Digital Engineering
Project Task 3

Y3890959
Y3878784

14th March 2023

Task 3: Handshaking and source-synchronous communication

1 - FSM Description

The FSM will start at IDLE state as usual. When the WRITE button is toggled, the logic should transition the FSM to WRINST_REQ state, where the FSM will send the write instruction to the SPI to enter write mode, once the SPI_WR_ACK line goes high, we assume the SPI has received the request however, we cannot assume that the handshake is finished, therefore we move to the WRINST_ACK state, where the FSM waits for the SPI's ACK line to go back low in this FSM state, the write request line is pulled low from high. Once WR ACK goes low, the FSM transitions to the next state, WADDR1_REQ. As the address vector is 24-bits wide, there needs to be 3 states to trasmit all the data due to the data line being 1-byte wide, furthermore, as the handshaking is only complete when the WR ACK line goes low, we end up with 3 pairs of states; WADDR*_REQ and WADDR*_ACK. Once all bytes of the address has been sent, the logic waits in the WRHOLD state for the user to set the switches to their desired value and to press the ENTER button. When ENTER is toggled, the FSM increments the INPT_CNT which is an up-down counter, then performs a write request passing the switch value in the data bus, once the SPI acknowledges and the handshake completes, the FSM returns to WRHOLD. From WRHOLD, when the WRITE button i toggled without writing any values, then the FSM goes to IDLE, if values have been written and write mode it exited, then the FSM moves to the RDHOLD state. In RDHOLD, the logic waits for the user to enter read mode by pressing the READ button. When that button is toggled, the read instruction, RDINST, is sent with two states (just like WRINST), the 24-bit address is sent a byte at a time, then the logic reaches the RDOUTP_REQ state, where a RD req. is sent to the SPI then the RDOUTP_ACK where we wait for the handshake to finish. After that we display the read value using the LEDs. Each time a value is output, the INPT_CNT is decremented. The FSM cycles through RDOUTP REQ, ACK, and LEDOUT until all values previously written have been read, as soon as that condition is met, the logic returns to the IDLE state. There is a up-only counter called DISP_CNT which is used to add a delay of 1 second after each LED output has been displayed in the LEDOUT state, this is for improved readability for the user. The FSM state diagram is shown on the next page.

NOTE: although not shown in the FSM state diagram (due to there not being enough space), when the RESET button is toggled, the FSM logic will return back to IDLE and reinitialise, ready to restart, regardless of what state it's in when RESET is pressed.

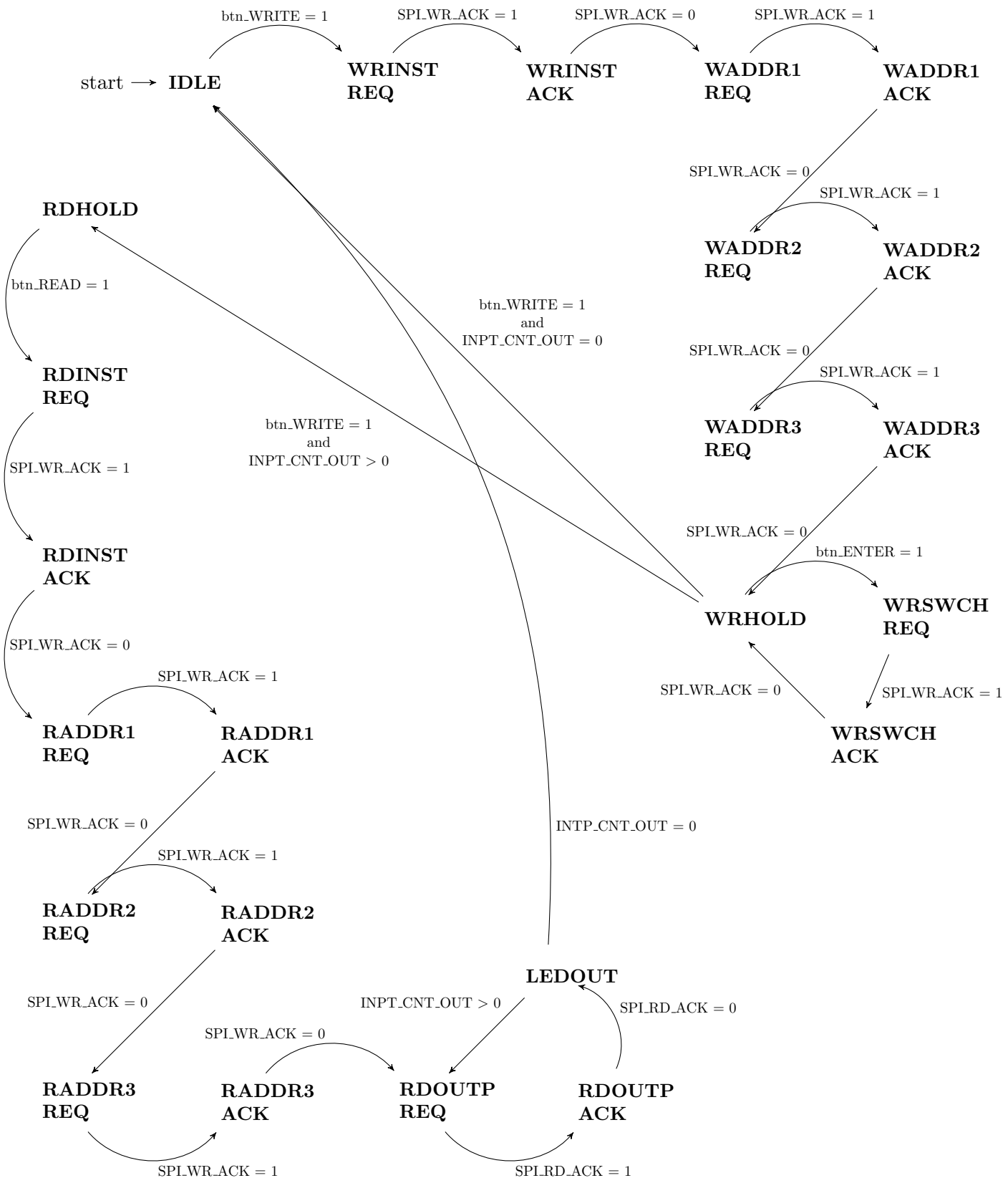

Figure 1: FSM state graph

STATE	*EN_SPI*	*SPI_WR_REQ*	*SPI_RD_REQ*	*DATA_TO_SPI*	*INPT_CNT_RST*	*INPT_CNT_EN_UP*	*INPT_CNT_EN_DOWN*	*DISP_CNT_RST*	*DISP_CNT_EN*	*LEDS*
IDLE	0	0	0	00000000	1	0	0	1	0	00000000
WRINST_REQ	1	1	0	WR ->00000010	0	0	0	0	0	00000000
RDINST_REQ				RD ->00000011						
WADDR1_REQ	1	1	0	ADDR1 ->SRAM_ADDRESS(23 : 16) ADDR2 ->SRAM_ADDRESS(15 : 8) ADDR3 ->SRAM_ADDRESS(7 : 0) WRSWCH ->SWITCHES	0	0	0	0	0	WRSWCH ->INPT_CNT_OUT else ->00000000
WADDR2_REQ										
WADDR3_REQ										
RADDR1_REQ										
RADDR2_REQ										
RADDR3_REQ										
WRSWCH_REQ										
WRINST_ACK	1	0	0	00000000	0	0	0	0	0	WRSWCH ->INPT_CNT_OUT else ->00000000
RDINST_ACK										
WADDR1_ACK										
WADDR2_ACK										
WADDR3_ACK										
RADDR1_ACK										
RADDR2_ACK										
RADDR3_ACK										
RDOUTP_ACK										
WRSWCH_ACK										
WRHOLD	1	0	0	00000000	0	1 when ENTER = 1 and INPT_CNT_OUT <input_limit	0	0	0	INPT_CNT_OUT
RDHOLD	0	0	0	00000000	0	0	0	0	0	00000000
RDOUTP_REQ	1	0	1	00000000	0	0	0	0	0	00000000
LEDOUT	1	0	0	00000000	0	0	1 when DISP_CNT_OUT = 0	0	1	DATA_FROM_SPI

Table 1: FSM table of outputs

2 - VHDL Entity Code

STUDENT_AREA Entity - Where the FSM Logic Lives

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

entity STUDENT_AREA is
    Generic (
        disp_delay : natural := 100000000
    );
    Port (
        CLK : in  STD_LOGIC;
        RST : in  STD_LOGIC;
        USER_PB : in  STD_LOGIC_VECTOR (4 downto 0);
        SWITCHES : in  STD_LOGIC_VECTOR (7 downto 0);
        LEDS : out  STD_LOGIC_VECTOR (7 downto 0);
        DATA_FROM_SPI : in  STD_LOGIC_VECTOR (7 downto 0);
        DATA_TO_SPI : out  STD_LOGIC_VECTOR (7 downto 0);
        EN_SPI : out  STD_LOGIC;
        SPI_WR_REQ : out STD_LOGIC;
        SPI_WR_ACK : in STD_LOGIC;
        SPI_RD_REQ : out STD_LOGIC;
        SPI_RD_ACK : in STD_LOGIC;
        SRAM_ADDRESS : in STD_LOGIC_VECTOR(23 downto 0)
    );
end STUDENT_AREA;

architecture Behavioral of STUDENT_AREA is
-- Signals for the btn assignments, I've renamed them so they're
-- more readable.
signal WRITE, ENTER, READ: STD_LOGIC;
-- We need to store the limit of how many inputs we can enter in
-- the SRAM during WRITE mode. While the SRAM datasheet states a
-- 128K x 8-bit organisation where we can store 128K bytes, we will
-- set a limit of 255.
constant input_limit : integer := 255;
```

```
-- Defining the FSM states and instantiating the state signals, a
-- grand total of 24 states have been designed for this circuit.
type state_type is (
    IDLE,        -- The usual IDLE state
    WRINST_REQ,  -- WRITE instruction is transmit and WR ACK goes high
    WRINST_ACK,  -- Wait for WR ACK to go back to low
    WADDR1_REQ,  -- Byte 1 of address is sent and WR ACK goes high
    WADDR1_ACK,  -- Wait for WR ACK to go back to low
    WADDR2_REQ,  -- Byte 2 of address is sent and WR ACK goes high
    WADDR2_ACK,  -- Wait for WR ACK to go back to low
    WADDR3_REQ,  -- Byte 3 of address is sent and WR ACK goes high
    WADDR3_ACK,  -- Wait for WR ACK to go back to low
    WRHOLD,      -- Wait for ENTER btn toggle before writing to SRAM
    WRSWCH_REQ,  -- Send SWITCHES and ACK goes high
    WRSWCH_ACK,  -- Wait for WR ACK to go back to low
    RDHOLD,      -- WRITE mode exited, waiting for READ btn toggle
    RDINST_REQ,  -- READ instruction is transmit and WR ACK goes high
    RDINST_ACK,  -- Wait for WR ACK to go back to low
    RADDR1_REQ,  -- Byte 1 of address is sent and WR ACK goes high
    RADDR1_ACK,  -- Wait for WR ACK to go back to low
    RADDR2_REQ,  -- Byte 2 of address is sent and WR ACK goes high
    RADDR2_ACK,  -- Wait for WR ACK to go back to low
    RADDR3_REQ,  -- Byte 3 of address is sent and WR ACK goes high
    RADDR3_ACK,  -- Wait for WR ACK to go back to low
    RDOUTP_REQ,  -- Send READ request, and RD ACK goes high
    RDOUTP_ACK,  -- Wait for RD ACK to go back to low
    LEDOUT       -- Displays byte that's read via the LEDS
);
signal state, next_state : state_type;
-- Internal signal instantiation for the up-down counter that
-- tracks how many values that've been written to the SRAM and
-- how many values that've been read.
signal INPT_CNT_OUT : UNSIGNED (log2(input_limit)-1 downto 0);
signal INPT_CNT_EN_UP, INPT_CNT_EN_DOWN, INPT_CNT_RST : STD_LOGIC;
-- Internal signal instantiation for the regular parameterisable
-- counter that's used to add a delay between each LED output for
-- increased readability for the user. This one's not up-down, it
-- only counts up and rolls-over to 0.
signal DISP_CNT_OUT : UNSIGNED (log2(disp_delay)-1 downto 0);
signal DISP_CNT_EN, DISP_CNT_RST : STD_LOGIC;

begin

WRITE <= USER_PB(1);
ENTER <= USER_PB(3);
READ <= USER_PB(2);

-- Instantiate the up-down counter which will be used to track
-- a count of values that have been written to SRAM and read
-- (popped) from the SRAM.
INPT_CNT : entity work.Param_Counter_UpDown
GENERIC MAP (LIMIT => input_limit)
PORT MAP(
        clk => clk,
        rst => INPT_CNT_RST,
        en_up => INPT_CNT_EN_UP,
    en_down => INPT_CNT_EN_DOWN,
        count_out => INPT_CNT_OUT
);
```

```
-- Instantiate the parameterisable counter that's used to add
-- a delay between each LED output.
DISP_CNT : entity work.Param_counter
GENERIC MAP (LIMIT => disp_delay)
PORT MAP(
        clk => clk,
        rst => DISP_CNT_RST,
        en => DISP_CNT_EN,
        count_out => DISP_CNT_OUT
);

--  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;


fsm_process : process (state, WRITE, ENTER, READ, SPI_WR_ACK, SPI_RD_ACK, INPT_CNT_OUT, DISP_CNT_OUT)
begin
    case STATE is

        when IDLE =>
            -- From IDLE the only state we can go to is the states for WRITE mode. We
            -- therefore wait for the WRITE button to be pressed.
            if (WRITE = '1') then
                next_state <= WRINST_REQ;
            else
                next_state <= state;
            end if;

        ----------------------------
        -- WRITE TRANSACTION LOGIC
        --------

        when WRINST_REQ =>
            -- Once the WRITE button is pressed from the IDLE state, we enter WRITE mode.
            -- In this state, we send the WRITE instruction to the SPI, when the WR_ACK
            -- line goes high, we then enter the WRINST_ACK state where we wait for the
            -- ACK line to go low again.
            if (SPI_WR_ACK = '1') then
                next_state <= WRINST_ACK;
            else
                next_state <= state;
            end if;
```

```
        when WRINST_ACK =>
            -- We have to wait until the SPI ACK line goes back to low again before we
            -- send the next instruction/value to the SPI. This state essentially waits
            -- for this to happen before it lets the circuit go to the next state. Please
            -- note that any state that ends with '_REQ' will have a state following that
            -- ends with '_ACK' that repeats this same logic - this will prevent repeated
            -- comments. The next state will be one where the first byte of the address is
            -- sent to the SPI.
            if (SPI_WR_ACK = '0') then
                next_state <= WADDR1_REQ;
            else
                next_state <= state;
            end if;

        when WADDR1_REQ =>
            -- Here, the first byte of the SRAM address is sent as part of the WRITE mode
            -- transaction.
            if (SPI_WR_ACK = '1') then
                next_state <= WADDR1_ACK;
            else
                next_state <= state;
            end if;
        when WADDR1_ACK =>
            -- Wait for the SPI ACK to go low before going to the next FSM state.
            -- address.
            if (SPI_WR_ACK = '0') then
                next_state <= WADDR2_REQ;
            else
                next_state <= state;
            end if;

        when WADDR2_REQ =>
            -- Here, the second byte of the SRAM address is sent.
            if (SPI_WR_ACK = '1') then
                next_state <= WADDR2_ACK;
            else
                next_state <= state;
            end if;
        when WADDR2_ACK =>
            -- Wait for ACK line to go low before continuing.
            if (SPI_WR_ACK = '0') then
                next_state <= WADDR3_REQ;
            else
                next_state <= state;
            end if;

        when WADDR3_REQ =>
            -- Here, the third and final byte of the SRAM address is sent.
            if (SPI_WR_ACK = '1') then
                next_state <= WADDR3_ACK;
            else
                next_state <= state;
            end if;
        when WADDR3_ACK =>
            -- Wait for ACK line to go low before continuing.
            if (SPI_WR_ACK = '0') then
                next_state <= WRHOLD;
            else
                next_state <= state;
            end if;
```

```
        when WRHOLD =>
            -- This state is reached once the WRITE transaction has been initialised
            -- i.e., the instruction was sent and received and all three bytes of the
            -- 24-bit SRAM address was sent and received. Now, we wait for the user to
            -- set the switches on the board and press the ENTER button to store a value
            -- or for the user to toggle ENTER to exit WRITE mode. If the user enters
            -- values and exits WRITE mode, then the logic can assume that the user will
            -- enter READ mode next. If the user exits WRITE without writing anything,
            -- we need to return back to IDLE. We only want to store as many values as
            -- defined by the input limit, ENTER toggles will be ignored if we've stored
            -- enough inputs.
            if (ENTER = '1' and INPT_CNT_OUT < input_limit) then
                next_state <= WRSWCH_REQ;                       -- User wants to enter values
            elsif (WRITE = '1' and INPT_CNT_OUT = 0) then
                next_state <= IDLE;                             -- Nothing was written
            elsif (WRITE = '1' and INPT_CNT_OUT > 0) then
                next_state <= RDHOLD;                           -- Something was written
            else
                next_state <= state;
            end if;

        when WRSWCH_REQ =>
            -- The value set by SWITCHES needs to be written to SRAM, so a WR req. is
            -- sent.
            if (SPI_WR_ACK = '1') then
                next_state <= WRSWCH_ACK;
            else
                next_state <= state;
            end if;
        when WRSWCH_ACK =>
            -- Wait for ACK low before returning back to WRHOLD so the user can enter
            -- another value into the SRAM or exit WRITE mode.
            if (SPI_WR_ACK = '0') then
                next_state <= WRHOLD;
            else
                next_state <= state;
            end if;


        ----------------------------
        -- READ TRANSACTION LOGIC
        --------

        when RDHOLD =>
            -- This state is reached when the user enters values into SRAM (INPT_CNT
            -- was incremented and the output is not equal to zero) and exits WRITE
            -- mode. Here we wait for the user to enter READ mode by toggling READ
            -- before we can send the READ instructions to the SPI.
            if (READ = '1') then
                next_state <= RDINST_REQ;
            else
                next_state <= state;
            end if;

        when RDINST_REQ =>
            -- Send the READ instruction to the SPI and wait for WR ACK.
            if (SPI_WR_ACK = '1') then
                next_state <= RDINST_ACK;
            else
                next_state <= state;
            end if;
```

```
        when RDINST_ACK =>
            -- Wait for RD ACK to return low.
            if (SPI_WR_ACK = '0') then
                next_state <= RADDR1_REQ;
            else
                next_state <= state;
            end if;

        when RADDR1_REQ =>
            -- Here, the first byte of the SRAM address is sent as part of the READ mode
            -- transaction.
            if (SPI_WR_ACK = '1') then
                next_state <= RADDR1_ACK;
            else
                next_state <= state;
            end if;
        when RADDR1_ACK =>
            -- Wait for the SPI ACK to go low before going to the next FSM state.
            -- address.
            if (SPI_WR_ACK = '0') then
                next_state <= RADDR2_REQ;
            else
                next_state <= state;
            end if;

        when RADDR2_REQ =>
            -- Here, the second byte of the SRAM address is sent.
            if (SPI_WR_ACK = '1') then
                next_state <= RADDR2_ACK;
            else
                next_state <= state;
            end if;
        when RADDR2_ACK =>
            -- Wait for ACK line to go low before continuing.
            if (SPI_WR_ACK = '0') then
                next_state <= RADDR3_REQ;
            else
                next_state <= state;
            end if;

        when RADDR3_REQ =>
            -- Here, the third and final byte of the SRAM address is sent.
            if (SPI_WR_ACK = '1') then
                next_state <= RADDR3_ACK;
            else
                next_state <= state;
            end if;
        when RADDR3_ACK =>
            -- Wait for ACK line to go low before continuing.
            if (SPI_WR_ACK = '0') then
                next_state <= RDOUTP_REQ;
            else
                next_state <= state;
            end if;
```

```
        when RDOUTP_REQ =>
            -- We send a RD req. to the SPI so we can return the value that was
            -- previously written to the SRAM.
            if (SPI_RD_ACK = '1') then
                next_state <= RDOUTP_ACK;
            else
                next_state <= state;
            end if;
        when RDOUTP_ACK =>
            -- Wait for the RD ACK to return to low before going to next state to
            -- display the output via the LEDS.
            if (SPI_RD_ACK = '0') then
                next_state <= LEDOUT;
            else
                next_state <= state;
            end if;

        when LEDOUT =>
            -- If there are still more values which needs to be read from the SRAM
            -- (i.e., the INPT_CNT hasn't been decremented to 0) then we return back
            -- to the RDOUTP_REQ state where we send a request to the SPI to read
            -- the next value from SRAM. If all values have been read and displayed
            -- (i.e., the counter has returned back to 0) then we can return back to
            -- the IDLE state, ready to enter WRITE mode again and write more values
            -- into the SRAM.
            if (INPT_CNT_OUT > 0 and DISP_CNT_OUT = disp_delay-1) then
                next_state <= RDOUTP_REQ;
            elsif (INPT_CNT_OUT = 0 and DISP_CNT_OUT = disp_delay-1) then
                next_state <= IDLE;
            else
                next_state <= state;
            end if;
    end case;
end process fsm_process;

----------------------------
-- OUTPUT COMBINATIONAL LOGIC
--------

SPI_WR_REQ <= '1' when state = WRINST_REQ or    -- WRITE MODE
                       state = WADDR1_REQ or
                       state = WADDR2_REQ or
                       state = WADDR3_REQ or
                       state = WRSWCH_REQ or
                       state = RDINST_REQ or    -- READ MODE
                       state = RADDR1_REQ or
                       state = RADDR2_REQ or
                       state = RADDR3_REQ else
              '0';

SPI_RD_REQ <= '1' when state = RDOUTP_REQ else  -- READ MODE: reading SRAM output
              '0';

EN_SPI <= '0' when state = IDLE or      -- Enable SPI except when logic is IDLE,
                  state = RDHOLD else   -- WRHOLD or RDHOLD, this is where logic
          '1';                          -- waits for user input (btn toggles).
```

```
DATA_TO_SPI <= "00000010" when state = WRINST_REQ else  -- WRITE instruction
               "00000011" when state = RDINST_REQ else  -- READ instruction
               SRAM_ADDRESS(23 downto 16) when state = WADDR1_REQ or state = RADDR1_REQ else
               SRAM_ADDRESS(15 downto 8) when state = WADDR2_REQ or state = RADDR2_REQ else
               SRAM_ADDRESS(7 downto 0) when state = WADDR3_REQ or state = RADDR3_REQ else
               SWITCHES when state = WRSWCH_REQ else    -- Send value of SWITCHES
               "00000000";                              -- zeros otherwise

LEDS <= STD_LOGIC_VECTOR(INPT_CNT_OUT) when state = WRHOLD or       -- Output # of inputs during
                                           state = WRSWCH_REQ or    -- these states...
                                           state = WRSWCH_ACK else
        DATA_FROM_SPI when state = LEDOUT else                      -- Output value from memory
        "00000000";                                                 -- Output zeros otherwise

INPT_CNT_RST <= '1' when state = IDLE else  -- Reset this counter when we come back to IDLE state
                '0';

INPT_CNT_EN_UP <= '1' when ENTER = '1' and                  -- Increment this counter only when
                           state = WRHOLD and               -- another value can be read into SRAM
                           INPT_CNT_OUT < input_limit else
                  '0';

INPT_CNT_EN_DOWN <= '1' when state = LEDOUT and     -- Decrement this counter when a value is popped
                             DISP_CNT_OUT = 0 else  -- from SRAM, at the start of displaying.
                    '0';

DISP_CNT_RST <= '1' when state = IDLE or state = RDOUTP_REQ else    -- Reset when IDLE or when loading
                '0';                                                -- next value in.

DISP_CNT_EN <= '1' when state = LEDOUT else -- Only start counting (add a delay) in the LEDOUT state
               '0';                         -- when the LED is displaying an output.

end Behavioral;
```

INPT_CNT Entity - The Up-Down Counter I Made

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

use work.DigEng.all;

entity Param_Counter_UpDown is
    generic (
        LIMIT : NATURAL := 17
    );
    port (
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        en_up : in STD_LOGIC;
        en_down : in STD_LOGIC;
        count_out : out UNSIGNED (log2(LIMIT)-1 downto 0)
    );
end Param_Counter_UpDown;

architecture Behavioral of Param_Counter_UpDown is

signal count_int : UNSIGNED (log2(LIMIT)-1 downto 0);

begin

-- An up-down counter without roll-over with two synchronous enables
-- (one for increment and another for decrement) and a synchronous
-- reset. When en_up is high, the counter will increment up to the
-- value set by LIMIT without rolling over to 0. When en_down is high
-- the counter will decrement until 0 is reached without rolling over
-- to LIMIT.
counter: process (clk)
begin
    if rising_edge(CLK) then
        if (rst = '1') then
            count_int <= (others => '0');       -- Counter resets.
        elsif (en_up = '1') then
            if (count_int < LIMIT) then         -- Increment when en_up until
                count_int <= count_int + 1;     -- LIMIT is reached.
            end if;
        elsif (en_down = '1') then
            if (count_int > 0) then             -- Decrement when en_down until
                count_int <= count_int - 1;     -- 0 is reached.
            end if;
        end if;
    end if;
end process counter;

-- Maps the internal counter value to the output signal.
count_out <= count_int;

end Behavioral;
```

3 - VHDL Testbench Code

```
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 inputs will initialise to
--      zeros, followed by a reset button click.
--
--  TEST 1 (verify by inspecting waveform):
--      Enter WRITE mode by pressing the write btn, ensure all
--      handshaking aspects of the SPI is working correctly,
--      mainly SPI enables, reqs are sent, acks are received and
--      the correct data is sent.
--
--  TEST 2 (verify by inspecting waveform):
--      Following from Test 1, exit WRITE mode by toggling the
--      write btn. The logic should be able to deactivate the SPI
--      and return back to IDLE.
--
--  TEST 3 (verify by inspecting waveform):
--      Toggle the READ btn to try to enter read mode. This will be blocked
--      by the logic as you can only enter read mode after storing 1 or more
--      values into the SRAM in the write mode.
--
--  TEST 4 (verify by inspecting waveform):
--      Following from TEST 2, toggle the write button again to enter
--      WRITE mode, then store 10 switch values, starting from h"00" and
--      incrementing by 1 each time. Then exit WRITE mode in preparation
--      for the next test. This test should verify that the SPI handshaking
--      logic works when writing data to the SRAM.
--
--  TEST 5 (verify by inspecting waveform):
--      Toggle the WRITE btn to try to enter write mode again. This should be
--      blocked by the logic which prevents write mode to be entered after
--      writing 1 or more values and exiting beforehand.
--
--  TEST 6 (verify by inspecting self-check console output):
--      As we've stored 10 (more than 0) switch values and exited WRITE
--      mode in the previous test, the circuit should now be in RDHOLD
--      state, waiting for the user to toggle the READ btn. As soon as
--      that btn is toggled, the circuit should read and display the
--      exact values that we're previously written, and should stop as
--      soon as all values previously written have been displayed. This
--      test will verify that the circuit can perform this function.
```

```
ARCHITECTURE behavior OF TOP_LEVEL_tb IS

    --Inputs
    signal GCLK : std_logic := '0';
    signal BTN : std_logic_vector(4 downto 0);
    signal SW : std_logic_vector(7 downto 0);

    --Outputs
    signal LD : std_logic_vector(7 downto 0);

    -- Internal SPI signals
    signal SPI_MISO: STD_LOGIC;
    signal SPI_MOSI: STD_LOGIC;
    signal SPI_CS_INV: STD_LOGIC;
    signal SPI_HOLD_INV: STD_LOGIC;
    signal SPI_SCK: STD_LOGIC;

    -- Clock period definitions
    constant GCLK_period : time := 10 ns;

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,
    LD => LD,
    SPI_MISO => SPI_MISO,
    SPI_SCK => SPI_SCK,
    SPI_CS_INV => SPI_CS_INV,
    SPI_HOLD_INV => SPI_HOLD_INV,
    SPI_MOSI => SPI_MOSI
);

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

SRAM: entity work.SRAM_Model
PORT MAP (
    SPI_MISO => SPI_MISO,
    SPI_SCK => SPI_SCK,
    SPI_CS_INV => SPI_CS_INV,
    SPI_HOLD_INV => SPI_HOLD_INV,
    SPI_MOSI => SPI_MOSI
);

-- Stimulus process
set_inputs: process
begin
    -- hold reset state for 1000 ns.
    wait for 1000 ns;
```

```
    ----------------------------
    -- Global Reset & Initialisation
    --------

    SW <= "00000000";
    BTN <= "00000";
    wait for GCLK_period*18;
    BTN(0) <= '1';              -- toggle the RESET btn
    wait for GCLK_period*36;
    BTN(0) <= '0';

    wait for GCLK_period*120;

    ----------------------------
    -- TEST 1
    --------

    BTN(1) <= '1';              -- toggle the WRITE btn
    wait for GCLK_period*36;
    BTN(1) <= '0';
                                -- check if WRITE mode is
    wait for GCLK_period*500;   -- entered by verifying SPI
                                -- handshaking int. signals.

    ----------------------------
    -- TEST 2
    --------

    BTN(1) <= '1';              -- toggle the WRITE btn
    wait for GCLK_period*36;
    BTN(1) <= '0';
                                -- check if WRITE mode is
    wait for GCLK_period*100;   -- exited.

    ----------------------------
    -- TEST 3
    --------

    BTN(2) <= '1';              -- toggle the READ btn
    wait for GCLK_period*36;
    BTN(2) <= '0';
                                -- check if READ mode request
    wait for GCLK_period*100;   -- is ignored.

    ----------------------------
    -- TEST 4
    --------

    BTN(1) <= '1';              -- toggle the WRITE btn to
    wait for GCLK_period*36;    -- enter WRITE MODE
    BTN(1) <= '0';
                                -- Wait for logic to init
    wait for GCLK_period*450;   -- the SPI and enter WRHOLD
                                -- before entering vals.
```

```
    test_4_input_loop : for i in 0 to 9 loop
        SW <= STD_LOGIC_VECTOR(
                to_unsigned(i, SW'length)
              );                    -- SW set to for-loop counter value
        BTN(3) <= '1';              -- toggle the ENTER btn to enter the
        wait for GCLK_period*36;    -- switch val to SRAM
        BTN(3) <= '0';
        wait for GCLK_period*120;   -- wait before next input.
    end loop;

    BTN(1) <= '1';              -- toggle the WRITE btn to
    wait for GCLK_period*36;    -- exit WRITE MODE, prepping
    BTN(1) <= '0';              -- for TEST 4.

    wait for GCLK_period*100;

    ----------------------------
    -- TEST 5
    --------

    BTN(1) <= '1';              -- toggle the WRITE btn
    wait for GCLK_period*36;
    BTN(1) <= '0';
                                -- check if WRITE mode request
    wait for GCLK_period*100;   -- is ignored.

    ----------------------------
    -- TEST 6
    --------
                                -- toggle the READ btn to enter READ
    BTN(2) <= '1';              -- mode. The circuit will send the
    wait for GCLK_period*36;    -- correct instructions to the SPI and
    BTN(2) <= '0';              -- will read values until all inputted
                                -- are read.
    wait;
end process;

check_outputs : process
begin

    -- Wait for hold and the global reset
    wait for GCLK_period*274;
    -- Wait for TEST 1, 2, 3, 4 and 5
    wait for GCLK_period*536;
    wait for GCLK_period*136;
    wait for GCLK_period*136;
    wait for GCLK_period*2182;
    wait for GCLK_period*136;
```

```
    -- Wait until logic starts displaying to the LED + a few clock
    -- cycles (~12 clock cycles).
    wait for GCLK_period*455;
    test_6_check_loop : for i in 0 to 9 loop
        -- Check if correct LED outputs are displayed.
        assert (( LD = STD_LOGIC_VECTOR(to_unsigned(i, SW'length)) ))
        report "TEST 6 VECTOR " & integer'image(i) &
        " FAIL. Observed LED Output = " & integer'image(to_integer(unsigned(LD))) &
        " Expected LED Output = " & integer'image(to_integer(to_unsigned(i, SW'length)))
        severity warning;
        report "TEST 6 VECTOR " & integer'image(i) & " PASS."
        severity note;
        -- It takes 119 clock cycles between each LEDOUT state
        wait for GCLK_period*119;
    end loop;

    wait;
end process;

END;
```

6 - Synthesis Report

RTL Component Statistics

```
Start RTL Component Statistics
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Adders :
           2 Input     27 Bit       Adders := 1
           2 Input      8 Bit       Adders := 1
           2 Input      3 Bit       Adders := 1
+---Registers :
                       27 Bit    Registers := 1
                        8 Bit    Registers := 4
                        3 Bit    Registers := 1
                        2 Bit    Registers := 6
                        1 Bit    Registers := 34
+---Muxes :
           2 Input     27 Bit        Muxes := 1
           2 Input      8 Bit        Muxes := 7
           3 Input      8 Bit        Muxes := 1
          28 Input      5 Bit        Muxes := 1
           2 Input      2 Bit        Muxes := 6
           4 Input      2 Bit        Muxes := 2
           2 Input      1 Bit        Muxes := 8
          24 Input      1 Bit        Muxes := 1
---------------------------------------------------------------------------------
Finished RTL Component Statistics
```

RTL Hierarchical Component Statistics

```
Start RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
Hierarchical RTL Component report
Module Param_Counter_UpDown
Detailed RTL Component Info :
+---Adders :
           2 Input      8 Bit       Adders := 1
+---Registers :
                        8 Bit    Registers := 1
+---Muxes :
           2 Input      2 Bit        Muxes := 1
           2 Input      1 Bit        Muxes := 1
Module Param_Counter
Detailed RTL Component Info :
+---Adders :
           2 Input     27 Bit       Adders := 1
+---Registers :
                       27 Bit    Registers := 1
+---Muxes :
           2 Input     27 Bit        Muxes := 1
Module STUDENT_AREA
Detailed RTL Component Info :
+---Muxes :
           2 Input      8 Bit        Muxes := 6
           3 Input      8 Bit        Muxes := 1
          28 Input      5 Bit        Muxes := 1
           2 Input      1 Bit        Muxes := 6
          24 Input      1 Bit        Muxes := 1
Module xpm_cdc_single
Detailed RTL Component Info :
+---Registers :
                        2 Bit    Registers := 1
Module SPI_MASTER
Detailed RTL Component Info :
+---Adders :
           2 Input      3 Bit       Adders := 1
+---Registers :
                        8 Bit    Registers := 3
                        3 Bit    Registers := 1
                        2 Bit    Registers := 1
                        1 Bit    Registers := 4
+---Muxes :
           2 Input      8 Bit        Muxes := 1
           2 Input      2 Bit        Muxes := 5
           4 Input      2 Bit        Muxes := 2
           2 Input      1 Bit        Muxes := 1
Module Debouncer
Detailed RTL Component Info :
+---Registers :
                        1 Bit    Registers := 3
---------------------------------------------------------------------------------
Finished RTL Hierarchical Component Statistics
```