CpE 487 Digital Design Lab

Lab 5: DAC Siren

1. Introduction

In this lab, we will program the FPGA on the Nexys2 board to generate a wailing audio siren using a digital to analog converter (DAC). This will require us to format our digital audio sequence into a serial stream that can be used to drive an outboard DAC.

2. DAC Interface

We will be using a 16-bit stereo Digital to Analog Converter module *PmodI^{2}S* which plugs into one of the PMOD connectors on the *Nexys2* board. The module and its connections are shown in Figure 1. The module uses a Cirrus CS4344 Stereo DAC chip. Documentation on the module and the chip can be found on the course website.

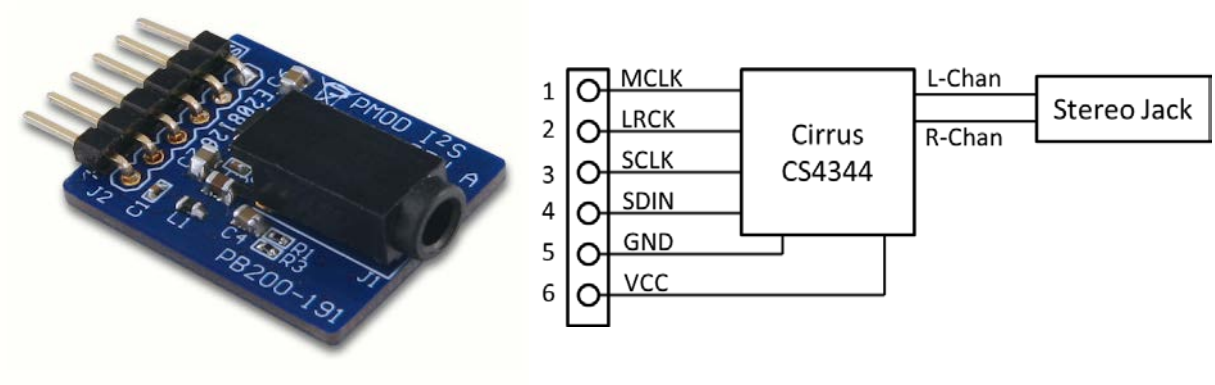

Figure 1 PModI^{2}S 16-bit Stereo DAC Module

The DAC converts "left" and "right" 16-bit digital data streams into two analog left and right channel signals which can be used to drive a stereo speaker system. In order to minimize the number of I/O pins on the chip and the module, data is presented to the DAC in serial format. The DAC employs an oversampled Delta-Sigma Modulator to convert the digital data into analog format. The FPGA supplies three clocks to the DAC:

LRCK: This is the audio sampling clock – it sets the audio sampling rate. We will be using an audio sampling rate of 48,800 samples/second. When LRCK is low, left channel data is transferred into the DAC. When LRCK is high, right channel data is transferred.

SCLK: This is the serial clock used to transfer data into the DAC. Data is transferred on the rising edge of SCLK. Because there are two channels, each requiring 16-bits of data, SCLK runs at 32x the sampling clock. We will be using an SCLK frequency of 1.56 MHz.

MCLK: This is a high speed clock that is used to drive the on-chip oversampled delta-sigma modulator. It can be set anywhere from 500kHz to 50 MHz. In this lab, we will be using an oversampling ratio of 256, which means that MCLK must be 256x the sampling clock. We will be using an MCLK frequency of 12.5 MHz.

Figure 2 shows the required timing relationship between LRCK, SCLK and the serial data. The CS4344 uses a serial data standard known as I^2S. Note that data is transferred MSBit first with the MSBit of each channel being transferred on the second rising edge of SCLK after LRCK changes from one channel to the other.

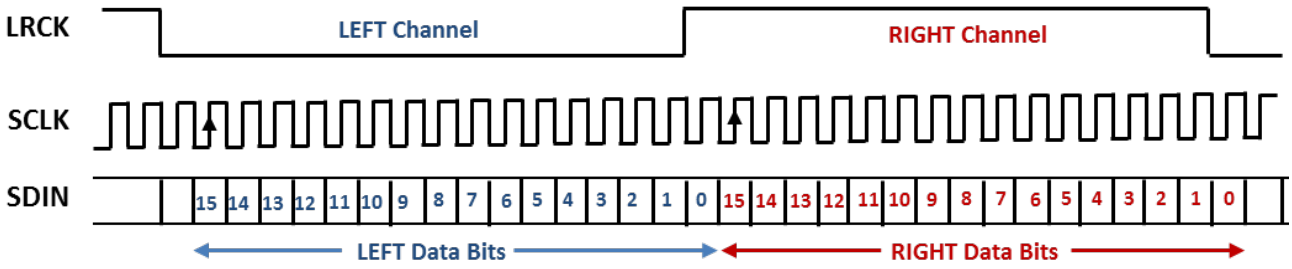

Figure 2 CS4344 I^2S Data Format

3. Hardware Setup

Plug the *PmodI^2S* module into PMOD jack *JA1* on the *Nexys2* boards as shown in Figure 3. Note that the *PmodI^2S* has only 6 pins whereas the jack JA1 has 12 sockets (2 rows of 6). Plug the module into the top row of sockets on JA1. Please be careful not to bend the pins on the *PmodI^2S* module. Now connect the audio cable from your stereo speaker system into the audio jack on the *PmodI^2S* module.

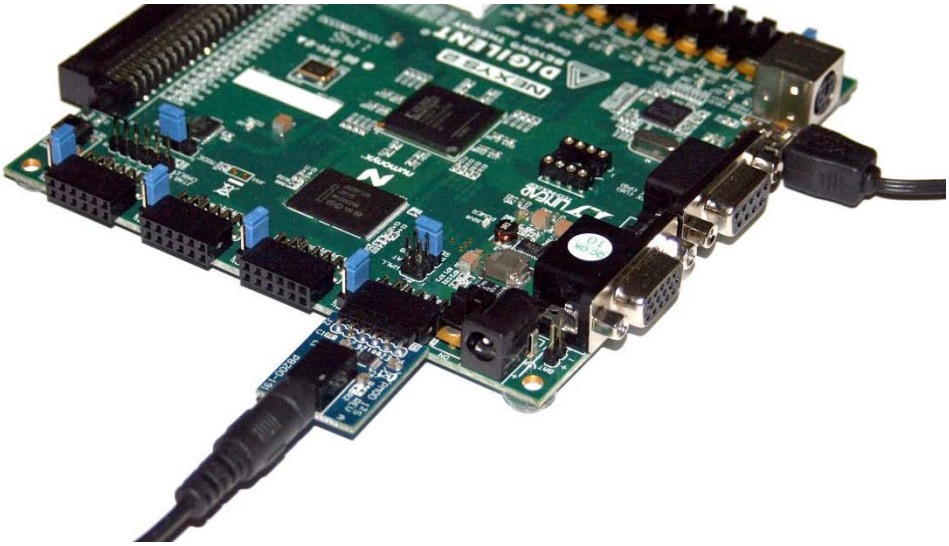

Figure 3 PmodI^2L Module inserted in PMOD jack JA1

4. Configuring the FPGA

4.1 Create a New Project

Use the Xilinx ISE software to create a new project named *Siren* using the same *project settings* as in Labs 1 and 2.

4.2 Add Source for "dac_if"

Create a new VHDL source module called *dac_if* and load the following source code into the edit window. Expand the **Synthesize** command in the *Process* window and run **Check Syntax** to verify that you have entered the code correctly.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity dac_if is
   Port (       SCLK : in  STD_LOGIC;                -- serial clock (1.56 MHz)
                L_start: in STD_LOGIC;               -- strobe to load LEFT data
                R_start: in STD_LOGIC;               -- strobe to load RIGHT data
                L_data : in SIGNED (15 downto 0);    -- LEFT data (15-bit signed)
                R_data : in SIGNED (15 downto 0);    -- RIGHT data (15-bit signed)
                SDATA : out  STD_LOGIC);             -- serial data stream to DAC
end dac_if;

architecture Behavioral of dac_if  is

signal sreg: STD_LOGIC_VECTOR (15 downto 0);     -- 16-bit shift register to do
                                                 -- parallel to serial conversion
begin

        -- SREG is used to serially shift data out to DAC, MSBit first.
        -- Left data is loaded into SREG on falling edge of SCLK when L_start is active.
        -- Right data is loaded into SREG on falling edge of SCLK when R_start is active.
        -- At other times, falling edge of SCLK causes REG to logically shift one bit left
        -- Serial data to DAC is MSBit of SREG
dac_proc: process
        begin
                wait until falling_edge(SCLK);
                if L_start = '1' then
                        sreg <= std_logic_vector (L_data);      -- load LEFT data into SREG
                elsif R_start = '1' then
                        sreg <= std_logic_vector (R_data);      -- load RIGHT data into SREG
                else sreg <= sreg(14 downto 0) & '0';           -- logically shift SREG one bit left
                end if;
```

```
        end process;

        SDATA <= sreg(15);                          -- serial data to DAC is MSBit of SREG

end Behavioral;
```

This module takes 16-bit parallel stereo data and converts it to the serial format required by the digital to analog converter. When *L_start* is high, a 16-bit left channel data word is loaded into the 16-bit serial shift register *SREG* on the falling edge of *SCLK*. When *L_start* goes low, SCLK shifts the data out of SREG, MSBit first to the serial output SDATA at a rate of 1.56 Mb/s. Simlarly, when *R_start* goes high, right channel data is loaded into SREG and then shifted out to SDATA. Output data changes on the falling edge of *SCLK*, so that it is stable when the DAC is reading the data on the rising edge of *SCLK*.

4.2 Add Source for "tone"

Create a new VHDL source module called *tone* and load the following source code into the edit window. Expand the **Synthesize** command in the *Process* window and run **Check Syntax** to verify that you have entered the code correctly.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

        -- Generates a 16-bit signed triangle wave sequence at a sampling rate determined
        -- by input clk and with a frequency of (clk*pitch)/65,536
entity tone is
        Port (  clk : in STD_LOGIC;                 -- 48.8 kHz audio sampling clock
                pitch : in UNSIGNED (13 downto 0);  -- frequency (in units of 0.745 Hz)
                data : out SIGNED (15 downto 0));   -- signed triangle wave out
end tone;

architecture Behavioral of tone is

signal count: unsigned (15 downto 0);               -- represents current phase of waveform
signal quad: std_logic_vector (1 downto 0);         -- current quadrant of phase
signal index: signed (15 downto 0);                 -- index into current quadrant

begin

        -- This process adds "pitch" to the current phase every sampling period. Generates
        -- an unsigned 16-bit sawtooth waveform. Frequency is determined by pitch. For
        -- example when pitch=1, then frequency will be 0.745 Hz. When pitch=16,384, frequency
        -- will be 12.2 kHz.
cnt_pr: process
        begin
                wait until rising_edge(clk);
```

```
        count <= count + pitch;
end process;

quad <= std_logic_vector (count (15 downto 14));        -- splits count range into 4 phases
index <= signed ("00" & count (13 downto 0));           -- 14-bit index into the current phase

-- This select statement converts an unsigned 16-bit sawtooth that ranges from 65,535
-- into a signed 12-bit triangle wave that ranges from -16,383 to +16,383

with quad select
data <=         index when "00",                        -- 1st quadrant
                        16383 - index when "01",        -- 2nd quadrant
                        0 - index when "10",            -- 3rd quadrant
                        index - 16383 when others;      -- 4th quadrant

end Behavioral;
```

This module generates a signed triangular wave (a tone) at a sampling rate of 48.8 KHz. The frequency of the tone is determined by the input *pitch*. The process *cnt_pr* generates an unsigned sawtooth waveform *count* by incrementing a 16-bit counter *pitch* values every clock cycle. The frequency with which it traverses the whole 16-bit count range is thus proportional to *pitch*. The lowest possible tone frequency is obtained when *pitch*=1. It then takes 2^{16}=65,536 cycles to traverse the range of the counter. The frequency is then 48.8kHz / 2^{16} ≈ 0.745 Hz. If *pitch* is set to 1000, the frequency would be 1000*0.745 Hz ≈745 Hz.

A select signal assignment statement is then used to convert the unsigned sawtooth count into a signed triangle wave. The sawtooth *count* is split up into 4 quadrants *quad* and an *index* value within the quadrant. The signals *quad* and *index* are used to generate a triangle wave as shown in Figure 4.

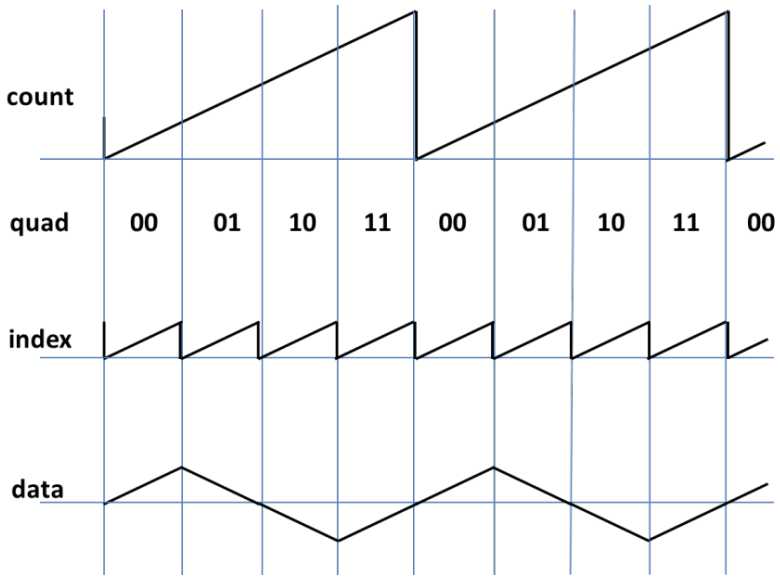

Figure 4 Conversion from 16-it unsigned sawtooth to 16-bit signed triangle waveform

4.3 Add Source for "wail"

Create a new VHDL source module called *wail* and load the following source code into the edit window. Expand the **Synthesize** command in the *Process* window and run **Check Syntax** to verify that you have entered the code correctly.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

        -- Generates a "wailing siren" sound by instancing a "tone" module and modulating
        -- the pitch of the tone. The pitch is increased until it reaches hi_pitch and then
        -- decreased until it reaches lo_pitch and then increased again...etc.
entity wail is
   Port ( lo_pitch : in  UNSIGNED (13 downto 0);        -- lowest pitch (in units of 0.745 Hz)
          hi_pitch : in  UNSIGNED (13 downto 0);        -- highest pitch (in units of 0.745 Hz)
          wspeed : in UNSIGNED (7 downto 0);            -- speed of wail in pitch units/wclk
          wclk : in  STD_LOGIC;                         -- wailing clock (47.6 Hz)
          audio_clk : in STD_LOGIC;                     -- audio sampling clock (48.8 kHz)
          audio_data : out  SIGNED (15 downto 0));      -- output audio sequence (wailing tone)
end wail;

architecture Behavioral of wail is

component tone is
        Port (    clk : in STD_LOGIC;
                  pitch : in UNSIGNED (13 downto 0);
                  data : out SIGNED (15 downto 0));
end component;

signal curr_pitch: UNSIGNED (13 downto 0);              -- current wailing pitch

begin

        -- this process modulates the current pitch. It keep a variable updn to indicate
        -- whether tome is currently rising or falling. Each wclk period it increments
        -- (or decrements) the current pitch by wspeed. When it reaches hi_pitch, it
        -- starts counting down. When it reaches lo_pitch, it starts counting up
wp: process
        variable updn: std_logic;
        begin
                wait until rising_edge(wclk);
                if curr_pitch >= hi_pitch then updn :='0';          -- check to see if still in range
                elsif curr_pitch <= lo_pitch then updn := '1';      -- if not, adjust updn
                end if;
                if updn = '1' then curr_pitch <= curr_pitch + wspeed;     -- modulate pitch according to
                else curr_pitch <= curr_pitch - wspeed;                   -- current value of updn
                end if;
        end process;
```

```
tgen: tone port map(    clk => audio_clk,              -- instance a tone module
                        pitch => curr_pitch,           -- use curr-pitch to modulate tone
                        data => audio_data);

end Behavioral;
```

This module creates an instance of the module tone and then modulates the pitch up and down to produce a “wailing” siren. The inputs *hi_pitch* and *lo_pitch* define the upper and lower limits of the generated tone. The inputs *wspeed* and *wclk* determine how fast the pitch changes.

The wailing tone is generated by the process *wp*. This process is run on the rising edge of *wclk*. This is a slow clock (approx. 48 Hz.) Every *wclk* cycle, the current pitch is raised or lowered depending on the value of *updn*. When *updn*=’1’, the pitch rises. When *updn*=’0’, the pitch lowers. The input *wspeed* determines how much the *pitch* changes every *wclk* cycle. When the current pitch exceeds *hi_pitch*, *updn* is set to ‘0’ so that the pitch will start decreasing. Similarly, when the current pitch is lower than *lo_pitch*, *updn* is set to ‘1’ to start the tone rising again.

4.4 Add Source for top level “siren”

Create a new VHDL source module called *siren* and load the following source code into the edit window. Expand the **Synthesize** command in the *Process* window and run **Check Syntax** to verify that you have entered the code correctly.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity siren is
   Port ( clk_50MHz : in  STD_LOGIC;          -- system clock (50 MHz)
          dac_MCLK : out  STD_LOGIC;          -- outputs to PMODI2L DAC
          dac_LRCK : out  STD_LOGIC;
          dac_SCLK : out  STD_LOGIC;
          dac_SDIN : out  STD_LOGIC);
end siren;


architecture Behavioral of siren is

constant lo_tone: UNSIGNED (13 downto 0) := to_unsigned (344, 14);   -- lower limit of siren = 256 Hz
constant hi_tone: UNSIGNED (13 downto 0) := to_unsigned (687, 14);   -- upper limit of siren = 512 Hz
constant wail_speed: UNSIGNED (7 downto 0) := to_unsigned (8, 8);    -- sets wailing speed

component dac_if is
   Port ( SCLK : in  STD_LOGIC;
          L_start: in STD_LOGIC;
```

```
        R_start: in STD_LOGIC;
        L_data : in signed (15 downto 0);
        R_data : in signed (15 downto 0);
        SDATA : out  STD_LOGIC);
end component;

component wail is
   Port ( lo_pitch : in  UNSIGNED (13 downto 0);
        hi_pitch : in  UNSIGNED (13 downto 0);
        wspeed : in UNSIGNED (7 downto 0);
        wclk : in  STD_LOGIC;
        audio_clk : in STD_LOGIC;
        audio_data : out  SIGNED (15 downto 0));
end component;

signal tcount: unsigned (19 downto 0) := (others=>'0');   -- timing counter
signal data_L, data_R: SIGNED (15 downto 0);              -- 16-bit signed audio data
signal dac_load_L, dac_load_R: STD_LOGIC;                 -- timing pulses to load DAC shift reg.
signal slo_clk, sclk, audio_CLK: STD_LOGIC;

begin

        -- this process sets up a 20 bit binary counter clocked at 50MHz. This is used
        -- to generate all necessary timing signals. dac_load_L and dac_load_R are pulses
        -- sent to dac_if to load parallel data into shift register for serial clocking
        -- out to DAC
tim_pr: process
        begin
                wait until rising_edge(clk_50MHz);
                if (tcount(9 downto 0) >= X"00F") and (tcount(9 downto 0) < X"02E") then
                        dac_load_L <= '1'; else dac_load_L <= '0';
                end if;
                if (tcount(9 downto 0) >= X"20F") and (tcount(9 downto 0) < X"22E") then
                        dac_load_R <= '1'; else dac_load_R <= '0';
                end if;
                tcount <= tcount+1;
        end process;

        dac_MCLK <= not tcount(1);              -- DAC master clock (12.5 MHz)
        audio_CLK <= tcount(9);                 -- audio sampling rate (48.8 kHz)
        dac_LRCK <= audio_CLK;                  -- also sent to DAC as left/right clock
        sclk <= tcount(4);                      -- serial data clock (1.56 MHz)
        dac_SCLK <= sclk;                       -- also sent to DAC as SCLK
        slo_clk <= tcount(19);                  -- clock to control wailing of tone (47.6 Hz)

dac: dac_if port map (  SCLK => sclk,                   -- instantiate parallel to serial DAC interface
                        L_start => dac_load_L,
                        R_start => dac_load_R,
                        L_data => data_L,
                        R_data => data_R,
                        SDATA => dac_SDIN );
```

```
w1: wail port map(      lo_pitch => lo_tone,            -- instantiate wailing siren
                        hi_pitch => hi_tone,
                        wspeed => wail_speed,
                        wclk => slo_clk,
                        audio_clk => audio_clk,
                        audio_data => data_L);

        data_R <= data_L;       -- duplicate data on right channel

end Behavioral;
```

This is the top level that hooks it all together. The constants *lo_tone*, *hi_tone* and *wail_speed* define the parameters of the siren. The 20-bit timing counter *tcount* is used to generate all the necessary timing signals. The module *wail* is instanced to generate the 16-bit signed audio sequences *data_L* and *data_R* (the same data is sent to both channels). These sequences are then sent to an instance of *dac_if* to convert them to the required serial stream. Primary outputs of this top level module go directly to the DAC.

4.5 Synthesis and Implementation

Highlight the *siren* module in the *Hierarchy* window and execute the **Synthesis** command in the *Process* window.

Add an Implementation Constraint source file *siren.ucf* and enter the following data into the edit window:

```
NET "clk_50MHZ" LOC = B8;

NET "dac_SDIN" LOC = M15;
NET "dac_SCLK" LOC = L17;
NET "dac_LRCK" LOC = K12;
NET "dac_MCLK" LOC = L15;

NET "clk_50MHZ" TNM_NET = ck_50_net;
TIMESPEC TS_ck_50 = PERIOD "ck_50_net" 20 ns HIGH 50%;
```

Now highlight the *siren* module in the Hierarchy window and run **Implement Design** followed by **Generate Programming File** (don't forget to change the *FPGA Start-up Clock* to be the *JTAG Clock*).

4.6 Download and Run

Use the *Adept* software to download your configuration file *siren.bit* and check out the result.

4.7 Now let's make some changes …

Modify your VHDL code to do the following:

(a) Change the upper and lower pitch limits and also the wailing speed
(b) Modify the tone module to create a square wave instead of a triangle wave when the first push button (BTN0) is depressed. You will need to add this push button as an input to the top level *siren* module and pass its value down to the *tone* module. You can get the correct pin number for this push button from the *Nexys2 Reference Manual*. Note the difference in the quality of the tone when you switch to a square wave tone.
(c) Use the eight slide switches (SW0-SW7) on the Nexys2 board to set the wailing speed. You will need to add these as inputs to the top-level *siren* module. You can get the correct pin numbers for these switches from the *Nexys2 Reference Manual*.
(d) Try adding a second wail instance to drive the right audio channel. Use different high and low tone limits and wailing speed for the right channel.