# Lab 8 Report

Matthew Crump A01841001
Nicholas Williams A02057223

## Objectives

The objective of this lab is to implement an ADC and wire VHDL to interface with it. This lab will also require finding and reading documentation for the IP block.

## Procedure

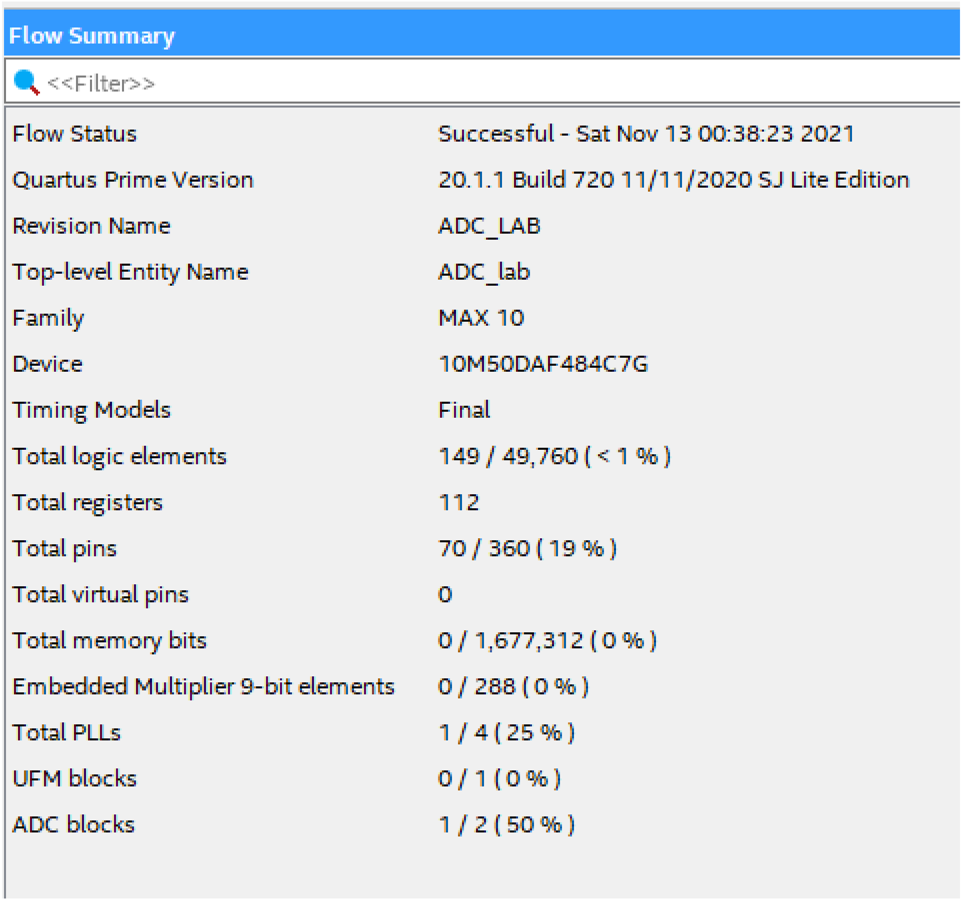
We started by reviewing our notes from the ADC lecture to figure out what settings we should use to generate the ADC IP. We determined we would use any of the analog inputs except for ADC_6 and ADC_7 as there are no headers for those. We configured the PLL to generate a clock of 10 MHz that then went into the ADC IP block. At first we were confused why this was needed but after reading the documentation for the ADC block, it's because the input clk can not be selected and only has one connection to the PLL clock output. We then referred to the documentation and the example projects and determined we could hard-code all of the command signals high.

We instantiated all 6 of the 7-segment displays, driving 3 of them off and the other 3 with data. We created a read process that would read in the ADC value whenever it was ready and save it in an intermediate register. From there, the intermediate value would be read into the display register every second. Pressing the reset button would reset the ADC block. We connected a potentiometer to ADC channel 0, connecting the other 2 pins to power and ground.

## Results

We successfully used the IP Catalog to create an ADC module. The hardest part of this lab was setting up the port map to connect our top module to the ADC module. We successfully created a PLL using the IP Catalog that was able to drive the ADC clock. We connected the lock signal of the ADC to the PLL. We also connected the reset button to the ADC.

Once we had all the signals connected inside of our design, we created two processes that were sensitive to the system clock. The first process simply updated a register with the response data every time the valid response signal was received. The second process counted to 10 million and then sent the adc data to the 7-segment displays. The second process would also pause the displayed value for as long as the reset button was pressed. We then compiled our design and had no problems. Figure 1 shows our design's flow summary.


Flow Summary

<<Filter>>

| | |
|---|---|
| Flow Status | Successful - Sat Nov 13 00:38:23 2021 |
| Quartus Prime Version | 20.1.1 Build 720 11/11/2020 SJ Lite Edition |
| Revision Name | ADC_LAB |
| Top-level Entity Name | ADC_lab |
| Family | MAX 10 |
| Device | 10M50DAF484C7G |
| Timing Models | Final |
| Total logic elements | 149 / 49,760 ( < 1 % ) |
| Total registers | 112 |
| Total pins | 70 / 360 ( 19 % ) |
| Total virtual pins | 0 |
| Total memory bits | 0 / 1,677,312 ( 0 % ) |
| Embedded Multiplier 9-bit elements | 0 / 288 ( 0 % ) |
| Total PLLs | 1 / 4 ( 25 % ) |
| UFM blocks | 0 / 1 ( 0 % ) |
| ADC blocks | 1 / 2 ( 50 % ) |

Figure 1. Flow Summary

We did not have a breadboard or a potentiometer the first time we programmed the DE10-lite board with our design. The display seemed to update every second, but the displayed value hovered around zero. The next day we were able to connect our FPGA to a potentiometer.

Figure 2 shows a DE10-lite board that is programmed with our final design. The red wire connects the input of the potentiometer to the 5V source on our breadboard. The green wire connects the potentiometer's ground to a ground pin on our DE10-lite board. The blue wire connects the output of the potentiometer to the input of our ADC. The displayed value changes as the potentiometer is adjusted with a screwdriver.

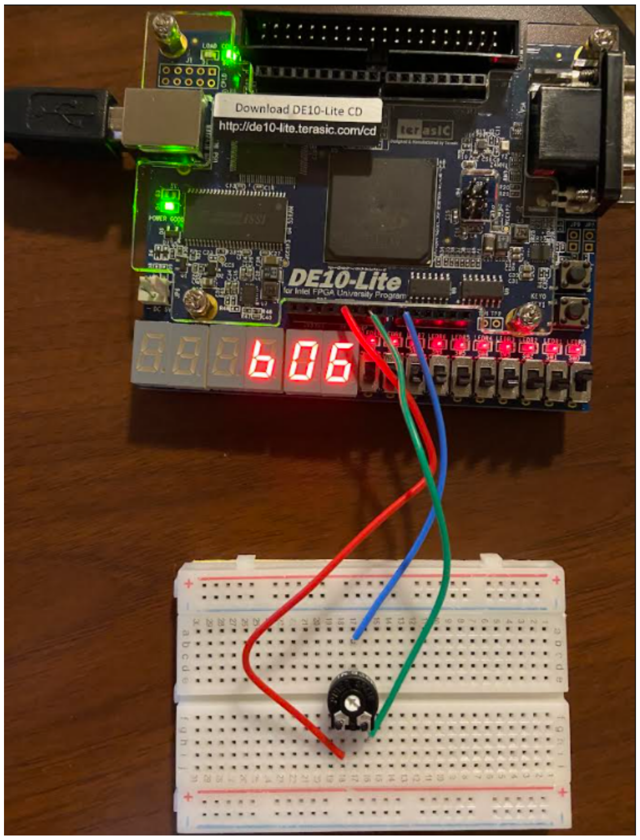


Figure 2. Finished Lab

## Conclusion

It’s great that the MAX10 FPGA includes an ADC, this opens up more possibilities when interfacing with sensors. This way no external ADC chip is needed, saving space on the board and allows for monitoring the package temperature of the FPGA. This lab also provided an opportunity to read documentation in the IP catalog. As well as, providing insight into how an ADC works and its limitations.

## Appendix A

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

entity ADC_lab is
    port (
        ADC_CLK_10 : in std_logic;
        MAX10_CLK1_50 : in std_logic;
        MAX10_CLK2_50 : in std_logic;
        ARDUINO_IO : inout std_logic_vector(15 downto 0);
        ARDUINO_RESET_N : inout std_logic;
        KEY : in std_logic_vector(1 downto 0);
        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)
    );
end ADC_lab;


architecture behave of ADC_lab is

    signal rstn_btn : std_logic;
    signal adc_clk : std_logic;
    signal lock : std_logic;
    signal adc_voltage : std_logic_vector(11 downto 0);
    signal cmd_valid : std_logic := '1';
    signal cmd_channel : std_logic_vector(4 downto 0) := "00001";
    signal cmd_start_pack : std_logic := '1';
    signal cmd_end_pack : std_logic := '1';
    signal cmd_ready : std_logic := '1';
    signal resp_valid : std_logic;
    signal resp_channel : std_logic_vector(4 downto 0);
    signal resp_data : std_logic_vector(11 downto 0);
    signal resp_start_pack : std_logic;
    signal resp_end_pack : std_logic;
    signal counter : natural;
    signal adc_voltage_reading : std_logic_vector(11 downto 0);

begin

p_sample : process (adc_clk)
    begin
        if rising_edge(adc_clk) then
            if resp_valid = '1' then
                adc_voltage_reading <= resp_data;
            end if;
        end if;
    end process;

p_one_hz : process (adc_clk)
begin
```

```
        if rising_edge(adc_clk) then
            if rstn_btn = '0' then
                adc_voltage <= adc_voltage;
            elsif counter >= (10000000 - 1) then
                counter <= 0;
                adc_voltage <= adc_voltage_reading;
            else
                counter <= counter + 1;
            end if;
        end if;
    end process;

    PL0: entity work.PLL(syn)
        port map (
            inclk0 => ADC_CLK_10,
            c0  => adc_clk,
            locked => lock
        );

    HX0: entity work.Seg_Decoder(rtl)
        port map (
            en => '1',
            binary => adc_voltage(3 downto 0),
            dp => '0',
            hex => HEX0
        );

    HX1: entity work.Seg_Decoder(rtl)
        port map (
            en => '1',
            binary => adc_voltage(7 downto 4),
            dp => '0',
            hex => HEX1
        );

    HX2: entity work.Seg_Decoder(rtl)
        port map (
            en => '1',
            binary => adc_voltage(11 downto 8),
            dp => '0',
            hex => HEX2
        );

    HX3: entity work.Seg_Decoder(rtl)
        port map (
            en => '0',
            binary => "0000",
            dp => '0',
            hex => HEX3
        );

    HX4: entity work.Seg_Decoder(rtl)
        port map (
            en => '0',
            binary => "0000",
            dp => '0',
            hex => HEX4
        );

    HX5: entity work.Seg_Decoder(rtl)
```

```vhdl
    port map (
        en => '0',
        binary => "0000",
        dp => '0',
        hex => HEX5
    );

    rstn_btn <= KEY(0);

adc_0 : entity work.adc(rtl)
    port map (
        clock_clk              => ADC_CLK_10,
        reset_sink_reset_n     => rstn_btn,
        adc_pll_clock_clk      => adc_clk,
        adc_pll_locked_export  => lock,
        command_valid          => cmd_valid,
        command_channel        => cmd_channel,
        command_startofpacket  => cmd_start_pack,
        command_endofpacket    => cmd_end_pack,
        command_ready          => cmd_ready,
        response_valid         => resp_valid,
        response_channel       => resp_channel,
        response_data          => resp_data,
        response_startofpacket => resp_start_pack,
        response_endofpacket   => resp_end_pack
    );

end behave;
```