Lab 6

Carter Owens, Kyle Turley

Procedures

Introduction. The goal of this lab was to implement an enhanced version of Lab 5's Accumulator design with additional functionality using two Finite State Machines (FSMs) operating at different clock frequencies and a clock-domain-crossing FIFO. The system accepts input from toggle switches, stores values in a FIFO, and accumulates them when specific conditions are met.

The procedures for this lab were as follows:

1. Create a Quartus project extending Lab 5's accumulator implementation
2. Design and implement two FSMs operating at different clock frequencies (5 MHz and 12.5 MHz)
3. Implement a clock-domain-crossing FIFO for safe data transfer between clock domains
4. Create the necessary PLL configurations for clock generation
5. Implement the required button functionality (reset and store)
6. Design the hex display logic for the 24-bit accumulated value
7. Test and verify the complete system functionality

Key Requirements: - Reset button to clear accumulated value - Store button to add toggle switch values to FIFO - FIFO auto-draining when 5 items are stored - 24-bit accumulator value displayed in hexadecimal - Two FSMs running at different clock speeds (5 MHz and 12.5 MHz) - LED display reflecting toggle switch states

Issues, Errors, and Stumbles. Several challenges were encountered during the implementation of this lab. The primary difficulties centered around understanding and properly utilizing the IP catalog for creating the necessary components. Initial module creation had to be redone due to incomplete consideration of the overall design architecture.

A significant challenge arose in the implementation of the state machines for reading and writing operations. The need to separate logic to prevent signal interference between the two clock domains required careful consideration. A critical issue was discovered regarding the master reset implementation - the original design caused the clocks to stop functioning when reset was high, creating a deadlock situation where the state machines couldn't progress to deassert the reset signal.

The reset mechanism proved particularly problematic as it affected the PLLs, effectively "bricking" the system. A key learning from this experience is that in future implementations, the reset debouncer should be detached from the PLLs to ensure the reset signal can properly return to low without disrupting the clock generation.

Results

The final implementation successfully met all the required specifications:

1. **Dual Clock Domain Operation:**
   - Write-side FSM operating at 5 MHz handling button inputs and FIFO writes
   - Read-side FSM operating at 12.5 MHz managing FIFO reads and accumulator updates
2. **FIFO Implementation:**
   - Successfully stores up to 5 values from toggle switches
   - Automatically triggers accumulation when full
   - Properly handles clock domain crossing
3. **User Interface:**
   - Six 7-segment displays showing 24-bit accumulator value in hexadecimal
   - Functional reset button clearing the accumulator

- Store button properly capturing toggle switch values
- LEDs accurately reflecting toggle switch states

4. **System Integration:**
   - Clock domain crossing handled safely through dedicated FIFO
   - Proper debouncing of button inputs
   - Stable operation across reset conditions

The system successfully demonstrates the principles of multi-clock domain design and safe data transfer between clock domains, while providing a user-friendly interface for input and output.

Figures and Code

The following sections contain the key VHDL modules implementing the system functionality:

Accumulator Module (accumulator.vhd)

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

entity accumulator is
    port(
        -- [INPUTS] --
        ADC_CLK_10  : in std_logic;                         -- System clock
        KEY         : in std_logic_vector(1 downto 0);    -- The pressable buttons
        SW          : in unsigned(9 downto 0);              -- The 10 switches

        -- [OUTPUTS] --
        HEX0        : out std_logic_vector(7 downto 0);     -- Segment display outs:
        HEX1        : out std_logic_vector(7 downto 0);
        HEX2        : out std_logic_vector(7 downto 0);
        HEX3        : out std_logic_vector(7 downto 0);
        HEX4        : out std_logic_vector(7 downto 0);
        HEX5        : out std_logic_vector(7 downto 0);

        LEDR        : out unsigned(9 downto 0)              -- The 10 output LEDs
    );
end entity accumulator;

-- Architecture implementation omitted for brevity. See full source code in appendix.
```

FIFO Module (fifo.vhd)

```
-- Generated using Altera's IP Catalog
library ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.all;

ENTITY fifo IS
    PORT
    (
        aclr        : IN STD_LOGIC  := '0';
        data        : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
        rdclk       : IN STD_LOGIC ;
        rdreq       : IN STD_LOGIC ;
        wrclk       : IN STD_LOGIC ;
```

```
        wrreq       : IN STD_LOGIC ;
        q           : OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
        rdusedw     : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
        wrusedw     : OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
END fifo;

-- Architecture implementation omitted for brevity. See full source code in appendix.
```

Button Debounce Module (debounce.vhd)

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

entity debounce is
    port(
        -- [INPUTS] --
        clk     : in std_logic;  -- The clock that drives the state machine
        d_in    : in std_logic;  -- The value to debounce

        -- [OUTPUTS] --
        d_out : out std_logic    -- The debounced value
    );
end entity debounce;

-- Architecture implementation omitted for brevity. See full source code in appendix.
```

PLL Modules

Both PLL modules (pll5mhz.vhd and pll12_5mhz.vhd) were generated using Altera's IP Catalog. The 5 MHz PLL divides the input clock by 2 and the 12.5 MHz PLL multiplies by 5 and divides by 4 to achieve the desired frequencies from the 10 MHz input clock.

Conclusion

This lab provided valuable experience in designing and implementing multi-clock domain digital systems. The challenges encountered, particularly with reset logic and clock domain crossing, offered important insights into real-world digital design considerations. The project successfully demonstrated the implementation of complex state machines, clock domain crossing techniques, and proper handling of user inputs and outputs.

Key learnings include: - The importance of careful planning in multi-clock domain designs - Proper implementation of reset logic in systems with PLLs - Techniques for safe data transfer between clock domains - Practical considerations in FSM design and implementation

The experience gained from this lab is directly applicable to real-world digital design scenarios where multiple clock domains and complex state machines are common requirements.

Appendix

Accumulator Module (accumulator.vhd)

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

entity accumulator is
    port(
        -- [INPUTS] --
```

```
        ADC_CLK_10  : in std_logic;                             -- System clock
        KEY         : in std_logic_vector(1 downto 0);      -- The pressable buttons
        SW              : in unsigned(9 downto 0);                  -- The 10 switches

        -- [OUTPUTS] --
        HEX0            : out std_logic_vector(7 downto 0);     -- Segment display outs:
        HEX1            : out std_logic_vector(7 downto 0);
        HEX2            : out std_logic_vector(7 downto 0);
        HEX3            : out std_logic_vector(7 downto 0);
        HEX4            : out std_logic_vector(7 downto 0);
        HEX5            : out std_logic_vector(7 downto 0);

        LEDR            : out unsigned(9 downto 0)                  -- The 10 output LEDs
    );
end entity accumulator;

architecture behavioral of accumulator is
    -- [COMPONENTS] --
    component seg
        port(
            point   : in std_logic;
            count   : in unsigned(3 downto 0);
            output: out std_logic_vector(7 downto 0)
        );
    end component seg;

    component debounce
        port(
            clk     : in std_logic;
            d_in    : in std_logic;
            d_out : out std_logic
        );
    end component debounce;

    component fifo is
        port (
            aclr        : in std_logic;
            data        : IN std_logic_vector (9 downto 0);
            rdclk       : in std_logic;
            rdreq       : in std_logic;
            wrclk       : in std_logic;
            wrreq       : in std_logic;
            q           : out std_logic_vector (9 downto 0);
            rdusedw : out std_logic_vector (2 downto 0);
            wrusedw : out std_logic_vector (2 downto 0)
        );
    end component fifo;

    component pll5mhz IS
        port (
            areset  : in std_logic;
            inclk0  : in std_logic;
            c0          : out std_logic;
            locked  : out std_logic
        );
    end component pll5mhz;
```

```
    component pll12_5mhz IS
        port (
            areset  : in std_logic;
            inclk0  : in std_logic;
            c0          : out std_logic;
            locked  : out std_logic
        );
    end component pll12_5mhz;

    -- [SIGNALS & CONSTANTS] --
    signal key_inverted      : std_logic_vector(1 downto 0);
    signal rst_master    : std_logic := '0';

    -- clocks
    signal clk_5mhz  : std_logic;
    signal locked_5mhz : std_logic;

    signal clk_12_5mhz   : std_logic;
    signal locked_12_5mhz : std_logic;

    -- fifo signals
    signal fifo_in          : std_logic_vector (9 downto 0) := (others => '0');
    signal fifo_out             : std_logic_vector (9 downto 0);
    signal fifo_read            : std_logic := '0';
    signal fifo_write       : std_logic := '0';
    signal fifo_read_count  : std_logic_vector (2 downto 0);

    signal psh_db   : std_logic;
    signal rst_db   : std_logic;
    signal count    : unsigned(23 downto 0) := (others => '0');

    signal a_current_state  : std_logic_vector(1 downto 0) := "00";
    signal a_next_state     : std_logic_vector(1 downto 0) := "00";

    signal b_current_state  : std_logic_vector(1 downto 0) := "00";
    signal b_next_state         : std_logic_vector(1 downto 0) := "00";

    constant A_IDLE : std_logic_vector(1 downto 0) := "00";
    constant A_RST  : std_logic_vector(1 downto 0) := "01";
    constant A_PUSH : std_logic_vector(1 downto 0) := "10";
    constant A_HOLD : std_logic_vector(1 downto 0) := "11";

    constant B_IDLE : std_logic_vector(1 downto 0) := "00";
    constant B_PULL : std_logic_vector(1 downto 0) := "01";
    constant B_RST  : std_logic_vector(1 downto 0) := "10";

begin
    -- [INSTANCES] --
    seg0_impl : seg port map(point => '0',  count => count(3 downto 0), output => HEX0);
    seg1_impl : seg port map(point => '0',  count => count(7 downto 4), output => HEX1);
    seg2_impl : seg port map(point => '0',  count => count(11 downto 8), output => HEX2);
    seg3_impl : seg port map(point => '0',  count => count(15 downto 12), output => HEX3);
    seg4_impl : seg port map(point => '0',  count => count(19 downto 16), output => HEX4);
    seg5_impl : seg port map(point => '0',  count => count(23 downto 20), output => HEX5);
```

```
pll5mhz_impl : pll5mhz port map(
    areset => '0',
    inclk0 => ADC_CLK_10,
    c0 => clk_5mhz,
    locked => locked_5mhz
);

pll12_5mhz_impl : pll12_5mhz port map(
    areset => '0',
    inclk0 => ADC_CLK_10,
    c0 => clk_12_5mhz,
    locked => locked_12_5mhz
);

psh_db_impl : debounce port map (clk => clk_5mhz, d_in => key_inverted(0), d_out => psh_db);
rst_db_impl : debounce port map (clk => clk_5mhz, d_in => key_inverted(1), d_out => rst_db);

fifo_impl : fifo port map (
    aclr => rst_master,
    data => fifo_in,
    rdclk => clk_12_5mhz,
    rdreq => fifo_read,
    wrclk => clk_5mhz,
    wrreq => fifo_write,
    q => fifo_out,
    rdusedw => fifo_read_count,
    wrusedw => open
);

-- [DIRECT BEHAVIOR] --
-- Directly assign the values of the switches to the LEDs
LEDR <= SW;
key_inverted <= not KEY(1) & not KEY(0);
fifo_in <= std_logic_vector(SW);

-- [PROCESSES] --
-- [BLOCK A] --
-- Handles the button presses and FIFO writing
process (clk_5mhz) begin
    if rising_edge(clk_5mhz) then
        a_current_state <= a_next_state;
        case (a_next_state) is
            when A_RST =>
                rst_master <= '1';
                fifo_write <= '0';

            when A_PUSH =>
                rst_master <= '0';
                fifo_write <= '1';

            when others =>
                rst_master <= '0';
                fifo_write <= '0';

     end case;
    end if;
```

```
    end process;

    process (a_current_state, psh_db, rst_db) begin
        case (a_current_state) is
            when A_IDLE =>
              if rst_db = '1' then a_next_state <= A_RST;
              elsif psh_db = '1' then a_next_state <= A_PUSH;
              else a_next_state <= A_IDLE; end if;

            when A_RST =>
              if rst_db = '1' then a_next_state <= A_HOLD;
              else a_next_state <= A_IDLE; end if;

            when A_PUSH=>
              if psh_db = '1' then a_next_state <= A_HOLD;
              else a_next_state <= A_IDLE; end if;

            when A_HOLD =>
              if psh_db = '1' or rst_db = '1' then a_next_state <= A_HOLD;
              else a_next_state <= A_IDLE; end if;

            when others => a_next_state <= A_IDLE;
     end case;
end process;

    -- [BLOCK B] --
    -- Handles FIFO reading and the accumulator
    process (clk_12_5mhz) begin
        if rising_edge(clk_12_5mhz) then
            b_current_state <= b_next_state;
            case (b_next_state) is
                when B_IDLE =>
                    fifo_read <= '0';
                    count <= count;

                when B_PULL =>
                    fifo_read <= '1';
                    count <= count + unsigned(fifo_out);

                when B_RST =>
                    fifo_read <= '0';
                    count <= (others => '0');

                when others =>
                    fifo_read <= '0';
                    count <= count;

            end case;
        end if;
    end process;

    process (b_current_state, fifo_read_count, rst_master) begin
        case (b_current_state) is
            when B_IDLE =>
                if fifo_read_count = "101" then b_next_state <= B_PULL;
                elsif rst_master = '1' then b_next_state <= B_RST;
```

```
                else b_next_state <= B_IDLE; end if;

            when B_PULL =>
                if fifo_read_count = "000" then b_next_state <= B_IDLE;
                else b_next_state <= B_PULL; end if;

            when B_RST =>
                if rst_master = '1' then b_next_state <= B_RST;
                else b_next_state <= B_IDLE; end if;

            when others => b_next_state <= B_IDLE;
        end case;
    end process;

end architecture behavioral;
```

FIFO Module (fifo.vhd)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.all;

ENTITY fifo IS
    PORT
    (
        aclr        : IN STD_LOGIC  := '0';
        data        : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
        rdclk       : IN STD_LOGIC ;
        rdreq       : IN STD_LOGIC ;
        wrclk       : IN STD_LOGIC ;
        wrreq       : IN STD_LOGIC ;
        q       : OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
        rdusedw     : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
        wrusedw     : OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
END fifo;


ARCHITECTURE SYN OF fifo IS

    SIGNAL sub_wire0    : STD_LOGIC_VECTOR (9 DOWNTO 0);
    SIGNAL sub_wire1    : STD_LOGIC_VECTOR (2 DOWNTO 0);
    SIGNAL sub_wire2    : STD_LOGIC_VECTOR (2 DOWNTO 0);



    COMPONENT dcfifo
    GENERIC (
        intended_device_family      : STRING;
        lpm_numwords        : NATURAL;
        lpm_showahead       : STRING;
        lpm_type        : STRING;
        lpm_width       : NATURAL;
        lpm_widthu      : NATURAL;
        overflow_checking       : STRING;
```

```
        rdsync_delaypipe        : NATURAL;
        read_aclr_synch     : STRING;
        underflow_checking      : STRING;
        use_eab     : STRING;
        write_aclr_synch        : STRING;
        wrsync_delaypipe        : NATURAL
    );
    PORT (
            aclr    : IN STD_LOGIC ;
            data    : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
            rdclk   : IN STD_LOGIC ;
            rdreq   : IN STD_LOGIC ;
            wrclk   : IN STD_LOGIC ;
            wrreq   : IN STD_LOGIC ;
            q   : OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
            rdusedw : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
            wrusedw : OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
    END COMPONENT;

BEGIN
    q    <= sub_wire0(9 DOWNTO 0);
    rdusedw    <= sub_wire1(2 DOWNTO 0);
    wrusedw    <= sub_wire2(2 DOWNTO 0);

    dcfifo_component : dcfifo
    GENERIC MAP (
        intended_device_family => "MAX 10",
        lpm_numwords => 8,
        lpm_showahead => "OFF",
        lpm_type => "dcfifo",
        lpm_width => 10,
        lpm_widthu => 3,
        overflow_checking => "ON",
        rdsync_delaypipe => 4,
        read_aclr_synch => "OFF",
        underflow_checking => "ON",
        use_eab => "ON",
        write_aclr_synch => "OFF",
        wrsync_delaypipe => 4
    )
    PORT MAP (
        aclr => aclr,
        data => data,
        rdclk => rdclk,
        rdreq => rdreq,
        wrclk => wrclk,
        wrreq => wrreq,
        q => sub_wire0,
        rdusedw => sub_wire1,
        wrusedw => sub_wire2
    );



END SYN;
```

Button Debounce Module (debounce.vhd)

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

entity debounce is
    port(
        -- [INPUTS] --
        clk     : in std_logic; -- The clock that drives the state machine
        d_in    : in std_logic; -- The value to debounce

        -- [OUTPUTS] --
        d_out : out std_logic   -- The debounced value
    );
end entity debounce;

architecture behavioral of debounce is
    -- [SIGNALS & CONSTANTS] --
    signal current_state : std_logic_vector(1 downto 0) := "00";
    signal next_state       : std_logic_vector(1 downto 0) := "00";

    constant IDLE           : std_logic_vector(1 downto 0) := "00";
    constant PUSH_DOWN  : std_logic_vector(1 downto 0) := "01";
    constant HOLD           : std_logic_vector(1 downto 0) := "10";
    constant PULL_UP        : std_logic_vector(1 downto 0) := "11";


begin
    -- [PROCESSES] --

    process (clk)
    begin
        if rising_edge(clk) then
            current_state <= next_state;
            case (next_state) is
                when IDLE => d_out <= '0';
                when PUSH_DOWN => d_out <= '0';
                when HOLD => d_out <= '1';
                when PULL_UP => d_out <= '1';
                when others => d_out <= '0';
            end case;
        end if;

    end process;

    process (current_state, d_in)
    begin
        case (current_state) is
            when IDLE =>
                -- If the button is zero, then idle.
                -- If the button is one, then start debouncing to 1
                if d_in = '0' then next_state <= IDLE;
                else next_state <= PUSH_DOWN; end if;

            when PUSH_DOWN =>
                -- If the button is zero again, then go back to IDLE
```

```
                -- If the button is two again, then HOLD that value
                if d_in = '0' then next_state <= IDLE;
                else next_state <= HOLD; end if;

            when HOLD =>
                -- If the button is zero, begin debouncing to 0
                -- If the button is one, then HOLD that value
                if d_in = '0' then next_state <= PULL_UP;
                else next_state <= HOLD; end if;

            when PULL_UP =>
                -- If the button is zero, then go back to IDLE
                -- If the button is one again, then HOLD that value
                if d_in = '0' then next_state <= IDLE;
                else next_state <= HOLD; end if;

            when others => next_state <= IDLE;
        end case;

    end process;


end architecture behavioral;
```