Project 4: Temperature Measurements: Clock Domains Crossing

Due: Monday 1st April, 2024

1 Introduction

Clock Domain Crossing is a common requirement of modern digital design. This is the transference of data from a flip-flop driven by one clock to another flip-flop driven by a different clock. Because the frequency of the clocks may be different and the clocks may be out of phase, proper synchronization is needed to avoid data corruption due to metastability at the input of the receiving flip-flop.

In this project, we explore FIFO synchronizers and use them to transfer data from one (slow) clock domain to a different (faster) clock domain.

2 Analog to Digital Converter And Temperature Sensor

The MAX10 FPGA in your board has a total of two analog to digital converters (ADCs). An ADC is a device which samples a voltage, then using a reference voltage returns a binary value at a certain resolution representing that value. The ADC in the MAX10 has a 12 bit resolution, meaning that it can divide the voltage range to the reference voltage into 2^{12} steps.

There are different types of ADCs. The ones present in the MAX10 FPGA are successive approximation (SAR) ADCs. A SAR ADC performs a binary search successively approximating the target voltage. As such, the ADC in the MAX10 will finish a conversion after at most 12 steps. In general an k bit SAR ADC will finish the conversion after at most k steps (there are 2^k possible values to search from, worst case scenario for binary search is $O(\log n)$ where n is the number of values to search from, therefore our worst case scenario is $O(\log 2^k) \equiv O(k)$.

We can directly instance an ADC in the MAX10 by using the `fiftyfivenm_adcblock` component megafunction from the `fiftyfivenm_components` package of the `wysiwyg` library. *This is an Intel/Altera proprietary package which maps directly to the FPGA fabric of the MAX10.* We provide the `max10_adc` entity as a wrapper to the first ADC in the MAX10 on your FPGA board. The entity's port map is described in Table 1.

The ADC as well as its control unit will lie in one of the two clock domains of your design. Use the 10 MHz clock to drive the ADC. This clock is connected to a PLL input, so make sure you utilize one. The PLL should be configured to output a 1 MHz clock. Use the output clock of the ADC to drive all logic related to this clock domain. You will need a control unit to drive the operation of the ADC, as well as to send data into the other clock domain. Henceforth this clock domain shall be named the *producer*.

Signal	Direction	Description
`pll_clk`	in	Input clock at 10 MHz
`chsel`	in	ADC channel select
`soc`	in	start of conversion, set to `1` to start conversion
`tsen`	in	temperature sensing mode if `1`
`dout`	out	data output
`eoc`	out	end of conversion, set to `1` when conversion has finished
`clk_dft`	out	clock output from clock divider

Table 1: Port map for the `max10_adc` entity.

3 Seven Segment Displays

A seven segment display module is a collection of LEDs in a form factor that resembles numbers. Figure 1 shows one of the traditional example encodings for a hexadecimal alphabet. Some encodings do not add "tails" to the *6* and the *9*, while other encodings render the *c* as a lowercase.

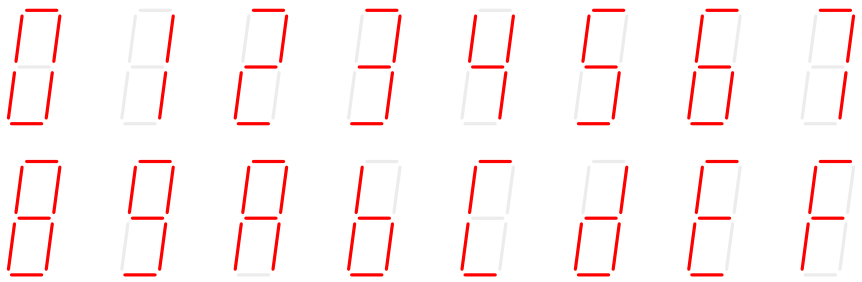

Figure 1: Seven segment display hexadecimal alphabet.

In the DE10-Lite, the seven segment displays are wired using a common anode configuration. Applying a logic high to a segment turns on the LED. Table 3-6 of the DE10-Lite User Manual describes the connection of the LEDs to the FPGA.

For the purposes of this assignment, the seven segment displays will be driven using the 50 MHz clock domain. Updates to the seven segment displays are to be done at this frequency. Henceforth this clock domain shall be named the *consumer*.

4 Clock Domains Crossing

Clock domain crossing is to be done using a FIFO synchronizer. The producer side drives the head of the FIFO synchronizer while the consumer side drives the tail of the synchronizer. This methodology is described in the paper *Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog* by Clifford E. Cummings. The paper contains SystemVerilog code for the implementation of the necessary hardware. The code may be used as a guide to construct the VHDL implementation.

We will use a two stage synchronizer to transfer data between one clock domain to the next. Because a two stage synchronizer can only act on a single bit, we will convert the index pointer of a clock domain into Gray code before transmitting it. The rationale for this conversion is that we are assured proper data transfer, as the Hamming distance between consecutive numbers when written in Gray code is 1. This ensures that at most one bit has changed when crossing clock domains, ensuring safe passage. The following entities may be used to perform binary to Gray code and Gray code to binary conversion.

Listing 1: Binary to Gray Conversion

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

entity bin_to_gray is
    generic (
        input_width:    positive := 16
    );
    port (
        bin_in:         in  std_logic_vector(input_width - 1 downto 0);
        gray_out:       out std_logic_vector(input_width - 1 downto 0)
    );
end entity bin_to_gray;

architecture rtl of bin_to_gray is
    signal shifted: std_logic_vector(bin_in'range);
begin
    shifted(input_width - 2 downto 0) <= bin_in(input_width - 1 downto 1);
    shifted(bin_in'high) <= '0';
    gray_out <= bin_in xor shifted;
end architecture rtl;
```

Listing 2: Gray to Binary Conversion

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

entity gray_to_bin is
    generic (
        input_width:    positive := 16
    );
    port (
        gray_in:        in  std_logic_vector(input_width - 1 downto 0);
        bin_out:        out std_logic_vector(input_width - 1 downto 0)
    );
end entity gray_to_bin;

architecture rtl of gray_to_bin is
    function unary_xor (
            vector: in  std_logic_vector
        ) return std_logic
    is
        variable ret: std_logic := '0';
    begin
        for i in vector'range loop
            ret := ret xor vector(i);
```

```
        end loop;
        return ret;
    end function unary_xor;
begin

    xor_tree: for i in bin_out'range generate
        -- if using VHDL-2008 then we just use the regular unary xor
        -- bin_out(i) <= xor gray_in(gray_in'high downto 0);
        -- since we do not have VHDL-2008 support we need an auxiliary function
        -- to do the unary XOR operation
        bin_out(i) <= unary_xor(gray_in(gray_in'high downto i));
    end generate xor_tree;

end architecture rtl;
```

Create a Synopsys Design Constraint file and perform timing analysis on your design. Ensure that your design is free of timing issues. A Synopsys Design Constraint file is a Tcl script with extension `.sdc` which defines the clocks of your design. You may use the following to get started.

```
% main 50 MHz clock
create_clock -period ??? [ get_ports ???? ]
create_clock -period ??? -name main_clock_virt

% ADC 10 MHz clock
create_clock -period ??? [ get_ports ???? ]
create_clock -period ??? -name adc_clock_virt

% ADC derived clock
create_generated_clock -name clk_div -source [ get_pins ??? ] \
    -divide_by ??? -multiply_by ??? [ get_pins ??? ]
```

For more information on this file, you may watch `https://www.youtube.com/watch?v=ggWxledaBFg` and/or consult the Quartus documentation.

5 Deliverables

The following items must be submitted.

- **(30pt)** A report summarizing your design, experimentation, and discussing your results. The report must include your hardware overhead of your design. You must also include a photograph of your board displaying the temperature. For graduate students, the report must be at least five (5) pages long and must contain a high level diagram of your design. Reports must be typeset as a single spaced, single column on a 12pt serif typeface (e.g. Computer Modern, or Times). Any code in the report must use a monospace typeface.
- **(20pt)** The finite state machine driving the ADC.
- **(20pt)** The decoder logic which converts a number into its decimal representation and shows it on the seven segment displays.

- **(30pt)** The VHDL code which performs the clock domain crossing. The Synopsys Design Constraint file used for timing analysis. Your report must discuss the results of this analysis.

Please submit a compressed version of your Quartus project alongside your report. You may choose to link to a repository containing your code instead.

5.1 Extra credit

For fifteen (15) extra points, implement the finite state machine that drives the ADC and sends temperatures to the FIFO synchronizer using microcode. You may choose your own microinstruction format as well as your sequencer implementation. If using this method, provide a description of your implementation in your report.

For ten (10) extra points, utilize the same sequencer and finite state machine to drive both clock domains. You will need to make two separate instances of this sequencer. Each instance should be given its microcode as a generic parameter.