Configurable Space Microsystems Innovations & Applications Center

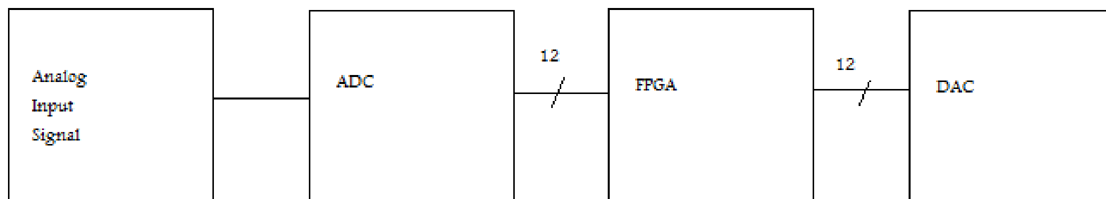
Tutorial 15 - ADC's and DAC's on the Spartan 3E Starter Board

Introduction

Analog to Digital converters, and their counterparts, Digital to Analog converters are used all the time in electronics. Indeed, they provide the only method by which one may interface a digital system with the real world, which functions in analog.
In this lab, for the Spartan 3E starter kit, we will be using two twelve bit chips, the PMOD-AD1 and the PMOD-DA2, which house the ADCS7476MSPS and the DAC121S101 chips respectively. We will take in an analog input signal, perform a simple transform (multiply and add by a constant), and output it again as a new signal. The two chips listed serialize their data (so they can fit in small form factors) and can be clocked to a maximum of 20MHz. Both chips require that data be available by the rising edge of the clock, and specify that they will present data on the falling edge of the same clock cycle.

Objective

To implement a ADC-DAC function in the FPGA.



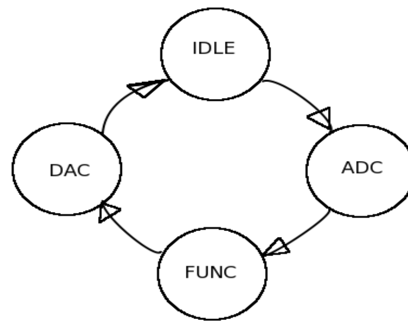
Process

1. Create a clock divider, and clock the rest of the circuit off of it's rising edge.
2. Design a state machine to loop through the process of reading from the ADC and writing to the DAC.
3. Implement ADC and DAC functionality by populating the states of the FSM.
4. Test the system with various functions and an oscilloscope.

Implementation

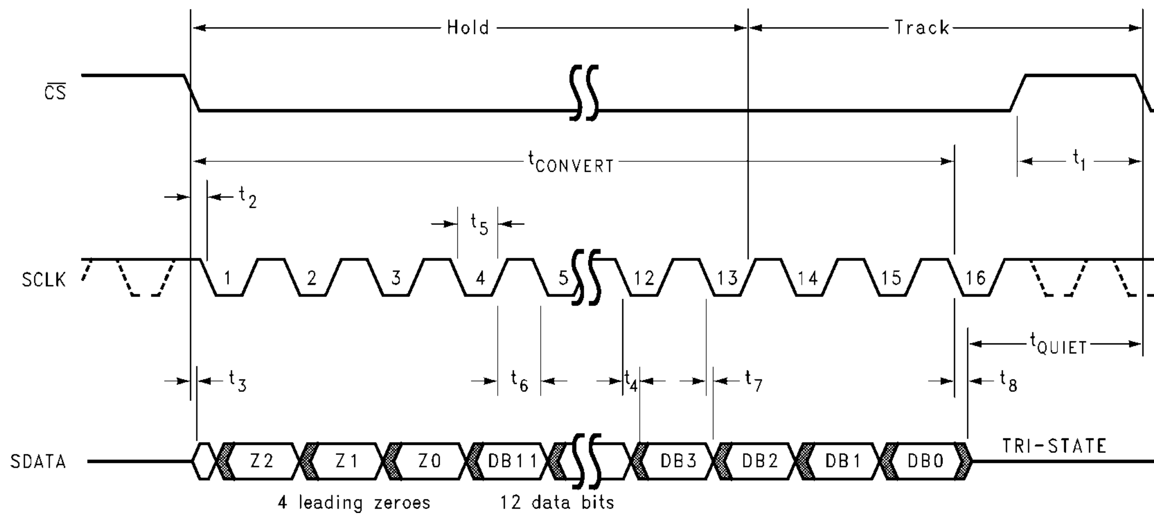
Finite State Machine

1. Use the following four states in succession: idle, read_adc, function, write_dac. This FSM scheme is suggested because it is simple and effective. The idle state serves to set up pins and timing signals before the system is running in earnest, the read_adc state reads the 12 bit signal from the ADC, the function state performs a simple transform on the number and the DAC is then updated to output the result of this transform.



ADC

1. The ADC follows a simple format, where, after pulling CS low, four zeroes are transmitted, following by a twelve bit number. This twelve bit number describes the voltage acquired by the circuit. The data is clocked with the MSB first.



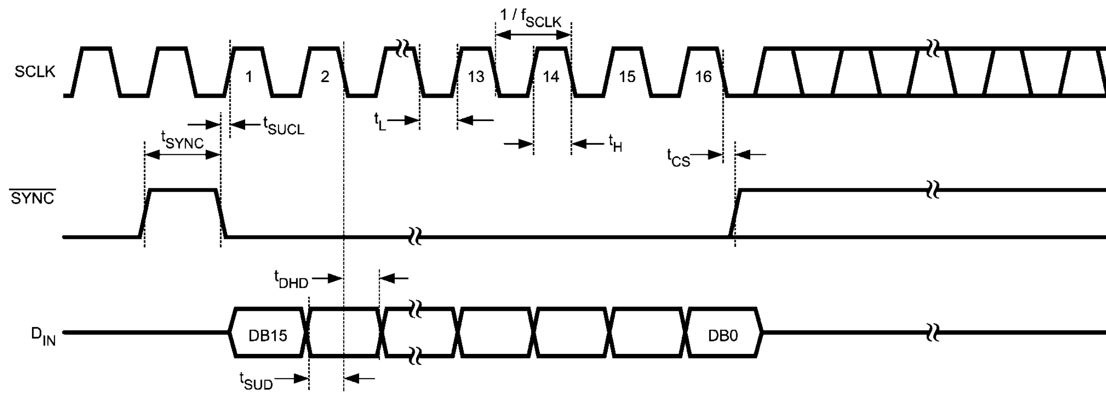
(from datasheet)

Function

1. In this example we simply multiply the twelve bit number by an arbitrary constant, however in principle one can do anything to it.

DAC

1. The DAC follows a similarly simple data scheme, whereby after pulling sync high (it can be left high for the duration of the procedure) 16 bits are written to the circuit. The first two need to be the zeroes, the next two are configuration bits, make certain they are zero as well. The last twelve bits describe the voltage that the DAC will output. In this case it will be the number obtained from the function step.

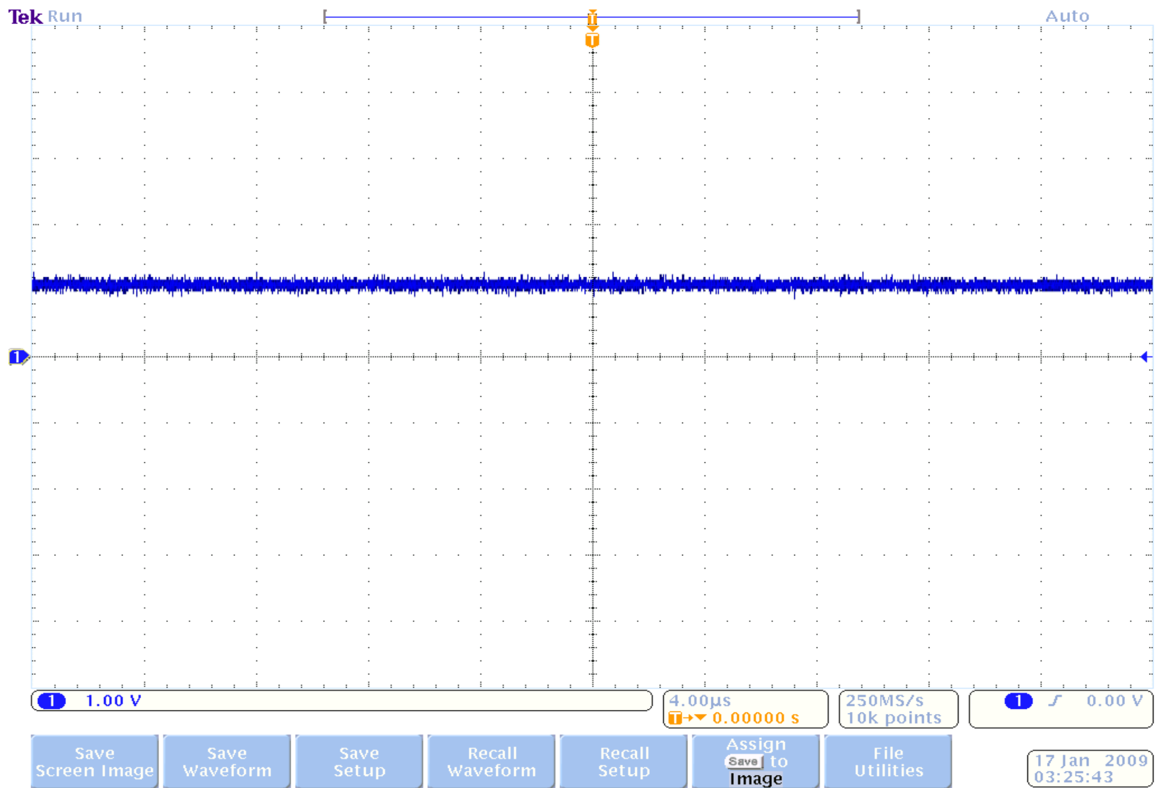


(from datasheet)

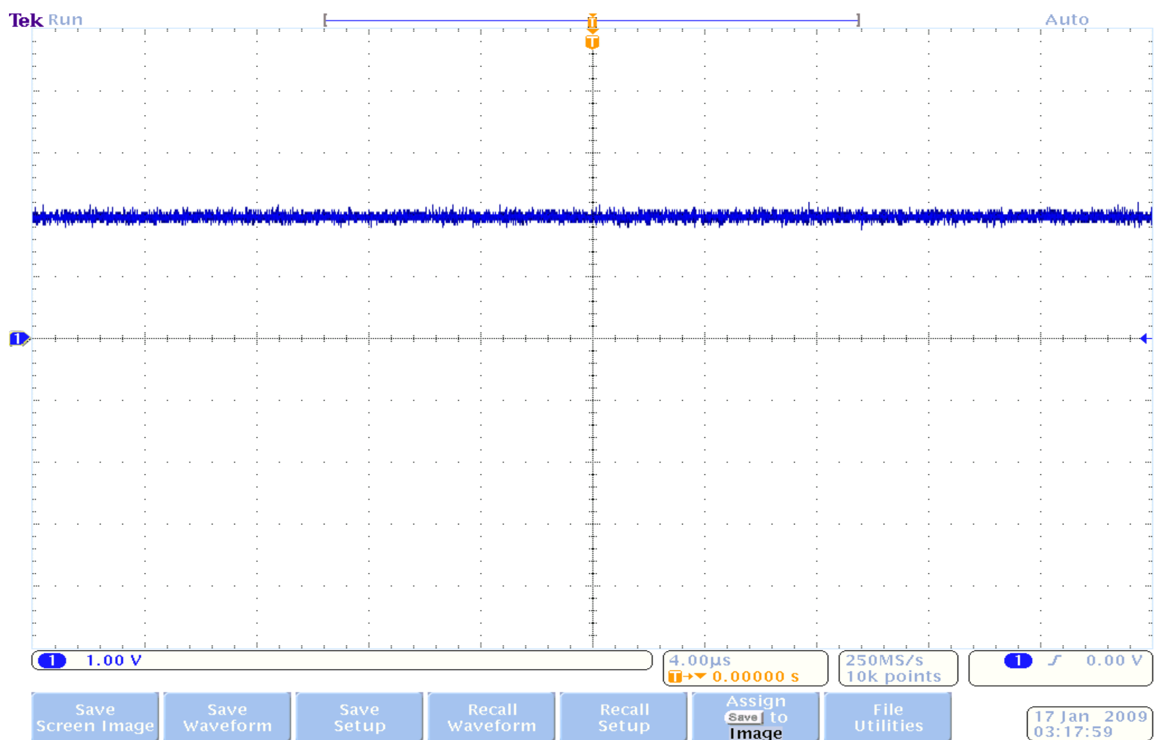
Verification

If one simply changes the multiplier to a large number, one will be able to note the large difference in gain simply by looking at the output voltage through an oscilloscope. In this case, as the input is a simple potentiometer, fewer turns of the screw will effect a greater change in output voltage. Similarly, if the multiplier is less than one, it will take more turns of the screw to make a change in the output voltage. For example, with a gain of 2x, picture 1, will turn into picture 2.

Picture 1: Input signal.



Picture 2: Results of the input signal being amplified by 2x.



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity motor is --constrained in ucf file
    Port (
      Clk      : in  std_logic; --spartan 3 e

      CS       : out std_logic;
             SYNC         : out std_logic;
      din    : in  std_logic; --adc
      dout    : out std_logic; --dac
      SClk     : out std_logic;  --adc
      SClk2     : out std_logic); --dac
end motor;

architecture Behavioral of motor is

type state_type is (idle, read, func, write);
signal state : state_type := read;

signal data  : unsigned(11 downto 0);
signal cnt  : integer range 0 to 20 := 0;
signal clkdiv : integer range 0 to 6;
signal newclk : std_logic := '0';
signal risingedge : std_logic := '1';

signal reset : std_logic := '0';

begin

       --drive the adc and dac clock pins
       SClk <= newclk;
       SClk2 <= newclk;


       clock_divider: process(clk, reset)
       begin
              if(reset = '1') then
              elsif(rising_edge(clk)) then
                     if(clkdiv = 5) then
                            risingedge <= risingedge xor '1';
                            newclk <= newclk xor '1';
                            clkdiv <= 0;
                     else
                            clkdiv <= clkdiv + 1;
                     end if;
              end if;
       end process clock_divider;

  main: process(clk, reset)
       variable temp : integer;
   begin
              if(reset = '1') then
              elsif(rising_edge(clk)) then
              if(clkdiv = 5 and risingedge = '1') then
                     case state is

                            when idle =>
                                   CS <= '1';
                                   SYNC <= '1';
                                   if(cnt = 15) then
```

```
                                        cnt <= 0;
                                        state <= read;
                                else
                                        cnt <= cnt + 1;
                                        state <= idle;
                                end if;

                        when read =>
                                CS <= '0';
                                SYNC <= '1';
                                cnt <= cnt + 1;
                                if(cnt < 4) then
                                        cnt <= cnt + 1;
                                        state <= read;
                                elsif(cnt > 3 and cnt < 16) then
                                        cnt <= cnt + 1;
                                        data(15 - cnt) <= din;
                                        state <= read;
                                elsif(cnt = 16) then
                                        cnt <= 0;
                                        state <= func;
                                end if;

                        when func =>
                                CS <= '1';
                                SYNC <= '1';
                                cnt <= 0;

                                temp := conv_integer(data);
                                temp := temp*2; --y=m*x+b
                                data <= conv_unsigned(temp,12);
                                state <= write;

                        when write =>
                                CS <= '1';
                                SYNC <= '0';

                                if(cnt = 0 or cnt = 1) then
                                        cnt <= cnt + 1;
                                        dout <= '0';
                                        state <= write;
                                elsif(cnt = 2 or cnt = 3) then
                                        cnt <= cnt + 1;
                                        dout <= '0';
                                        state <= write;
                                elsif(cnt > 3 and cnt < 16) then

                                        cnt <= cnt + 1;
                                        dout <= data(15 - cnt);
                                        state <= write;
                                elsif(cnt = 16) then
                                        cnt <= 0;
                                        state <= idle;
                                end if;

                        end case;
                end if;
        end if;
   end process main;

end Behavioral;
```