Lab 6 Report

Matthew Crump A01841001
Nicholas Williams A02057223

Objectives

The objective of this lab is to implement a FIFO with the write port and read ports in different clock domains from each other. Learn how to use the IP catalog built into Quartus to customize a FIFO and PLL. Design and implement more complex state machines compared to the last one.

Procedure

We started out by reading the lab-6 requirements for the FIFO, and we reviewed the lecture notes on FIFOs and PLLs. We then used the top module from Lab 5 as a base, since this lab uses the same peripherals. We determined that we needed to add a PLL with 2 output clocks, one at 5 MHz and the other at 12.5 MHz. We removed the reset and lock signals as we wouldn't be using them. On the FIFO, we used: the read/write enables, async clear, dual clocks, empty, and used words.

After creating those modules, we needed to sort out how the read and write state machines were going to work. Since we created the FIFO first, we just needed to figure out which signals were for the read or write state machine then how we would control them. For the read machine, we waited in one state for the used words to reach 5 then moved to the next state. From here the FIFO was read consecutively and accumulated until the empty flag was raised. We then return to the waiting state. For the write machine, we had a similar process to what was in Lab 5, with the exception of having a write enable signal on the transition to save the value from the switches.

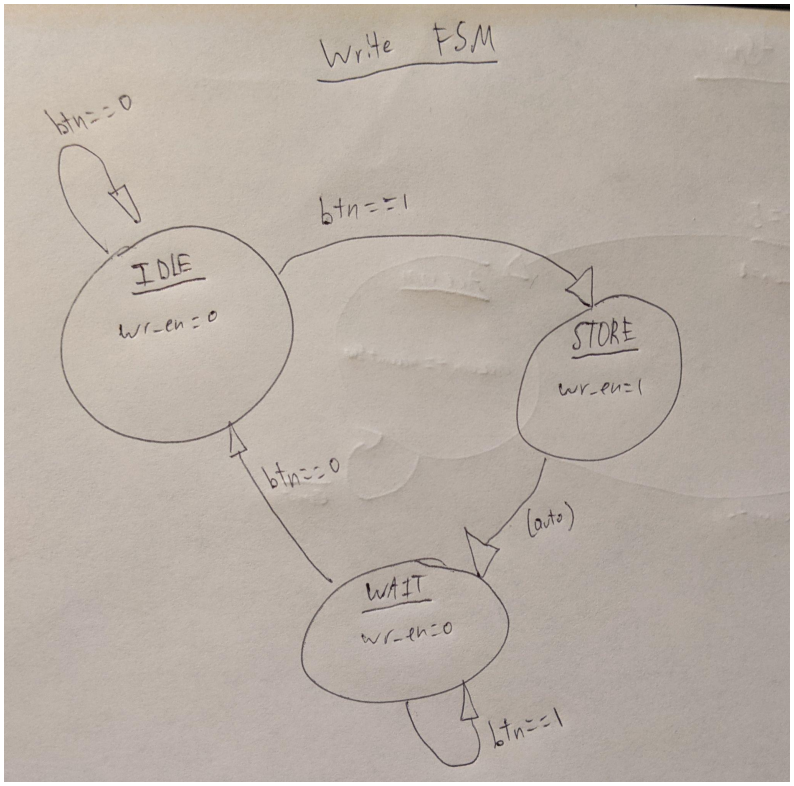


Figure 1. Write Finite State Machine

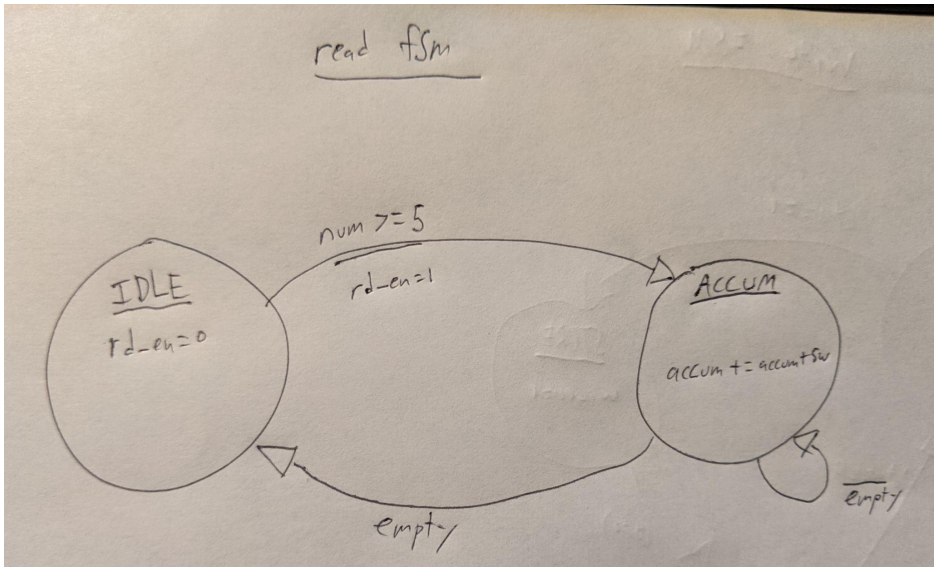


Figure 2. Read Finite State Machine

Results

We created a testbench to make sure that our two state machines were working properly together. Figure 3 shows the waveform from simulation. The waveform shows

that the read clock operates at a faster frequency than the write clock. The ACCUM state occurs once there are five values in the FIFO. When we first simulated our design we realized that we were writing six values to the FIFO before emptying the FIFO. We discovered that our clocks were slightly off so that our read enable value was getting set one read clock cycle too early. We fixed the problem and decided that our design was ready to be programmed to the DE10-lite board.

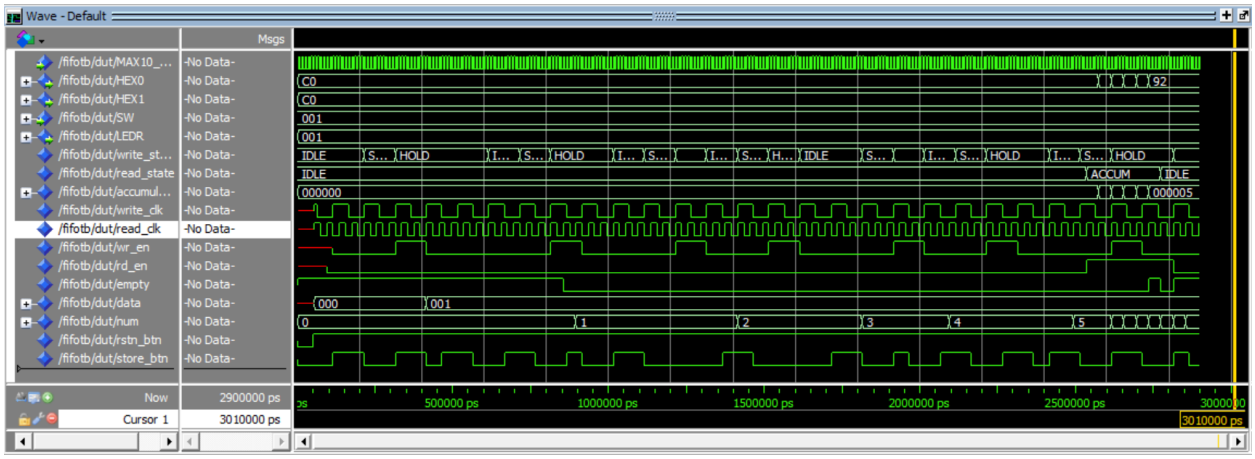
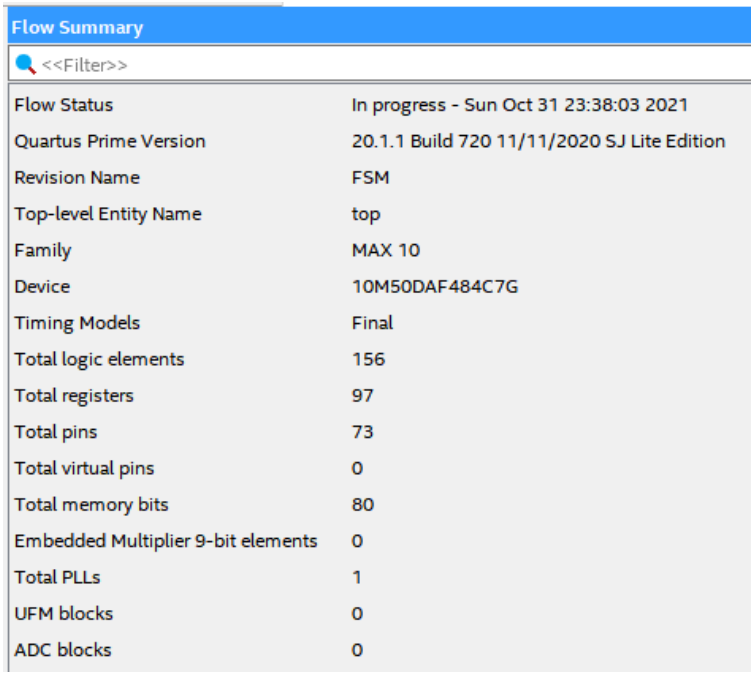


Figure 3. Simulation Waveform

After compiling our design we looked through the flow summary. The first time we looked through the flow summary it showed that we were using zero registers. Fortunately, we noticed this before we programmed the device. We then realized that we had set our project up incorrectly in quartus. We fixed the simple mistake and received the flow summary shown in Figure 4. The flow summary in figure 4 was a lot closer to what we expected.



Flow Summary

<<Filter>>

Flow Status	In progress - Sun Oct 31 23:38:03 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	FSM
Top-level Entity Name	top
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	156
Total registers	97
Total pins	73
Total virtual pins	0
Total memory bits	80
Embedded Multiplier 9-bit elements	0
Total PLLs	1
UFM blocks	0
ADC blocks	0

Figure 4. Flow Summary

Once we had programmed our device with our design, we tested our device with different button presses and different numbers to accumulate by. Our button presses behaved as expected, and our combinations of different numbers behaved as expected. After pressing the store button 5 times the sum of the five values appeared on the six 7-segment displays. The reset button cleared out the values in the FIFO successfully and cleared the values represented on the six 7-segment displays. Appendix A shows our VHDL code for this lab.

Conclusion

We learned how to design and create two separate finite state machines that work together to accomplish a task. This lab is very important as building a design that has multiple clock domains is very relevant. Many projects in the real world have several clock domains. It's important to know how to work with them and how to send data back and forth across the borders.

Appendix A

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

entity top is
    port (
        ADC_CLK_10 : in std_logic;
        MAX10_CLK1_50 : in std_logic;
        MAX10_CLK2_50 : in std_logic;
        HEX0 : out std_logic_vector(7 downto 0);
        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);
        KEY : in std_logic_vector(1 downto 0);
        SW : in std_logic_vector(9 downto 0);
        LEDR : out std_logic_vector(9 downto 0)
    );
end top;

architecture behave of top is
    type state_type is (IDLE, STORE, HOLD, ACCUM);
    signal write_state : state_type;
    signal read_state : state_type;
    signal accumulator : std_logic_vector(23 downto 0);
    signal write_clk : std_logic;
    signal read_clk : std_logic;
    signal wr_en : std_logic;
    signal rd_en : std_logic;
    signal empty : std_logic;
    signal data : std_logic_vector(9 downto 0);
    signal num : std_logic_vector(2 downto 0);
    signal rstn_btn : std_logic;
    signal store_btn : std_logic;
begin

    WRITE_MACHINE : process (write_clk)
    begin
        store_btn <= not KEY(1);
        if rising_edge(write_clk) then
            case write_state is
                when IDLE =>
                    wr_en <= '0';
                    if store_btn = '1' then
                        write_state <= STORE;
```

```
                end if;
            when STORE =>
                wr_en <= '1';
                write_state <= HOLD;
            when HOLD =>
                wr_en <= '0';
                if store_btn = '0' then
                    write_state <= IDLE;
                end if;
            when others =>
                write_state <= IDLE;
        end case;
    end if;
end process;

READ_MACHINE : process (read_clk)
begin
    if rstn_btn = '0' then
        accumulator <= (others => '0');
    elsif rising_edge(read_clk) then
        case read_state is
            when IDLE =>
                rd_en <= '0';
                if num >= 5 then
                    rd_en <= '1';
                    read_state <= ACCUM;
                end if;
            when ACCUM =>
                if empty = '1' then
                    read_state <= IDLE;
                else
                    rd_en <= '1';
                    accumulator <= accumulator + data;
                end if;
            when others =>
                read_state <= IDLE;
        end case;
    end if;
end process;

FF0: entity work.FIFO(syn)
    port map (
        aclr => NOT rstn_btn,
        data => SW,
        rdclk   => read_clk,
        rdreq   => rd_en,
        wrclk   => write_clk,
        wrreq   => wr_en,
        q => data,
```

```
        rdempty => empty,
        rdusedw => num
    );

PL0: entity work.PLL(syn)
    port map    (
        inclk0 => MAX10_CLK1_50,
        c0  => write_clk,
        c1  => read_clk
    );

HX0: entity work.Seg_Decoder(rtl)
    port map (
        binary => accumulator(3 downto 0),
        dp => '0',
        hex => HEX0
    );

HX1: entity work.Seg_Decoder(rtl)
    port map (
        binary => accumulator(7 downto 4),
        dp => '0',
        hex => HEX1
    );

HX2: entity work.Seg_Decoder(rtl)
    port map (
        binary => accumulator(11 downto 8),
        dp => '0',
        hex => HEX2
    );

HX3: entity work.Seg_Decoder(rtl)
    port map (
        binary => accumulator(15 downto 12),
        dp => '0',
        hex => HEX3
    );

HX4: entity work.Seg_Decoder(rtl)
    port map (
        binary => accumulator(19 downto 16),
        dp => '0',
        hex => HEX4
    );

HX5: entity work.Seg_Decoder(rtl)
    port map (
        binary => accumulator(23 downto 20),
```

```
            dp => '0',
            hex => HEX5
        );

    LEDR <= SW;
    rstn_btn <= KEY(0);

end behave;
```

Appendix B

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

entity Seg_Decoder is
    port (
        binary : in std_logic_vector(3 downto 0);
        dp : in std_logic;
        hex : out std_logic_vector(7 downto 0)
    );
end Seg_Decoder;

architecture rtl of Seg_Decoder is

    -- decoder declaration
    type memory is array (0 to 15) of std_logic_vector(7 downto 0);
    constant seg_decoder : memory := (x"3F", x"06", x"5B", x"4F", x"66", x"6D", x"7D", x"07",
x"7F", x"67", -- 0-9
                                      x"77", x"7C", x"39", x"5E", x"79", x"71"); -- A-F
begin

    hex(6 downto 0) <= not seg_decoder(to_integer(unsigned(binary)))(6 downto 0);
    hex(7) <= not dp;

end rtl;
```