**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.

1. **DAC Interface**

|  |
| --- |
|  |
| **Figure 1 PModI2S 16-bit Stereo DAC Module** |

We will be using a 16-bit stereo Digital to Analog Converter module *PmodI2S* 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.

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 I2S. 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 I2S Data Format** |

1. **Hardware Setup**

Plug the *PmodI2S* module into PMOD jack *JA1* on the *Nexys2* boards as shown in Figure 3. Note that the *PmodI2S* 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 *PmodI2S* module. Now connect the audio cable from your stereo speaker system into the audio jack on the *PmodI2S* module.

|  |
| --- |
|  |
| **Figure 3 PmodI2L Module inserted in PMOD jack JA1** |

1. **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 216=65,536 cycles to traverse the range of the counter. The frequency is then 48.8kHz / 216 ≈ 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:

1. Change the upper and lower pitch limits and also the wailing speed
2. 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.
3. 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*.
4. 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.