_Data_rdy_in control signal from
ADC interface module and proceeds to next state if it is asserted
WHEN State_ADC_Data_Latch =>
  Data_Count <= Data_Count;
  ADC_Avg_Data <= ADC_Avg_Data;
  ADC_AVG_Data_Rdy <= '0';
  ADC_AVG_Data_Rdy_latch <= '0';
  Next_state_ADC_RD <= State_ADC_Data_Count_chk;

-- Below state is State_ADC_Data_Count_chk state where we are checking the count whether it is 200 and then
proceeding to state State_ADC_AVG_Data_Rdy or stay in the same state until the count is 200
WHEN State_ADC_Data_Count_chk =>
  Data_Count <= Data_Count;
  ADC_Avg_Data <= ADC_Avg_Data;
  ADC_AVG_Data_Rdy <= '0';
  ADC_AVG_Data_Rdy_latch <= '0';
  if Data_Count = "11001000" then
    Next_state_ADC_RD <= State_ADC_AVG_Data_Rdy;
  else
    Next_state_ADC_RD <= State_ADC_Data_Latch;
  end if;

-- Below state is State_ADC_AVG_Data_Rdy state where we are asserting ADC_AVG_Data_Rdy latch signal to latch the
valid 200th data from the ADC
WHEN State_ADC_AVG_Data_Rdy1 =>
  Data_Count <= Data_Count;
  ADC_Avg_Data <= ADC_Avg_Data;
  ADC_AVG_Data_Rdy <= '0';
  ADC_AVG_Data_Rdy_latch <= '0';
  if Data_Count = "11001000" then
    Next_state_ADC_RD <= State_ADC_AVG_Data_Rdy;
  else
    Next_state_ADC_RD <= State_ADC_Data_Latch;
  end if;

-- Below state is State_ADC_AVG_Data_Rdy1 state where we are asserting ADC_AVG_Data_Rdy latch signal to latch the
valid 200th data from the ADC
WHEN state_ADC_AVG_data_Rdy1 ==
  Data_Count <= (others => '0');
  ADC_Avg_Data <= ADC_Avg_Data;
  ADC_AVG_Data_Rdy <= '0';
  ADC_AVG_Data_RdyLatch <= '1';
  Next_state_ADC_RD <= State_ADC_AVG_data_Rdy2;

-- Below state is State_ADC_AVG_data_Rdy2 state where we are asserting ADC_AVG_Data_Rdy signal which will be
-- assigned to Motor control module to latch the valid ADC data from this module

WHEN State_ADC_AVG_data_Rdy2 ==
  Data_Count <= Data_Count;
  ADC_Avg_Data <= (others => '1');
  ADC_AVG_Data_Rdy <= '1';
  ADC_AVG_Data_RdyLatch <= '0';
  Next_state_ADC_RD <= State_ADC_Nxt_data_wait;

-- Below state is State_ADC_Nxt_data_wait is a dummy state to introduce one clock delay and from here it goes to
-- first state State_ADC_Data_chk and the flow continues.

WHEN State_ADC_Nxt_data_wait ==
  Data_Count <= Data_Count;
  ADC_Avg_Data <= ADC_Avg_Data;
  ADC_AVG_Data_Rdy <= '0';
  ADC_AVG_Data_RdyLatch <= '0';
  Next_state_ADC_RD <= State_ADC_Data_chk;

-- Below state is a others state where it is a default state when no condition is satisfied

WHEN others ==
  Data_Count <= (others => '0');
  ADC_Avg_Data <= (others => '1');
  ADC_AVG_Data_Rdy <= '0';
  ADC_AVG_Data_RdyLatch <= '0';
  Next_state_ADC_RD <= Reset_RD;

End if;
End process;

End of State machine

End ADC_Data_rd_arch;

End of Module
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity Motor_Control is
port ( Reset_in: in std_logic; — Reset input to Line control block
ADC_CLK_in: in std_logic; — 10MHz Clock to the module
ADC_Data_rdy: in std_logic; — Average Data Input from ADC Data Read Module
Current_speed_capture: in std_logic; — Current speed capture switch
ADC_Sum_data_in: in std_logic_vector(2 downto 0); — ADC data Input
Speed_Set_switch: in std_logic_vector(7 downto 0); — ADC data Input
LED_Out: out std_logic;
Pid_Data_out: out std_logic_vector(7 downto 0)); -- Pid data to DAC.
end Motor_control;

Architecture Motor_control_arch of Motor_control is

begin
    -- The below state machine Motor_Control_State is to process and assign the current PID and Previous PID values and also the current and previous errors.
    type Motor_Control_State is (Reset_Motor_SM/ State_ADC_Data_Rdy__Nxt, State_ADC_Data_Rdyf State_ADC__Data_Rdy_chk, State_ADC_Data_capture_chk, State__Cal__Delay, State_Assignl, State_Assign2, State_Assign3, State__Assign4);
    Signal Next_state_MC : Motor_Control_State;

    -- The below state machine PID_Assign_State is to assign the current PID value to DAC based on the value of the calculated PID value and also the polarity.
    type PID_Assign_State is (Reset_PID__SM, State_PID_Data_lth_chk( State_Data_chk_Max_Min, State__Data_Min, State_Data_Max, State_ActualJData);
    Signal Next_state_PID : PID_Aaaign_State;

    -- The below state machine SW..Debounce_..State is to overcome the debouncing issue when using the toggle swich for enabling the capture.
    type SW_Debounce_State is (Reset_SDB_SM, State_capture_chk, state_capture_wait, state_capture);
    Signal Next_state_SDB : SW_Debounce_State;

    -- The below state machine Speed_Lock_State is to lock the current speed to set speed when the current speed lies in the range +100 rpm or -100 rpm of set speed. This approximation is because we cannot handle the floating values for the Kp, Ki and Kd values.
    type Speed_Lock_State is (Reset_SL_SM, State_Speed_capture_check, 3tate_.ADC_data_.rdy, state"speed_range_chk, state_nxt_speed_set);
    Signal Next_state_SL : Speed_Lock_State;

    -- The below component declaration is for the Xilinx specific divider module which is generated using the coregen from xilinx.
    component div_gen_vl_0 IS
        port ( elk : IN std_logic;
            aclr : IN std_logic;
            dividend : IN std_logic_VECTOR(31 downto 0);
            divisor : IN std_logic_VECTOR(31 downto 0);
            quotient : OUT std_logic_VECTOR(31 downto 0);
            rfd : OUT std_logic);
    end component div_gen__v 1_0 ;

    -- The below component declaration is for the PID_Eqn_Keonst module which is instantiated in this module and which will compute the value of PID equation Kp*(En - En-i) + Ki*En + Kd*(En - 2*{En-l))+ En-2
    Component PID_Eqn_Kconst is
        port ( Reset_in : in std_logic; — Reset input to Line control block
            ...
Motor_control.vhd

ADC_Clk_in : in std_logic; -- 10MHz Clock to the module
Km_in : in std_logic_vector(15 downto 0);
Kn_1_in : in std_logic_vector(15 downto 0);
Kn_2_in : in std_logic_vector(15 downto 0);
Kn_polatity : in std_logic;
Kn_1_polatity : in std_logic;
Kn_2_polatity : in std_logic;
K_Equation : Out std_logic_vector(31 downto 0);
K_polatity : Out std_logic;

begin

end component PID_Eqn_Konset;

The below statements are for signal declaration for the signals used in the Motor_Control module

--- Signal Declaration ---

Signal Sum_Data_Reg
Signal AVG_Data
Signal Set_Speed_Sp
Signal Cal_Speed
Signal Current_Speed
Signal Present_error
Signal Delay_count
Signal En_polatity
Signal En_1_polatity
Signal En_2_polatity
Signal Previous_Vn_1_polatity
Signal Current_Speed_lock
Signal K_Equation
Signal K_Polarity
Signal Speed_data
Signal Current_PID_Val_Vn
Signal Current_Vn_Polarity
Constant Max_PID_val_to_Dac
Constant Previous_PID_Val_Vn_1
Constant Speed_const
Constant clk_enb
Signal Pid_Data_latch
Signal Pid_Data_lock
Signal Pid_Data
Signal Div_clock_enb
Signal PID_Data_div
Signal remainder
Signal Kfd
Signal ADC_Data_select
Signal R_nreset
Signal Vn_1_Keqn_pol
Signal Set_speed_Range_max
Signal Set_speed_Range_min
Signal SP_for_prev_val
Signal LED_debug
Signal capture_SDB_val
Signal RSB_count

begin

Pid_Data_out <= Pid_Data; --- Calculated PID data to DAC
R_nreset <= not Reset_in; -- Negating the reset to take active high reset for divider module
LED_out <= LED_debug; -- This LED is to show the current speed lock to set speed

--- Signal assignment ---

Vn_1_Keqn_pol <= Previous_Vn_1_Polarity & K_Polarity;
SP_for_prev_val <= "00000000000000" & Set_Speed_Sp;

--- Case statement ---

The below case statement is for assigning the value to set speed Set_Speed_Sp based on the input toggle switches
value Speed ... Set_switch.
process(Reset_in, ADC_Clk_in)
begin
if(Reset_in = '0') then
    Set_Speed_Sp <= "0000100101101000"; -- 1200
elsif(ADC_Clk_in'event and ADC_Clk_in = '1') then
    if capture_SDB_val = '1' then
        case Speed_Set_switch is
        when "000" =>
            Set_Speed_Sp <= "0000100111000100"; -- 2500
        when "001" =>
            Set_Speed_Sp <= "0000101010110000"; -- 1200
        when "010" =>
            Set_Speed_Sp <= "0000010110100000"; -- 600
        when "011" =>
            Set_Speed_Sp <= "0000010010110000"; -- 1400
        when "100" =>
            Set_Speed_Sp <= "0000111110100000"; -- 2000
        when "101" =>
            Set_Speed_Sp <= "0000010010110000"; -- 600
        when "110" =>
            Set_Speed_Sp <= "0000100101100000"; -- 1100
        when "111" =>
            Set_Speed_Sp <= "0000100111000100"; -- 2500
        end case;
    else
        Set_Speed_Sp <= Set_Speed_Sp;
    end if;
end if;
end process;

-- The below below register assignment is for registering the ADC_Data_select based on Pid_Data_latch from the
-- Motor control state machine.
process(Reset_in, ADC_Clk_in)
begin
if(Reset_in = '0') then
    ADC_Data_select <= '0';
elsif(ADC_Clk_in'event and ADC_Clk_in = '1') then
    if(Pid_Data_latch = '1') then
        ADC_Data_select <= '1';
    else
        ADC_Data_select <= ADC_Data_select;
    end if;
end if;
end process;

-- The below below register assignment is for assigning the value for the LED_debug LED to show whether the
-- current speed lock to set speed.
process(Reset_in, ADC_Clk_in)
begin
if(Reset_in = '0') then
    LED_debug <= '0';
elsif(ADC_Clk_in'event and ADC_Clk_in = '1') then
    if(Pid_Data_lock = '1') then
        LED_debug <= '1';
    else
        LED_debug <= LED_debug;
    end if;
end if;
end process;

-- The below below register assignment is for registering the ADC data input from the ADC_Data_rd module based on
-- control input from the ADC_Data_rd module.
process(Reset_in, ADC_Clk_in)
begin
if(Reset_in = '0') then
    Sum_Data_Reg <= (others => '0');
elsif(ADC_Clk_in'event and ADC_Clk_in = '1') then
    if(ADC_Data_rdy = '1') then
        Sum_Data_Reg <= ADC_Data_in;
    else
        Sum_Data_Reg <= Sum_Data_Reg;
    end if;
end if;
end process;
Register assignment

--- The below below register assignment is for registering the set speed or current speed to actual speed of the motor based on the PID_Data_lock from Speed_Lock_State.

```vhdl
process (Reset_in, ADC_Clk_in)
begin
  if (Reset_in = '0') then
    Speed <= (others => '0');
  elsif (ADC_Clk_in' event and ADC_Clk_in = '1') then
    if (PID_Data_lock = '1') then
      Speed <= Set_Speed_Sp;
    else
      Speed <= Cal_Speed;
    end if;
  end if;
end process;
```

Component Instantiation

--- The below component instantiation is for the PID_Konst module which is instantiated in this module and which will compute the value of PID equation \( K_p \times (E_n - E_{n-1}) + K_i \times E_n + K_d \times (E_n - 2 \times E_{n-1}) + E_{n-2} \).

```vhdl
PID_Konst : PID_Konst port map (Reset_in, ADC_Clk_in, En_in, En_1_in, En_2_in, En_polarity, En_1_polarity, En_2_polarity, K_Equation, K_polarity);
```

--- The below component instantiation is for the Xilinx specific divider module which is generated using the coregen.

```vhdl
Divider : div_gen_v1_0 port map (clk, ce, aclr, dividend, divisor, quotient, remainder, rfd, PID_Data_div, remainder, Rfd);
```

Signal assignment

--- The below statement is for locking the current speed to set speed when it lies in the set speed Max and Min range.

```vhdl
Current_Speed_lock <= '1' when (Current_Speed < Set_speed_Range_max and Current_Speed > Set_speed_Range_min) else '0';
```

Motor_Control State Machine

--- The below state machine Motor_Control_State is to process and assign the current PID and Previous PID values and also the current and previous errors.

Process (ADC_Clk_in, Reset_in)
Motor_control.vhd
begin
if(Reaet_in = '0') then
  Previous_PID_Val_Vn_l <= (others => '0');
  Pid_Data_latch <= '0';
  Div_clock_enb <= '0';
  Previous_Error_En_1 <= (others => '0');
  Previous_Error_En_2 <= (others => '0');
  En_2_polarity <= '0';
  En_1_polarity <= '0';
  Previous_Vn_l_polarity <= '0';
  Delay_count <= (others => '0');
  Next_state_MC <= Reset_Motor_SM;
elsif (ADC_Clk_in' event and ADC_Clk_in = '1') then
  case Next_state_MC is
  when Reset_Motor_SM =>
    Previous_PID_Val_Vn_l <= (others => '0');
    Pid_Data_latch <= '0';
    Div_clock_enb <= '0';
    Previous_Error_En_1 <= (others => '0');
    Previous_Error_En_2 <= (others => '0');
    En_2_polarity <= '0';
    En_1_polarity <= '0';
    Previous_Vn_l_polarity <= '0';
    Delay_count <= (others => '0');
    Next_state_MC <= State_ADC_Data_Rdy;
  when State_ADC_Data_Rdy =>
    Previous_PID_Val_Vn_l <= Previous_PID_Val_Vn_1;
    Pid_Data_latch <= '0';
    Div_clock_enb <= '0';
    Delay_count <= Delay_count;
    En_2_polarity <= En_2_polarity;
    En_1_polarity <= En_1_polarity;
    Previous_Vn_l_polarity <= Previous_Vn_l_polarity;
    Previous_Error_En_1 <= Previous_Error_En_1;
    Previous_Error_En_2 <= Previous_Error_En_2;
    Previous_Vn_l_polarity <= Previous_Vn_l_polarity;
    Next_state_MC <= State_ADC_Data_Rdy_check;
  when State_ADC_Data_Rdy_Nxt =>
    Pid_Data_latch <= '0';
    Div_clock_enb <= '0';
    Delay_count <= Delay_count;
    En_2_polarity <= En_2_polarity;
    En_1_polarity <= En_1_polarity;
    Previous_Vn_l_polarity <= Previous_Vn_l_polarity;
    Previous_Error_En_1 <= Previous_Error_En_1;
    Previous_Error_En_2 <= Previous_Error_En_2;
    Previous_Vn_l_polarity <= Previous_Vn_l_polarity;
    Next_state_MC <= State_ADC_Data_Rdy_check;
  when State_ADC_Data_Rdy_check =>
    Previous_PID_Val_Vn_l <= Previous_PID_Val_Vn_1;
    Pid_Data_latch <= '0';
    Div_clock_enb <= '0';
    Delay_count <= Delay_count;
    En_2_polarity <= En_2_polarity;
    En_1_polarity <= En_1_polarity;
    Previous_Vn_l_polarity <= Previous_Vn_l_polarity;
    Previous_Error_En_1 <= Previous_Error_En_1;
    Previous_Error_En_2 <= Previous_Error_En_2;
    Previous_Vn_l_polarity <= Previous_Vn_l_polarity;
    Next_state_MC <= State_ADC_Data_Rdy_check;
  when State_ADC_Data_capture_chk =>
    Previous_PID_Val_Vn_l <= Previous_PID_Val_Vn_1;
    Pid_Data_latch <= '0';
    Div_clock_enb <= '0';
    Delay_count <= Delay_count;
    En_2_polarity <= En_2_polarity;
    En_1_polarity <= En_1_polarity;
    Previous_Vn_l_polarity <= Previous_Vn_l_polarity;
    Previous_Error_En_1 <= Previous_Error_En_1;
    Previous_Error_En_2 <= Previous_Error_En_2;
    Previous_Vn_l_polarity <= Previous_Vn_l_polarity;
    Next_state_MC <= State_ADC_Data_capture_chk;
    next;
  end case;
end if;
end if;
WHEN State_ADC_Data_capture_chk =>
  Previous_PID_Val_Vn_1 <= Previous_PID_Val_Vn_1;
  Pid_Data_latch <= '0';
  Div_clock_enb <= '0';
  Delay_count <= Delay_count;
  En_2_polarity <= En_2_polarity;
  En_1_polarity <= En_1_polarity;
  Previous_Vn_1_polarity <= Previous_Vn_1_polarity;
  Previous_Error_En_1 <= Previous_Error_En_1;
  Previous_Error_En_2 <= Previous_Error_En_2;
  if (Current_speed_capture = '1') then
    Next_state_MC <= State_Cal_Delay;
  else
    Next_state_MC <= State_ADC_Data_Rdy_chk;
  end if;

-- The below state is to introduce some delay for ADC data to get stable after the feed back. ----------------------------------

WHEN State_Cal_Delay =>
  Previous_PID_Val_Vn_1 <= Previous_PID_Val_Vn_1;
  Pid_Data_latch <= '0';
  Div_clock_enb <= '0';
  Delay_count <= Delay_count + '00001';
  En_2_polarity <= En_2_polarity;
  En_1_polarity <= En_1_polarity;
  Previous_Vn_1_polarity <= Previous_Vn_1_polarity;
  Previous_Error_En_1 <= Previous_Error_En_1;
  Previous_Error_En_2 <= Previous_Error_En_2;
  if Delay_count = "11110" then
    Next_state_MC <= State_Assign1;
  else
    Next_state_MC <= State_Cal_Delay;
  end if;

-- The below state is to introduce some delay for ADC data to get stable after the feed back. ----------------------------------

WHEN State_Assign1 =>
  Previous_PID_Val_Vn_1 <= Previous_PID_Val_Vn_1;
  Pid_Data_latch <= '0';
  Div_clock_enb <= '0';
  Delay_count <= 00000;
  En_2_polarity <= En_2_polarity;
  En_1_polarity <= En_1_polarity;
  Previous_Vn_1_polarity <= Previous_Vn_1_polarity;
  Previous_Error_En_1 <= Previous_Error_En_1;
  Previous_Error_En_2 <= Previous_Error_En_2;
  if Delay_count = "00001" then
    Next_state_MC <= State_Assign2;
  else
    Next_state_MC <= State_Assign1;
  end if;

-- The below state is to introduce some delay for doing division of calculated PID value to determine the DAC data

WHEN State_Assign2 =>
  Previous_PID_Val_Vn_1 <= Previous_PID_Val_Vn_1;
  Pid_Data_latch <= '0';
  Div_clock_enb <= '0';
  Delay_count <= Delay_count + '00001';
  En_2_polarity <= En_2_polarity;
  En_1_polarity <= En_1_polarity;
  Previous_Vn_1_polarity <= Previous_Vn_1_polarity;
  Previous_Error_En_1 <= Previous_Error_En_1;
  Previous_Error_En_2 <= Previous_Error_En_2;
  if Delay_count = "01111" then
    Next_state_MC <= State_Assign3;
  else
    Next_state_MC <= State_Assign2;
  end if;

-- In below state Pid_Data_latch signal is asserted to latch the calculated value of PID data for DAC --

WHEN State_Assign3 =>
  Previous_PID_Val_Vn_1 <= Previous_PID_Val_Vn_1;
  Pid_Data_latch <= '1';
  Div_clock_enb <= '0';
  Delay_count <= Delay_count;
  En_2_polarity <= En_2_polarity;
  En_1_polarity <= En_1_polarity;
  Previous_Vn_1_polarity <= Previous_Vn_1_polarity;
  Previous_Error_En_1 <= Previous_Error_En_1;
  Previous_Error_En_2 <= Previous_Error_En_2;
  Next_state_MC <= State_ADC_Data_Rdy_Nxt;

-- In below state Div clock enb signal is asserted to latch the divided value of PID data for DAC --

WHEN State_Assign4 =>
  Previous_PID_Val_Vn_1 <= Previous_PID_Val_Vn_1;
  Pid_Data_latch <= '1';
  Div_clock_enb <= '1';
  Delay_count <= Delay_count;
  En_2_polarity <= En_2_polarity;
  En_1_polarity <= En_1_polarity;
  Previous_Vn_1_polarity <= Previous_Vn_1_polarity;
  Previous_Error_En_1 <= Previous_Error_En_1;
  Previous_Error_En_2 <= Previous_Error_En_2;
--- The below state is default state when none of the conditions are satisfied the control passes to this state ----

WHEN others =>
  Next_state_MC <= (others => '0');
  Pid_Data_latch <= '0';
  Div_clock_enb <= '0';
  Previous_Error_En_1 <= (others => '0');
  Previous_Error_En_2 <= (others => '0');
  En_2_polarity <= '0';
  En_1_polarity <= '0';
  Previous_Vn_1_polarity <= '0';
  Delay_count <= (others => '0');
  Next_state_MC <= Reset_Motor_SM;
End case;
End if;
End process;

--- End of state machine ---

--- The below assignment statement is for calculating the present error and En_polarity will define whether the calculated error is positive or negative. 

process(Rest_in, ADC_Clk_in)
begin
  if Rest_in 'O' then
    Present_error_En <= (others => '0');
    En_polarity <= '0';
  elsif(ADC_Clk_in'event and ADC_Clk_in = '1') then
    if(Speed > Set_Speed_Sp) then
      Present_error_En <= Speed - Set_Speed_Sp;
      En_polarity <= '1';
    else
      Present_error_En <= Set_Speed_Sp - Speed;
      En_polarity <= '0';
    end if;
  end if;
end process;

--- The below assignment statement is for calculating the Current_PID_Val_Vn and Current_Vn_Polarity where the Previous_PID_Val_Vn_1 + K_Equation is nothing but (Vn-1) + (Kp*(En - En-1) + Ki*En + Kd*(En - 2*(En-1)) + En-2)
-- Current_Vn_Polarity will define whether the calculated value is positive or negative.

process(Rest_in, ADC_Clk_in)
begin
  if Rest_in = '0' then
    Current_PID_Val_Vn <= (others => '0');
    Current_Vn_Polarity <= '0';
  elsif(ADC_Clk_in'event and ADC_Clk_in = '1') then
    if(Div_clock_enb = '1') then
      case Vn_1_Keep_pol is
      when '00' =>
        Current_PID_Val_Vn <= Previous_PID_Val_Vn_1 + K_Equation;
        Current_Vn_Polarity <= '0';
      when '01' =>
        if(K_Equation > Previous_PID_Val_Vn_1) then
          Current_PID_Val_Vn <= Previous_PID_Val_Vn_1 + K_Equation;
          Current_Vn_Polarity <= '1';
        else
          Current_PID_Val_Vn <= Previous_PID_Val_Vn_1 - K_Equation;
          Current_Vn_Polarity <= '0';
        end if;
      when '10' =>
        if(Present_PID_Val_Vn_1 > K_Equation) then
          Current_PID_Val_Vn <= Present_PID_Val_Vn_1 - K_Equation;
          Current_Vn_Polarity <= '0';
        else
          Current_PID_Val_Vn <= Present_PID_Val_Vn_1 + K_Equation;
          Current_Vn_Polarity <= '0';
        end if;
      when '11' =>
        Current_PID_Val_Vn <= Previous_PID_Val_Vn_1 + K_Equation;
        Current_Vn_Polarity <= '0';
      when others =>
        Current_PID_Val_Vn <= (others => '0');
        Current_Vn_Polarity <= '0';
      end case;
    else
      Current_PID_Val_Vn <= Current_PID_Val_Vn;
      Current_Vn_Polarity <= Current_Vn_Polarity;
    end if;
  end if;
end process;
Process(ADC_CLK_in, Reset_in)
begin
if(Reset_in = '0') then
    Pid_Data <= (others => '0');
    Next_state_PID <= Reset_PID_SM;
elsif (ADC_CLK_in'event and ADC_CLK_in = '0') then
    case Next_state_PID IS
        WHEN Reset_PID_SM =>
            Pid_Data <= (others => '0');
            Next_state_PID <= State_PID_Data_lth_chk;
        WHEN State_PID_Data_lth_chk =>
            if(Pid_Data_latch = '1') then
                Next_state_PID <= State_Data_chk_Max_Min;
            else
                Next_state_PID <= State_PID_Data_lth_chk;
            end if;
        WHEN State_Data_chk_Max_Min =>
            if(Current_speed_capture = '1') then
                Next_state_PID <= State_ADC_data_rdy;
            else
                Next_state_PID <= State_SlIDE;  
            end if;
        WHEN State_ADC_data_rdy =>
            if(ADC_Data_select = '1') then
                Next_state_PID <= State_SlIDE;  
            else
                Next_state_PID <= State_SlIDE;  
            end if;
        WHEN state_speed_range_chk =>
            if(Current_SlIDE_select = '0') then
                Next_state_PID <= State_ADC_data_rdy;
            end if;
    end case;
end if;
end process;

Process(ADC_CLK_in, Reset_in)
begin
if(Reset_in = '0') then
    Pid_Data_lock <= '0';
    Next_state_SL <= Reset_SL_SM;
elsif (ADC_CLK_in'event and ADC_CLK_in = '0') then
    case Next_state_SL IS
        WHEN Reset_SL_SM =>
            Pid_Data_lock <= '0';
        WHEN State_Speed_capture_check =>
            if(Current_speed_capture = '1') then
                Next_state_SL <= state_ADC_data_rdy;
            else
                Next_state_SL <= State_Speed_capture_check;
            end if;
        WHEN state_ADC_data_rdy =>
            if(ADC_Data_select = '1') then
                Next_state_SL <= state_speed_range_chk;
            else
                Next_state_SL <= state_ADC_data_rdy;
            end if;
        WHEN state_speed_range_chk =>
            if(Current_speed_capture = '0') then
                Next_state_SL <= state_speed_range_chk;
            end if;
    end case;
end if;
end process;
else
    Next_state_SL <= state_next_speed_set;
end if;
WHEN others =>
    Pid_Data_lock <= '0';
    Next_state_SL <= Reset_SL_SM;
end case;
end if;
end process;

-- The below state machine SW_Debounce_State is to overcome the debouncing issue when using the toggle switch for
-- enabling the capture.

process (ADC_Clk_in, Reset_in)
begin
    if(Reset_in = '0') then
        capture_SDB_val <= '0';
        SDB_count <= (others => '0');
        Next_state_SDB <= Reset_SDB_SM;
    elsif (ADC_Clk_in'event and ADC_Clk_in = '0') then
        case Next_state_SDB is
            WHEN Reset_SDB_SM =>
                capture_SDB_val <= '0';
                SDB_count <= (others => '0');
                Next_state_SDB <= State_capture_chk;
            WHEN State_capture_chk =>
                capture_SDB_val <= '0';
                SDB_count <= (others => '0');
                if(Current_speed_capture = '1') then
                    Next_state_SDB <= state_capture_wait;
                else
                    Next_state_SDB <= State_capture_chk;
                end if;
            WHEN state_capture_wait =>
                capture_SDB_val <= '0';
                SDB_count <= SDB_count + x"0001";
                if(SDB_count = x"6000") then
                    Next_state_SDB <= state_capture;
                else
                    Next_state_SDB <= state_capture_wait;
                end if;
            WHEN state_capture =>
                capture_SDB_val <= '1';
                SDB_count <= (others => '0');
                if(Current_speed_capture = '0') then
                    Next_state_SDB <= State_capture_chk;
                else
                    Next_state_SDB <= state_capture;
                end if;
            WHEN others =>
                capture_SDB_val <= '0';
                SDB_count <= (others => '0');
                Next_state_SDB <= Reset_SDB_SM;
        end case;
    end if;
end process;

end Motor_control_arch;
This Module is implementing the part of PID equation which is $K_p(En - En-1) + Ki En + Kd(En - 2*(En-1)) + En-2$

This implementation is done in stages

```vhdl
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity PID_Eqn_Kconst is
port
  +ve {'0'} : in
  Reset_in : in
  ADC_Clk_in : in
  En_in : in
  En_l_in : in
  En_2_in : in
  En_polarity : in
  En_l_polarity : in
  En_2_polarity : in
  K_Equation : Out
  K_polarity : Out
end PID_Eqn_Kconst;

architecture PXDJ5qnJKconst_arch of PID__Eqn_Kconst is

begin

-- The below declaration for constants Kp, Ki and Kd which will be used in PID equation
Constant Kp : Std_logic_vector(15 downto 0) := "0000000000000011";
Constant Ki : Std_logic_vector(15 downto 0) := "0000000000000010";
Constant Kd : Std_logic_vector(9 downto 0) := "00000001";

constant two : Std_logic_vector(1 downto 0) := "10";

-- The below declarations are used in calculating the PID equation
Signal Kp_eqn : Std_logic_vector(31 downto 0);
Signal Ki_eqn : Std_logic_vector(31 downto 0);
Signal Kd_eqn : Std_logic_vector(31 downto 0);
Signal Kd_eqn_inter : Std_logic_vector(23 downto 0);
Signal En_2_inter : Std_logic_vector(31 downto 0);
Signal En_2_val : Std_logic_vector(15 downto 0);
Signal En_val : Std_logic_vector(22 downto 0);
Signal Rd_eqn_inter : Std_logic_vector(31 downto 0);
Signal Rd_eqn : Std_logic_vector(31 downto 0);
Signal Ki_eqn_inter : Std_logic_vector(31 downto 0);
Signal Ki_eqn : Std_logic_vector(31 downto 0);
Signal Ki_eqn_inter : Std_logic_vector(31 downto 0);
Signal Kd_eqn_inter : Std_logic_vector(23 downto 0);

begin

Rd_18 <= two * En_l_in;

end PID_Eqn_Kconst_arch;
```

Architecture PID_Eqn_Kconst_arch of PID_Eqn_Kconst is

begin

-- The below declarations are for constants Kp, Ki and Kd which will be used in PID equation
Constant Kp : Std_logic_vector(15 downto 0) := "0000000000000011";
Constant Ki : Std_logic_vector(15 downto 0) := "0000000000000010";
Constant Kd : Std_logic_vector(9 downto 0) := "00000001";

constant two : Std_logic_vector(1 downto 0) := "10";

-- The below declarations are used in calculating the PID equation
Signal Kp_eqn : Std_logic_vector(31 downto 0);
Signal Ki_eqn : Std_logic_vector(31 downto 0);
Signal Kd_eqn : Std_logic_vector(31 downto 0);
Signal Kd_eqn_inter : Std_logic_vector(23 downto 0);
Signal En_2_inter : Std_logic_vector(31 downto 0);
Signal En_2_val : Std_logic_vector(15 downto 0);
Signal En_val : Std_logic_vector(22 downto 0);
Signal Rd_eqn_inter : Std_logic_vector(31 downto 0);
Signal Rd_eqn : Std_logic_vector(31 downto 0);
Signal Ki_eqn_inter : Std_logic_vector(31 downto 0);
Signal Ki_eqn : Std_logic_vector(31 downto 0);
Signal Ki_eqn_inter : Std_logic_vector(31 downto 0);
Signal Kd_eqn_inter : Std_logic_vector(23 downto 0);

begin

Rd_18 <= two * En_l_in;

end PID_Eqn_Kconst_arch;
```
The below equations denote the calculated value for \( K_p \), \( K_i \) and \( K_d \) and their intermediate value with their respective equations are positive or negative.

\[
\begin{align*}
K_p\_eqn\_pol & \left< \text{En}_\text{polarity} \& \text{En}_\text{l}_\text{polarity} \right> \text{— This is for } K_p \*(\text{En - En}_l) \\
K_i\_eqn\_pol & \left< \text{En}_\text{polarity} \right> \text{— This is for } K_i \* \text{En} \\
K_d\_eqn\_pol & \left< \text{En}_\text{polarity} \& \text{En}_\text{l}_\text{polarity} \right> \text{— This is for } K_d \*(\text{En - 2*(En}_l) + \text{En}_2 \right) \\
K_d\_\text{En}_2\_eqn\_pol & \left< K_d\_\text{pol} \& \text{En}_\text{2}_\text{pol} \right> \text{— This is for } K_d \* \left( \text{En - 2*(En}_l) \right) \left. \& \text{En}_2 \right) \\
K_p\_K_i\_eqn\_pol & \left< K_p\_\text{pol} \& \text{K_i}_\text{pol} \right> \text{— This is for } K_p \*(\text{En - En}_l) + K_i \* \text{En} \\
K_p\_K_i\_K_d\_eqn\_pol & \left< K_p\_K_i\_\text{pol} \& K_d\_\text{En}_2\_\text{pol} \right> \text{— This is for } K_p \*(\text{En - En}_l) + K_i \* \text{En} + K_d \* \left( \text{En - 2*(En}_l) \right) + \text{En}_2 \\
K_p\_\text{Equation} & \left< K_p\_K_i\_K_d\_\text{pol} \right> \text{— This is assignment for computing } K_p \*(\text{En - En}_l) + K_i \* \text{En} + K_d \* \left( \text{En - 2*(En}_l) \right) + \text{En}_2 \\
K_p\_\text{polarity} & \left< K_p\_K_i\_K_d\_\text{pol} \right> \text{— This assignment will define whether } K_p\_\text{Equation} \text{ value is positive or negative}
\end{align*}
\]

The below case statement will compute the value for \( (\text{En - 2*(En}_l) \) in \( K_d \* (\text{En - 2*(En}_l) \) and \( K_d\_\text{pol} \) will determine whether the computed value is positive (the value will be '0') or negative (the value will be '1').

Here \( K_d\_\text{eqn}_l \) will be checked which is concatenated value of \( \text{En} \) (whether it is positive or negative) and \( \text{En}_2 \) (whether it is positive or negative).

\[\text{Kd}\_\text{eqn}_l \] is assignment of \( K_d \* (\text{En - 2*(En}_l) \) in PID Equation

\[\text{Kd}\_\text{pol} \] is assignment will determine Whether \( K_d \* (\text{En - 2*(En}_l) \) is positive or negative.

The below case statement will compute the value for \( (\text{En - En}_l) \) in \( K_p \* (\text{En - En}_l) \) and \( K_p\_\text{pol} \) will determine whether the computed value is positive (the value will be '0') or negative (the value will be '1').

Here \( K_p\_\text{eqn}_l \) will be checked which is concatenated value of \( \text{En} \) (whether it is positive or negative) and \( \text{En}_2 \) (whether it is positive or negative).

\[\text{Kp}\_\text{eqn}_l \] is assignment of \( K_p \* (\text{En - En}_l) \) in PID Equation

\[\text{Kp}\_\text{pol} \] is assignment will determine Whether \( K_p \* (\text{En - En}_l) \) is positive or negative.
process(Reset_in,ADC_Clk_in) begin
if Reset_in = '0' then
Kp_sm_val <= (others => '0');
Kp_pol <= '0';
elsif (ADC_Clk_in'event and ADC_Clk_in = '1') then
  case Kp_sm_val_pol is
    when "00" =>
      Kp_sm_val <= En_in + En_1_in;
      Kp_pol <= '0';
    when "01" =>
      Kp_sm_val <= En_in + En_1_in;
      Kp_pol <= '1';
    when "10" =>
      if (En_in > En_1_in) then
        Kp_sm_val <= En_in - En_1_in;
        Kp_pol <= '1';
      else
        Kp_sm_val <= En_1_in - En_in;
        Kp_pol <= '0';
      end if;
    when others =>
      Kp_sm_val <= (others => '0');
      Kp_pol <= '0';
  end case;
end if;
end process;

-- The below case statement will compute the value for $K_p(En - En-1)$ and $K_d(En - 2*(En-1))$ and $K_i*En$ and $K_d(En - 2*(En-1))$ will determine
-- whether the computed value is positive (the value will be '0') or negative (the value will be '1').
-- Bare $K_d(En - 2*(En-1))$ will be checked which is concatenated value of $K_d$ equation $K_d(En - 2*(En-1))$ (whether it is
-- positive or negative) and $En-2$ (whether it is positive or negative). Here we have considered '0' for positive
-- and '1' for negative.

-- The below case statement will compute the value for $K_d(En - 2*(En-1)) + K_i*En$ and $K_p(En - En-1)$ and $K_d(En - 2*(En-1))$ will determine
-- whether the computed value is positive (the value will be '0') or negative (the value will be '1').
-- Bare $K_d(En - 2*(En-1))$ will be checked which is concatenated value of $K_d$ equation $K_d(En - 2*(En-1))$ (whether it is
-- positive or negative) and $En-2$ (whether it is positive or negative). Here we have considered '0' for positive
-- and '1' for negative.
--- ------------------------------—- Case Assignment---------------—*-------------------------*-----------------—
— The below case statement will compute the value for $K_p(En - En-1) + K_i En + K_d(En - 2*(En-1)) + En-2$ and
— $K_p K_i K_d$ will determine whether the computed value is positive (the value will be '0') or negative (the value
— will be '1'). Here $K_p K_i K_d$ will be checked which is concatenated value of $K_p K_i$ equation
— $K_p(En - En-1) + K_i En$ (whether it is positive or negative) and $K_d(En - 2*(En-1)) + En-2$ (whether it is positive
— or negative). Here we have considered '0' for positive and '1' for negative.

process (Reset_in, ADC_Clk_in)
begin
if Reset_in = '0' then
  $K_p K_i K_d = (others => '0');$
elsif (ADC_Clk_in' event and ADC_Clk_in = '1') then
  case $K_p K_i K_d$ is
  when "00" =>
    $K_p K_i K_d$ = $K_p K_i K_d = '0';$
  when "01" =>
    if ($K_p K_i K_d > $K_p K_i K_d$) then
      $K_p K_i K_d$ = $K_p K_i K_d$ - $K_p K_i K_d$;
    else
      $K_p K_i K_d$ = $K_p K_i K_d$ - $K_p K_i K_d$;
    end if;
  when "10" =>
    if ($K_p K_i K_d > $K_p K_i K_d$) then
      $K_p K_i K_d$ = $K_p K_i K_d$ - $K_p K_i K_d$;
    else
      $K_p K_i K_d$ = $K_p K_i K_d$ - $K_p K_i K_d$;
    end if;
  when "11" =>
    if ($K_p K_i K_d > $K_p K_i K_d$) then
      $K_p K_i K_d$ = $K_p K_i K_d$ - $K_p K_i K_d$;
    else
      $K_p K_i K_d$ = $K_p K_i K_d$ - $K_p K_i K_d$;
    end if;
  when others =>
    $K_p K_i K_d = (others => '0');$
    $K_p K_i K_d = '0';$
  end case;
end if;
end process;
End PID_Eqn_Kconst_arch;
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.std_logic_arith.all;  
use ieee.std_logic_unsigned.all;  

entity Seven_Segment is  
port (  
  Reset_in : in std_logic; -- Reset input to Line control block  
  ADC_Clk_in : in std_logic; -- 5MHz Clock to the module  
  ADC_data_in : in std_logic_vector(7 downto 0); -- ADC data input  
  OE1_out : out std_logic; -- Output Enable for Display output Latch Buffer  
  OE2_out : out std_logic; -- Output Enable for Display output Latch Buffer  
  Display_Data_out : out std_logic_vector(7 downto 0); -- Display data for Seven Segment Display  
);  
end Seven_Segment;  

Architecture Seven_Segment_arch of Seven_Segment is  
  Signal Disp_data_LSB : Std_logic_vector(3 downto 0);  
  Signal Disp_data_MSB : Std_logic_vector(3 downto 0);  
  Signal Disp_dataout_LSB : Std_logic_vector(7 downto 0);  
  Signal Disp_dataout_MSB : Std_logic_vector(7 downto 0);  
  type Disp_State is (Reset_Display, State_MSB, State_MSB_Latch, State_LSB, State_LSB_Latch, State_LSB_Latched);  
  Signal Next_state_Disp : Disp_State;  
  Begin  
  -- Below statements are signal assignment for Seven segment display which will be DAC data in the main module  
  Disp_data_MSB <= ADC_data_in(7 downto 4);  
  Disp_data_LSB <= ADC_data_in(3 downto 0);  
  -- Below case statements for data assignment for seven segment which represents LSB bits 3:0 of DAC Data  
  Process(Reset_in, ADC_Clk_in)  
  Begin  
    if Reset_in = '0' then  
      Disp_dataout_LSB <= "00000000";  
    elsif(ADC_Clk_in'event and ADC_Clk_in = '1') then  
      case Disp_data_LSB is  
        when "0000" =>  
          Disp_dataout_LSB <= "11000000";  
        when "0001" =>  
          Disp_dataout_LSB <= "11111001";  
        when "0010" =>  
          Disp_dataout_LSB <= "10100100";  
        when "0011" =>  
          Disp_dataout_LSB <= "10110000";  
        when "0100" =>  
          Disp_dataout_LSB <= "10011001";  
        when "0101" =>  
          Disp_dataout_LSB <= "10010010";  
        when "0110" =>  
          Disp_dataout_LSB <= "10000011";  
        when "0111" =>  
          Disp_dataout_LSB <= "11000110";  
        when "1000" =>  
          Disp_dataout_LSB <= "10000010";  
        when "1001" =>  
          Disp_dataout_LSB <= "11111000";  
        when "1010" =>  
          Disp_dataout_LSB <= "10000000";  
        when "1011" =>  
          Disp_dataout_LSB <= "10001000";  
        when "1100" =>  
          Disp_dataout_LSB <= "10000000";  
        when "1101" =>  
          Disp_dataout_LSB <= "10000000";  
        when "1110" =>  
          Disp_dataout_LSB <= "10000110";  
        when "1111" =>  
          Disp_dataout_LSB <= "10001110";  
        others =>  
          Disp_dataout_LSB <= "11111111";  
      end case;  
    end if;  
  end Process;  
end Seven_Segment_arch;
Process(Reset_in, ADC_Clk_in)
Begin
if Reset_in = '0' then
   Gen_display_out <= Gen_display_out
   Next_stateDisplay <= Next_stateDisplay
elseif(ADC_Clk_in' event and ADC_Clk_in = '1') then
   case Next_stateDisplay is
      WHEN Reset_Display =>
        Gen_display_out <= '1';
        OE1_out <= '1';
        OE2_out <= '1';
        Display_Data_out <= (others => '1');
        Next_stateDisp <= Reset_Display;
      WHEN State_MSB =>
        Gen_display_out <= '0';
        OE1_out <= '1';
        OE2_out <= '1';
        Display_Data_out <= Disp_dataout_MSB;
        Next_stateDisp <= State_MSB_Latch;
      WHEN State_MSB_Latch =>
        Gen_display_out <= '1';
        OE1_out <= '1';
        OE2_out <= '1';
        Display_Data_out <= Disp_dataout_MSB;
        Next_stateDisp <= State_MSB_Latched;
      WHEN State_MSB_Latched =>
        Gen_display_out <= '1';
        OE1_out <= '1';
        OE2_out <= '1';
        Display_Data_out <= Disp_dataout_LSB;
        Next_stateDisp <= State_LSB;
      WHEN State_LSB =>
        Gen_display_out <= '0';
        OE1_out <= '1';
        OE2_out <= '0';
        Display_Data_out <= Disp_dataout_LSB;
        Next_stateDisp <= State_LSB_Latch;
end case;
end if;
end process;

-- Below case statement is for data assignment for seven segment which represents MSB bits 7:4 of DAC Data
---

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

entity seven_segment is
  Port ( display_data_in : in  std_logic_vector (7 downto 0);
         state_in : in  std_logic;
         reset_out : out  std_logic
     );
end seven_segment;

architecture Seven_Segment_arch of seven_segment is
  type next_state_type is (State_LSB_Latched, State_MSB);
  signal next_state : next_state_type;
  signal state : next_state_type := State_LSB_Latched;
  signal display_data_out : std_logic_vector (7 downto 0);
  constant disp_dataout_LSB : std_logic_vector (7 downto 0) :=
    "1 1 1 1 1 1 1";
  constant disp_dataout_MSB : std_logic_vector (7 downto 0) :=
    "1 1 1 1 1 1 1";
begin
  display_data_out <= disp_dataout_LSB when state = State_LSB_Latched else
                     disp_dataout_MSB;
  next_state <= State_MSB when state = State_LSB_Latched else
                State_LSB_Latched;
  state <= next_state;
  reset_out <= '1' when state = State_MSB else
              '0';
end Seven_Segment_arch;
--- End of module
```
--- This file is owned and controlled by Xilinx and must be used
--- solely for design, simulation, implementation and creation of
--- design files limited to Xilinx devices or technologies. Use
--- with non-Xilinx devices or technologies is expressly prohibited
--- and immediately terminates your license.
---
--- XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
--- SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR
--- XILINX DEVICES. BY PROVIDING THIS DESIGN, CODE, OR INFORMATION
--- AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION
--- OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS
--- IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT.
--- AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
--- FOR YOUR IMPLEMENTATION. XILINX EXPRESSLY DISCLAIMS ANY
--- WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
--- IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
--- REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
--- INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
--- FOR A PARTICULAR PURPOSE.
---
--- Xilinx products are not intended for use in life support
--- appliances, devices, or systems. Use in such applications are
--- expressly prohibited.
---
--- (c) Copyright 1995-2007 Xilinx, Inc.
--- All rights reserved.
---
--- -- You must compile the wrapper file div_gen_vl_0.vhd when simulating
--- -- the core, div_gen_v1.0. When compiling the wrapper file, be sure to
--- -- reference the XilinxCoreLib VHDL simulation library. For detailed
--- -- instructions, please refer to the "CORE Generator Help".
--- -- The synthesis directives "translate_off/translate_on" specified
--- -- below are supported by Xilinx, Mentor Graphics and Synplicity
--- -- synthesis tools. Ensure they are correct for your synthesis tool(s).
---
--- LIBRARY ieee;
--- USE ieee.std_logic_1164.ALL;
--- synthesis translate_off
--- Library XilinxCorelib;
--- synthesis translate_on
--- ENTITY div_gen_vl_0 IS
--- port (
--- clk : IN std_logic;
--- ce : IN std_logic;
--- aclr : IN std_logic;
--- dividend : IN std_logic_VECTOR(31 downto 0);
--- divisor : IN std_logic_VECTOR(31 downto 0);
--- quotient : OUT std_logic_VECTOR(31 downto 0);
--- remainder : OUT std_logic_VECTOR(31 downto 0);
--- rfd : OUT std_logic);
--- END div_gen_vl_0;
---
--- ARCHITECTURE div_gen_vl_0_a OF div_gen_vl_0 IS
--- component wrapped_div_gen_vl_0
--- port (
--- clk : IN std_logic;
--- ce : IN std_logic;
--- aclr : IN std_logic;
--- dividend : IN std_logic_VECTOR(31 downto 0);
--- divisor : IN std_logic_VECTOR(31 downto 0);
--- quotient : OUT std_logic_VECTOR(31 downto 0);
--- remainder : OUT std_logic_VECTOR(31 downto 0);
--- rfd : OUT std_logic);
--- end component;
---
--- Configuration specification
--- for all : wrapped_div_gen_vl_0 use entity XilinxCorelib.div_gen_vl_0(behavioral)
--- generic map(
--- divclk_sel => 1,
--- exponent_width => 8,
--- bias => 0,
--- c_has aclr => 0,
--- latency => 1,
--- c_has ce => 1,
--- c_has aclr => 1,
--- c_sync enable => 0,
--- fractional_width => 32,
--- mantissa_width => 8,
--- signed_b => 0,
--- fractional_b => 0,
--- algorithm type => 1,
--- divisor width => 32,
--- dividend width => 32);
--- synthesis translate_on
--- BEGIN
--- -- synthesis translate off
--- U0 : wrapped_div_gen_vl_0
--- port map ( clk => clk,
--- ce => ce,
aclr => aclr,
dividend => dividend,
divisor => divisor,
quotient => quotient,
remainder => remainder,
rfd => rfd);

-- synthesis translate_on
--- This Module is a test bench written for simulating the PID controller top module.

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY PID_Top_tb IS
END PID_Top_tb;

ARCHITECTURE PID_Top__tb_arch OF PID_Top__tb IS

--- The below statements are component declaration for PID controller Top module

COMPONENT PID_controller_top

PORT("Reset_in" : in std_logic; -- Reset input to Line control block
     "ADC_Clk_in" : in std_logic; -- 10MHz Clock to the module
     "ADC_Data_rdy_in" : in std_logic; -- Average Data Input from ADC Data Read Module
     "Current_speed_capture" : in std_logic; -- Current speed capture switch
     "Speed_Set_switch" : in std_logic_vector(2 downto 0); -- This is to set the set speed
     "ADC_data_in" : in std_logic_vector(7 downto 0); -- ADC data input
     "Pid_Data_out" : out std_logic_vector(7 downto 0)); -- Pid data to DAC

END COMPONENT;

--- The below statements are Signal declaration for PID_controller Top module

SIGNAL "Reset_in" : std_logic := '0';
SIGNAL "ADC_Clk_in" : std_logic := '0';
SIGNAL "ADC_Data_rdy_in" : std_logic := '0';
SIGNAL "ADC_data_in" : std_logic_vector(7 downto 0) := (others=> '0');
SIGNAL "Current_speed_capture" : std_logic;
SIGNAL "Speed_Set_switch" : std_logic_vector(2 downto 0);
SIGNAL "Set_speed" : std_logic_vector(2 downto 0) := "100";
SIGNAL "Pid_Data_out" : std_logic_vector(7 downto 0);
BEGIN

"Reset_in" <= '0', '1' after 200 ns; -- This is power on reset
"Current_speed_capture" <= '0', '1' after 100 us; -- This is for current speed toggle switch
"Speed_Set_switch" <= Set_speed; -- This is to set the toggle switch values for set speed

--- The below statements is a process statement which will define the clock of 50 MHz. In actual design the clock speed will be 10 MHz. Here in order to simplify and to see the waveform in a single window we have taken the frequency as 50 MHz.

Process
BEGIN
"ADC_Clk_in" <= transport '1';
wait for 10 ns;
"ADC_Clk_in" <= transport '0';
wait for 10 ns;
end process;

--- The below statement is to assign the valid ADC data to the top module

"ADC_data_in" <= "11111111";

--- The below statement is to give the ADC conversion time delay

Process("Reset_in","ADC_Clk_in")
BEGIN
if("Reset_in" = '0') then
"ADC_Data_count" <= "0000";
elif("ADC_Clk_in"'event and "ADC_Clk_in" = '1') then
"ADC_Data_count" <= "ADC_Data_count" + "0001";
end if;
end process;

"ADC_Data_rdy_in" <= '1' when "ADC_Data_count" = "0011" else '0'; -- Data ready for ADC interface module

--- The below statement is a port mapping for instantiated PID controller top module

uut: PID_controller_top PORT MAP( "Reset_in", "ADC_Clk_in" => "Reset_in",
"ADC_Clk_in" => "ADC_Clk_in",
"ADC_Data_rdy_in" => "ADC_Data_rdy_in",
"Current_speed_capture" => "Current_speed_capture",
"Speed_Set_switch" => "Speed_Set_switch",
"Set_speed" => "Set_speed",
"Pid_Data_out" => "Pid_Data_out","ADC_data_in" => "ADC_data_in"
""