Digital Engineering
Project Task 1

Y3890959
Y3878784

28th February 2023

Task 1: Clock management through enable signals

1 - FSM Description

$$\begin{tikzcd}[column sep=huge]
\text{start} \arrow[r] & \textbf{IDLE} \arrow[r, "\text{en} = 1"] & \textbf{DISP} \arrow[l, bend right, "\begin{array}{c}\text{DISP\_CNT\_OUT} = \text{disp\_delay -1} \\ \text{BSEL\_CNT\_OUT-1} = \text{total\_bytes -1}\end{array}"'] \arrow[r, "\begin{array}{c}\text{DISP\_CNT\_OUT} = \text{disp\_delay -1} \\ \text{BSEL\_CNT\_OUT-1} < \text{total\_bytes -1}\end{array}"] & \textbf{BSEL} \arrow[l, bend left]
\end{tikzcd}$$

Figure 1: FSM state graph

Abbreviations: DISP_CNT - the counter that adds a delay to the LED display output. BSEL_CNT - the counter that increments for BSEL (byte select) so the correct byte can be displayed.

The FSM starts at IDLE state, where ‘EN_SOURCE’ is low, all internal counters (BSEL and DISP) are reset and the LED array is off. When the ‘en’ pushbutton is toggled, the DISP state is entered. Entering this state after IDLE, the counters will be at 0, therefore ‘EN_SOURCE’ will be set high, then low as soon as either of the counters start incrementing. This will trigger ‘DATA_SOURCE’ to compute the next value which will be displayed. The DISP counter will start incrementing to add a delay for the LED display, once the counter reaches a maximum and the BSEL counter hasn’t yet reached a maximum, the BSEL state will be set. Otherwise, the FSM goes back to the IDLE state as all bytes have been displayed with the delay. In the BSEL state, the DISP counter is reset and disabled, and the BSEL counter is enabled for an increment. Going back to DISP state will reset the BSEL counter.

STATE	***EN_SOURCE***	***DISP_CNT_EN***	***DISP_CNT_RST***	***BSEL_CNT_EN***	***BSEL_CNT_RST***	***LED_DISPLAY***
IDLE	0	0	1	0	1	00000000
DISP	1 when DISP_CNT_OUT and DISP_CNT_OUT = 0	1	0	0	0	SOURCE_DATA[7:0] when DISP_CNT_OUT is 0
						SOURCE_DATA[15:8] when DISP_CNT_OUT is 1
	0 otherwise					SOURCE_DATA[23:16] when DISP_CNT_OUT is 2
						SOURCE_DATA[31:24] when DISP_CNT_OUT is 3
BSEL	0	0	1	1	0	00000000

Table 1: FSM table of outputs

2 - VHDL Code for 'STUDENT_AREA'

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.DigEng.all;

-- This is where your work goes. Of course, you will have to put
--   your own comments in, to describe your work.

entity STUDENT_AREA is
    Generic (
        disp_delay : natural := 100000000;
        total_bytes : natural := 4
    );
    Port ( CLK_100MHZ : in  STD_LOGIC;
           -- Debounced button inputs (4=U, 3=C, 2=R, 1=L, 0=D)
           USER_PB : in  STD_LOGIC_VECTOR (4 downto 0);
           -- Board switches (not debounced)
           SWITCHES : in  STD_LOGIC_VECTOR (7 downto 0);
                          -- Board LEDs
           LED_DISPLAY : out  STD_LOGIC_VECTOR (7 downto 0);
                          -- Control signals for the data source
           RST_SOURCE : out  STD_LOGIC;
           EN_SOURCE : out  STD_LOGIC;
           SOURCE_DATA : in  STD_LOGIC_VECTOR (31 downto 0)
                          );
end STUDENT_AREA;

architecture Behavioral of STUDENT_AREA is

signal EN, RST, CLK : STD_LOGIC;
type state_type is (IDLE, DISP, BSEL);  -- The FSM states
signal state, next_state: state_type;   -- The states as signals

-- Internal signals for the counters enable and resets
signal DISP_CNT_OUT : UNSIGNED (log2(disp_delay)-1 downto 0);
signal BSEL_CNT_OUT : UNSIGNED (log2(total_bytes)-1 downto 0);
signal DISP_CNT_EN, BSEL_CNT_EN, DISP_CNT_RST, BSEL_CNT_RST : STD_LOGIC;

begin

RST <= USER_PB(1);
EN <= USER_PB(3);
CLK <= CLK_100MHZ;
RST_SOURCE <= RST;

--  Sets the state as IDLE (reset state) when the reset input is set high.
--  At each clock cycle if the reset isn't high, the state is set to the next
--  state.
state_assignment : process (clk) is
begin
    if rising_edge(clk) then
        if (rst = '1') then
            state <= IDLE;
        else
            state <= next_state;
        end if;
    end if;
end process state_assignment;
```

```vhdl
-- Port map for the display delay counter
DISP_CNT: entity work.parameterizable_counter
GENERIC MAP (LIMIT => disp_delay)
PORT MAP(
        clk => clk,
        rst => DISP_CNT_RST,
        enable => DISP_CNT_EN,
        count_out => DISP_CNT_OUT
);

-- Port map for the byte select counter
BSEL_CNT: entity work.parameterizable_counter
GENERIC MAP (LIMIT => total_bytes)
PORT MAP(
        clk => clk,
        rst => BSEL_CNT_RST,
        enable => BSEL_CNT_EN,
        count_out => BSEL_CNT_OUT
);

fsm_process : process (state, en, DISP_CNT_OUT, BSEL_CNT_OUT)
begin
    case state is
        when IDLE =>
            if en = '1' then
                next_state <= DISP;
            end if;
        when DISP =>
            -- When all of the bytes has been displayed and the delay has been completed,
            -- we can go back to the IDLE state to wait for the next enable signal.
            if (DISP_CNT_OUT = disp_delay-1 and BSEL_CNT_OUT = total_bytes - 1) then
                next_state <= IDLE;
            -- If the display delay has completed but there are still more bytes to be
            -- displayed, the next state can be set to BSEL (byte select), where the
            -- BSEL counter is incremented.
            elsif (DISP_CNT_OUT = disp_delay-1 and BSEL_CNT_OUT < total_bytes - 1) then
                next_state <= BSEL;
            end if;
        when BSEL =>
            -- Once the BSEL counter is incremented once, we can go back to the DISP state
            -- so the next byte can be displayed.
            next_state <= DISP;
    end case;
end process fsm_process;

-- We want the display delay counter to be reset only when the state turns IDLE or BSEL
DISP_CNT_RST <= '1' when state = IDLE or state = BSEL else
                '0';
-- The byte select counter needs to be reset only at IDLE state.
BSEL_CNT_RST <= '1' when state = IDLE else
                '0';
-- The display delay counter needs to be enabled at the DISP state so the logic can hold
-- the LED display so it's visible to the user.
DISP_CNT_EN <= '1' when state = DISP else
               '0';
-- The byte select counter needs to be enabled only at the BSEL state so the counter can
-- increment so that the DISP state logic can use the counter value as an index for which
-- byte to display.
BSEL_CNT_EN <= '1' when state = BSEL else
               '0';
```

```
-- Once the DISP state has been reached with the delay and byte select counter at zero,
-- the data source needs to be enabled so the next value can be computed and displayed.
EN_SOURCE <= '1' when state = DISP and DISP_CNT_OUT = 0 and BSEL_CNT_OUT = 0 else
             '0';

-- The correct byte needs to displayed via the LED display. In the DISP state, the BSEL
-- counter is used as an index and the bytes are displayed.
LED_DISPLAY <= SOURCE_DATA(31 downto 24) when state = DISP and BSEL_CNT_OUT = 0 else
               SOURCE_DATA(23 downto 16) when state = DISP and BSEL_CNT_OUT = 1 else
               SOURCE_DATA(15 downto 8) when state = DISP and BSEL_CNT_OUT = 2 else
               SOURCE_DATA(7 downto 0) when state = DISP and BSEL_CNT_OUT = 3 else
               "00000000";

end Behavioral;
```

3 - VHDL Testbench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


ENTITY TOP_LEVEL_tb IS
END TOP_LEVEL_tb;

-- TEST STRATEGY
--
--  Global Reset & Initialisation:
--      Outside of this vector, all btn inputs will initialise
--      to zeros, followed by a reset button click.
--
--  TEST 1:
--      Cycle through the first 5 values, this is the ensure that
--      all logic within this circuit is functioning properly.
--
--  TEST 2:
--      Following from TEST 1, a reset will be inputted via the
--      pushbutton. This is the verify that the circuit resets
--      properly and goes back to the initial state.
--
--  TEST 3:
--      Cycle through the first 5 values just like in TEST 1, but
--      the enable button will be pressed while the FSM is in the
--      DISP state (after the 2nd input). This test will verify
--      that the enable input will be ignored when the FSM is not
--      in IDLE state.
--
--  TEST 4:
--      Continue from TEST 3, the reset button will be toggled after
--      the 4th input, this will verify that the circuit can be reset
--      while the FSM is in operation.
ARCHITECTURE behavior OF TOP_LEVEL_tb IS
    --Inputs
    signal GCLK : STD_LOGIC;
    signal BTN : STD_LOGIC_VECTOR(4 downto 0);
    signal SW : STD_LOGIC_VECTOR(7 downto 0);
    --Outputs
    signal LED : STD_LOGIC_VECTOR(7 downto 0);
    -- Clock period definitions
    constant GCLK_period : time := 10 ns;
    -- Defining a record of valid outputs to verify the circuit.
    -- The inputs to this circuit (via pushbuttons) will be done
    -- outside of the test vector array.
    type valid_output_array is array (natural range <>) of STD_LOGIC_VECTOR(7 downto 0);
    constant valid_outputs : valid_output_array := (
        -- OUTPUT 1 (4800C00B)
        (X"48"),
        (X"00"),
        (X"C0"),
        (X"0B"),
        -- OUTPUT 2 (0000CA5F)
        (X"00"),
        (X"00"),
        (X"CA"),
        (X"5F"),
        -- OUTPUT 3 (0000570E)
        (X"00"),
        (X"00"),
        (X"57"),
        (X"0E"),
        -- OUTPUT 4 (0380DFAD)
        (X"03"),
        (X"80"),
        (X"DF"),
        (X"AD"),
        -- OUTPUT 5 (0000006C)
        (X"00"),
        (X"00"),
        (X"00"),
        (X"6C")
    );
BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: entity work.TOP_LEVEL
    GENERIC MAP (disp_delay => 30)
    PORT MAP (
        GCLK => GCLK,
        BTN => BTN,
        SW => SW,
        LED => LED
    );

    -- Clock process definitions
    GCLK_process : process
    begin
        GCLK <= '0';
        wait for GCLK_period/2;
        GCLK <= '1';
        wait for GCLK_period/2;
    end process;

    -- Stimulus process
    set_inputs : process
    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;

        -- TEST 1:
        --  Global Reset & Initialisation
        --  Duration: 24 clock cycles
        BTN <= "00000";
        SW <= "00000000";
        wait for GCLK_period*18;
        BTN(1) <= '1';
        wait for GCLK_period*6;
        BTN(1) <= '0';

        wait for GCLK_period*18;

        -- TEST 1
        --  Duration:
        --      5 vectors * (132 clk cycles)
        --      660 clock cycles
        test_1_loop : for i in 0 to 4 loop
            BTN(3) <= '1';
            wait for GCLK_period*6;
            BTN(3) <= '0';
            -- We need to wait for 124 clock cycles as we need to
            -- account for 4x DISP (30 clk cycles) + 3x BSEL (1 clk cycle)
            -- plus 3 clock cycles for the IDLE state
            wait for GCLK_period*126;
        end loop;

        -- TEST 2
        --  Duration: 6 clock period
        BTN(1) <= '1';
        wait for GCLK_period*6;
        BTN(1) <= '0';

        wait for GCLK_period*18;

        -- TEST 3
        --  Duration:
        --      5 vectors * (132 clk cycles)
        --      660 clock cycles
        test_3_loop : for i in 0 to 3 loop
            BTN(3) <= '1';
            wait for GCLK_period*6;
            BTN(3) <= '0';

            -- Toggle enable after the 2nd vector has been passed.
            if (i = 1) then
                wait for GCLK_period*25;
                BTN(3) <= '1';
                wait for GCLK_period*6;
                BTN(3) <= '0';
                wait for GCLK_period*95;
            elsif (i /= 3) then
                -- We need to wait for 124 clock cycles as we need to
                -- account for 4x DISP (30 clk cycles) + 3x BSEL (1 clk cycle)
                -- plus 3 clock cycles for the IDLE state
                wait for GCLK_period*126;
            end if;
        end loop;

        -- TEST 4
        --  Duration: 31 clock period
        wait for GCLK_period*25;
        BTN(1) <= '1';
        wait for GCLK_period*6;
        BTN(1) <= '0';
        wait for GCLK_period*18;

        wait;
    end process;

    check_outputs : process
    begin
        -- Wait for hold reset state.
        wait for 100 ns;
        -- Wait for Global Reset & Initialisation
        wait for GCLK_period*42;

        -- TEST 1
        test_1_test_check_loop : for i in 0 to 19 loop
            -- Wait for the enable signal to toggle
            wait for GCLK_period*6;
            -- Check if the LED output matches the array of known valid output at
            -- the end of every enable btn toggle, as soon as the btn is depressed.
            -- Notify if it's a pass or fail. I'm using severity warning to ensure
            -- that the simulation doesn't halt as soon as a vector fails like it
            -- does for severity failure.
            assert (( LED = valid_outputs(i) ))
            report "TEST VECTOR " & integer'image(i) &
                    " FAIL. Observed LED Output = " & integer'image(to_integer(unsigned(LED))) &
                    " Expected LED Output = " & integer'image(to_integer(unsigned(valid_outputs(i))))
            severity warning;
            report "TEST VECTOR " & integer'image(i) & " PASS."
            severity note;
            wait for GCLK_period*25;
            -- Wait for an additional 8 clk periods at the end of the 4 LED bytes
            -- display to synchronise with the next input.
            if ( (i+1) mod 4 = 0 ) then
                wait for GCLK_period*8;
            end if;
        end loop;

        -- TEST 2
        --  wait for 6 clk periods for the push button to finish toggling
        wait for GCLK_period*6;
        assert (( LED = X"00" ))
        report "TEST 2 " &
                " FAIL. Observed LED Output = " & integer'image(to_integer(unsigned(LED))) &
                " Expected LED Output = 0"
        severity warning;
        report "TEST 2 PASS."
        severity note;

        wait for GCLK_period*18;
        -- TEST 3
        test_3_test_check_loop : for i in 0 to 11 loop
            -- Wait for the enable signal to toggle
            wait for GCLK_period*6;
            -- Check if the LED output matches the array of known valid output at
            -- the end of every enable btn toggle, as soon as the btn is depressed.
            -- Notify if it's a pass or fail. I'm using severity warning to ensure
            -- that the simulation doesn't halt as soon as a vector fails like it
            -- does for severity failure.
            assert (( LED = valid_outputs(i) ))
            report "TEST VECTOR " & integer'image(i) &
                    " FAIL. Observed LED Output = " & integer'image(to_integer(unsigned(LED))) &
                    " Expected LED Output = " & integer'image(to_integer(unsigned(valid_outputs(i))))
            severity warning;
            report "TEST VECTOR " & integer'image(i) & " PASS."
            severity note;
            wait for GCLK_period*25;
            -- Wait for an additional 8 clk periods at the end of the 4 LED bytes
            -- display to synchronise with the next input.
            if ( (i+1) mod 4 = 0 ) then
                wait for GCLK_period*8;
            end if;
        end loop;

        -- TEST 4
        -- wait for 6 clk periods for the push button to finish toggling
        wait for GCLK_period*6;
        -- wait for reset to be toggled
        wait for GCLK_period*31;
        assert (( LED = valid_outputs(0) ))
        report "TEST 4 " &
                " FAIL. Observed LED Output = " & integer'image(to_integer(unsigned(LED))) &
                " Expected LED Output = " & integer'image(to_integer(unsigned(valid_outputs(0))))
        severity warning;
        report "TEST 4 PASS."
        severity note;

        wait;
    end process;
END;
```