



SAN DIEGO STATE
UNIVERSITY

Implementing Filters on FPGAs

Department of Electrical and Computer Engineering

Real-Time DSP and FPGA Development Lab

Mark S. Manalo and Ashkan Ashrafi

Table of Contents

Introduction.....	3
Analog to Digital Conversion.....	3
Adding a DC Offset.....	6
Designing a Lowpass Digital Filter in MATLAB.....	9
Implementing a Filter on an FPGA.....	12
Least Mean Square Adaptive Filter on FPGA.....	16
Appendix.....	21
Analog to Digital Conversion Code.....	21
MATLAB Filter Code.....	24
Digital Filter Code.....	26
Adaptive Filter HDL.....	30
References.....	36

Introduction

This paper explains the process of designing a digital filter in MATLAB, converting it to HDL using HDL Coder and programming it onto an FPGA. The reader should have experience with creating a project in Xilinx ISE, implementing the project and programming it onto an FPGA, MATLAB/Simulink, and basic digital filter concepts. The reader should also have a working knowledge of the VHDL language and concepts such as component instantiation. The software version used are as follows:

Xilinx ISE 13.2 (64-bit), MATLAB (R2011a) with DSP toolbox and Simulink HDL Coder.

Analog to Digital Conversion

The ADC/DAC that we will be using is the PMODAD1 and PMODDA2 modules from Digilent. The PMODAD1 uses the Analog Devices AD7476 12-bit, A/D Converter. The PMODDA2 uses the National Semiconductor CAD121S101, 12-bit D/A Converter. These devices are controlled using the Serial Peripheral Interface (SPI). For a tutorial and example VHDL code on using these modules on the Spartan-3E starter board see: http://www.cosmiac.org/tutorial_15.html

First start ISE 13.2:

First create a new directory and call it 'filter_ML506'

Then start Xilinx ISE and create a new project for the Virtex5 ML506 board called 'lowpass_ml506' and save it under the 'filter_ML506' directory.

Create a new VHDL module and name it 'adc_dac'.

Create a new project and copy the the code located in Appendix: Analog to Digital Conversion Code for controlling the ADC/DAC. The code simply receives an analog signal, digitizes it with the ADC and then sends it to the DAC which recreates the analog signal. (note: This code is from the tutorial located at http://www.cosmiac.org/tutorial_15.html, but modified for use with the Virtex5 ML506 board).

Next create an Implementation Constraints File (.ucf), name it 'ml506' and copy this code (note: we are using the XGI Expansion Headeres, J6 Header of the ML506 to interface with the PmodAD1/PmodDA2):

```
# ML506 100 MHz clock pin  
NET "CLK"          LOC = "AH15";
```

```
# Pmod AD1 ADC Pins on J6 Header  
NET "CS"          LOC = "H33";  
NET "DIN"         LOC = "F34";  
NET "SCLK"        LOC = "G33";
```

```
# Pmod DA2 DAC Pins on J6 Header
NET "SYNC"      LOC = "J34";
NET "DOUT"      LOC = "L33";
NET "SCLK2"     LOC = "P34";
```

Your project workspace should now look something like (figure 1):

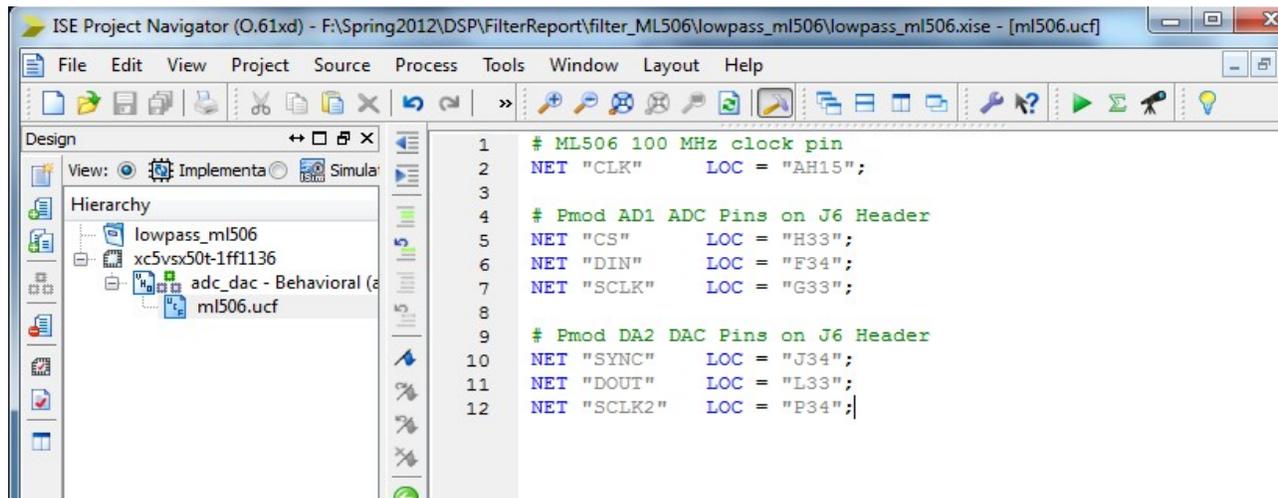


Figure 1: .ucf file

Now connect the pmods to the ML506 J6 headers, making sure that every thing is connected properly. The J6 Header also supplies the power and ground pins of the Pmods. (Figure 2) is a picture of the setup:

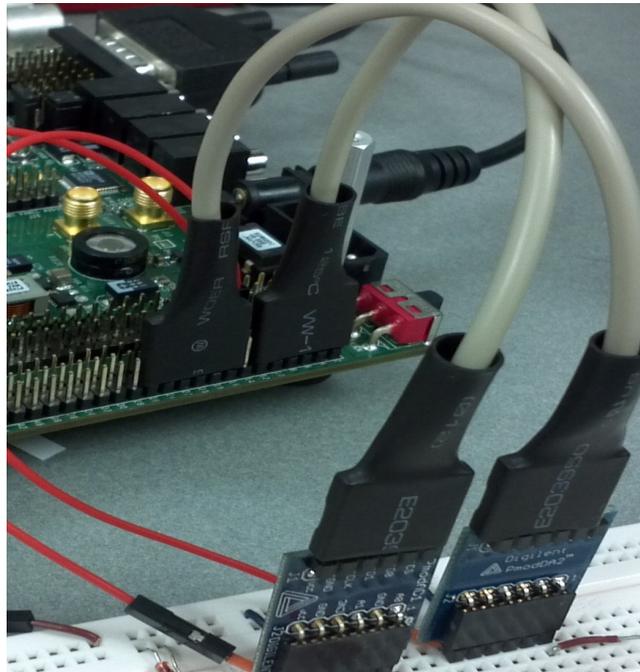
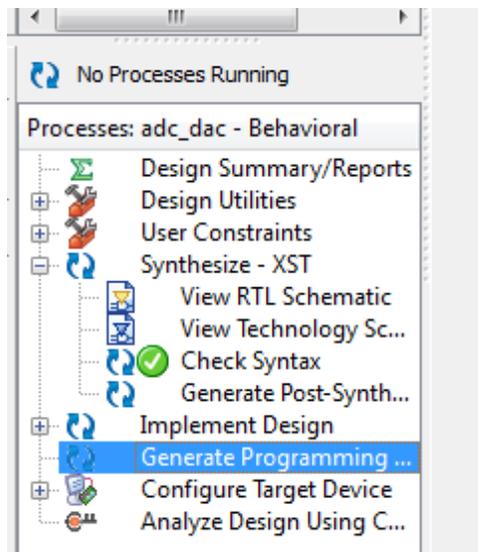


Figure 2: ML506 J6 Header

After everything is connected, generate a programming file in Xilinx ISE by clicking on 'Generate Programming File' in the process window:



Turn on the board and program the FPGA with iMPACT or any other programming tool of choice.

To test this code a function generator is used to produce a 500Hz sine wave with 0.5Vpp and high-z output. This sine wave is then fed into the A0 pin of the ADC. The signal gets converted to digital and then gets converted back to analog. As you can see in (Figure 3), the output seems to be rectified. This is because the ADC only accepts voltages between 0-3.3V and the DAC can only output 0-3.3V, but the

output of the function generator is between -0.5V and 0.5V.

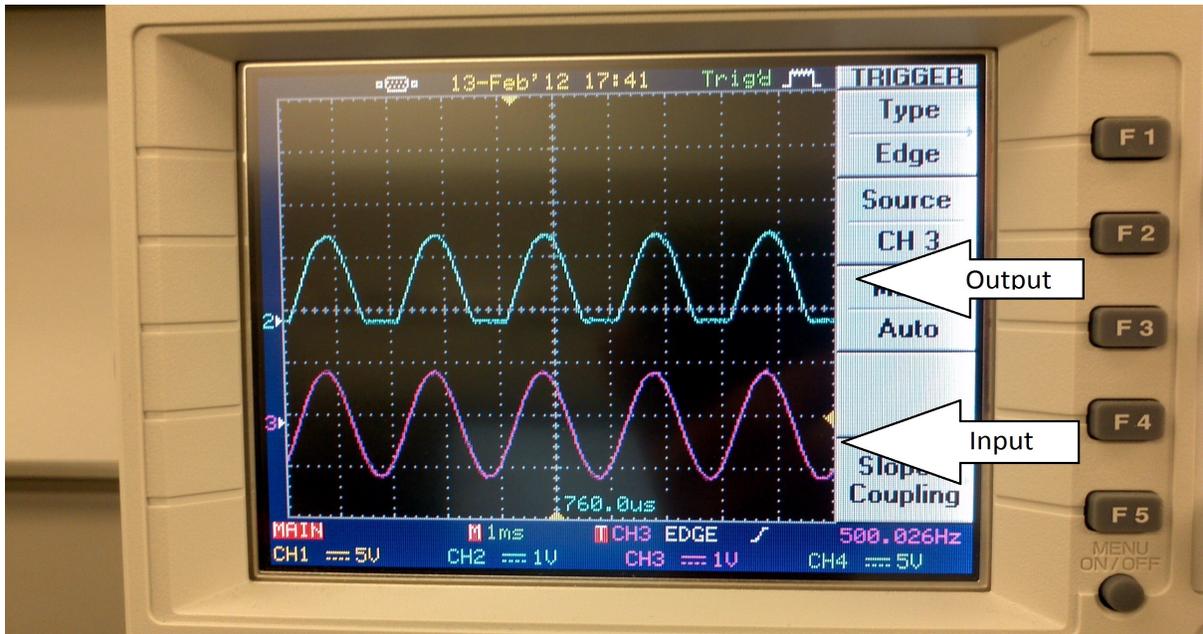


Figure 3: Rectified Signal

Thus, we need to add a DC offset to our input signal so we can properly digitize it. This can easily be done by changing the function generator settings to automatically add a DC offset. But if we are using an audio signal, we don't have the luxury of doing this. Instead, we must build a circuit which can add a DC offset to our signal. This is discussed in the next section.

Adding a DC Offset

The ADC/DAC modules we used only supports voltages of 0-3.3V. However the voltage coming from a typical audio signal -0.5 to 0.5V. So before sending this signal into the ADC we must first add a DC offset to the input signal. Since our input voltage is 0.5Vpp we chose a DC offset of 1.5V. The circuit that does this is shown in (Figure 4)

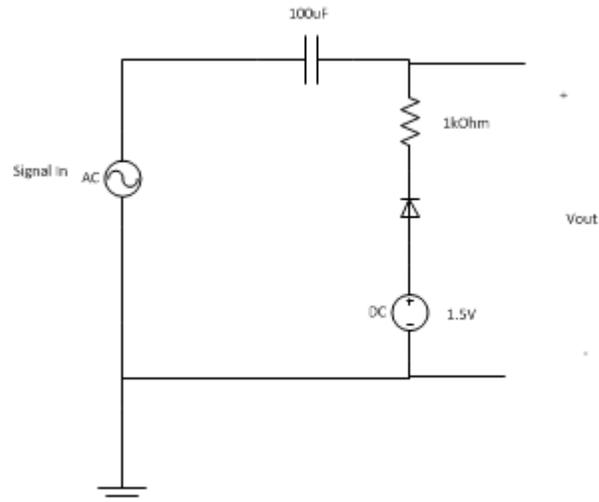


Figure 4: DC Offset

Here is the circuit implemented on a breadboard (note that the circuit shown has two inputs and both the inputs are added a DC offset, you only need one input at the moment, the second input is used for the adaptive filter discussed later):

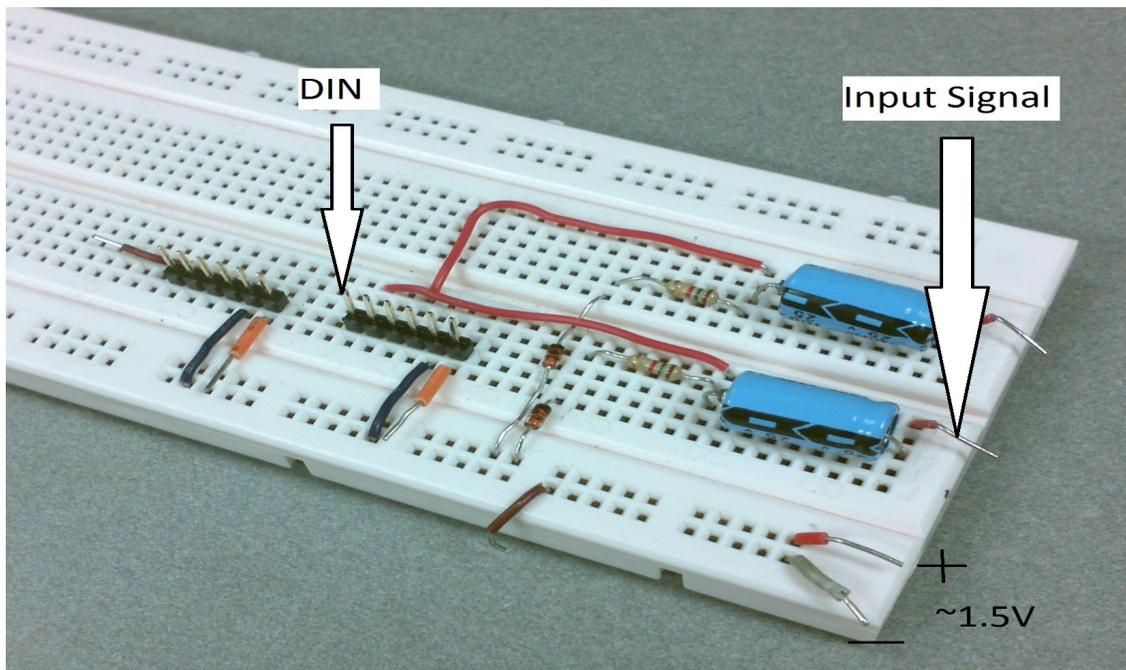


Figure 5: DC Offset Circuit on Breadboard

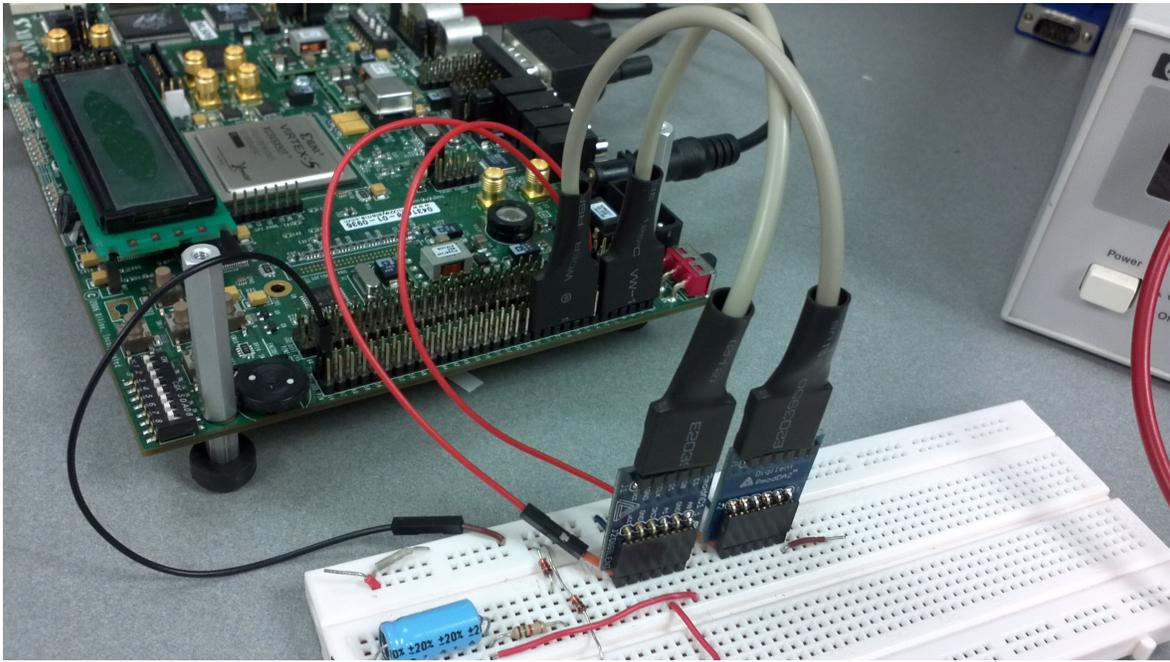


Figure 6: DC Offset circuit Connect to ADC/DAC chips

Now when we apply our signal with the DC offset we get no rectification as shown below:

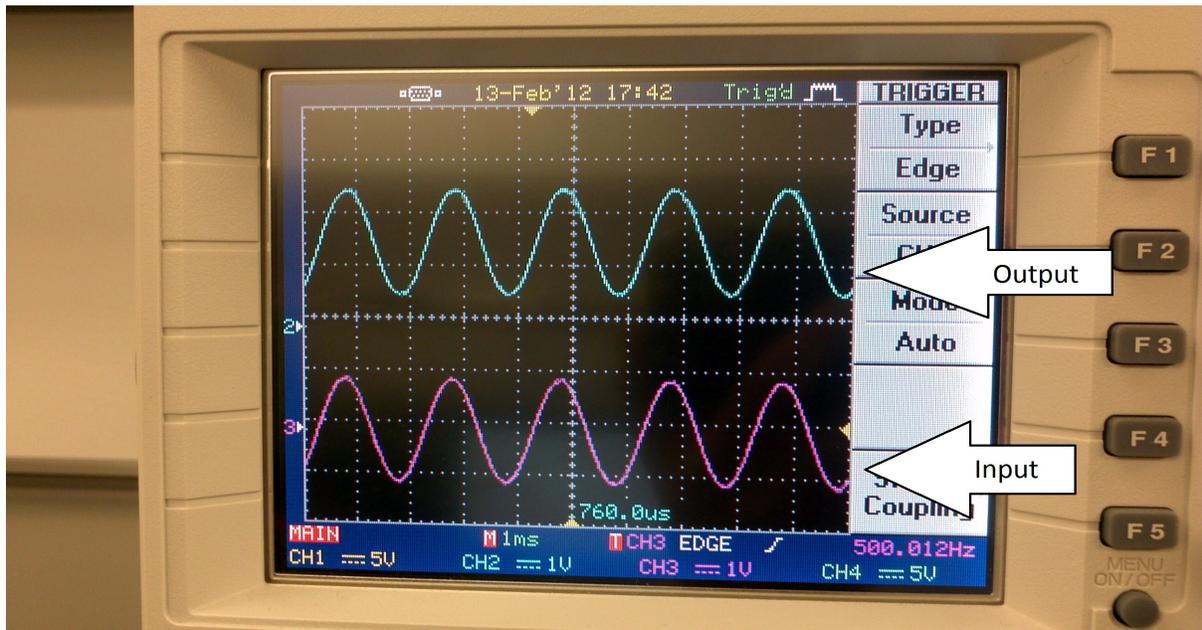


Figure 7: Signal with no rectification

Designing a Lowpass Digital Filter in MATLAB

After converting the analog signal to a digital signal, several DSP techniques can be used to manipulate the input signal. In this case we will implement a lowpass digital filter.

First we will create a lowpass filter using MATLAB. To facilitate this process we will use the MATLAB demo 'HDL Butterworth Filter'.

(To find this demo, start MATLAB and click help → Product Help. In the search bar type: 'HDL Butterworth Filter')

In MATLAB create a new script file as shown in (Figure 8):

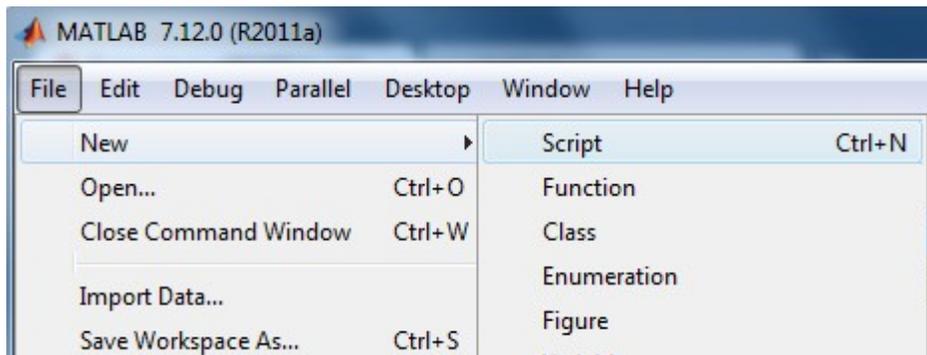


Figure 8: Create a new MATLAB script file

Then Click File → save as.. and name the file 'my_lowpass.m' in a directory called 'lowpass_MATLAB': as shown in (Figure 9)

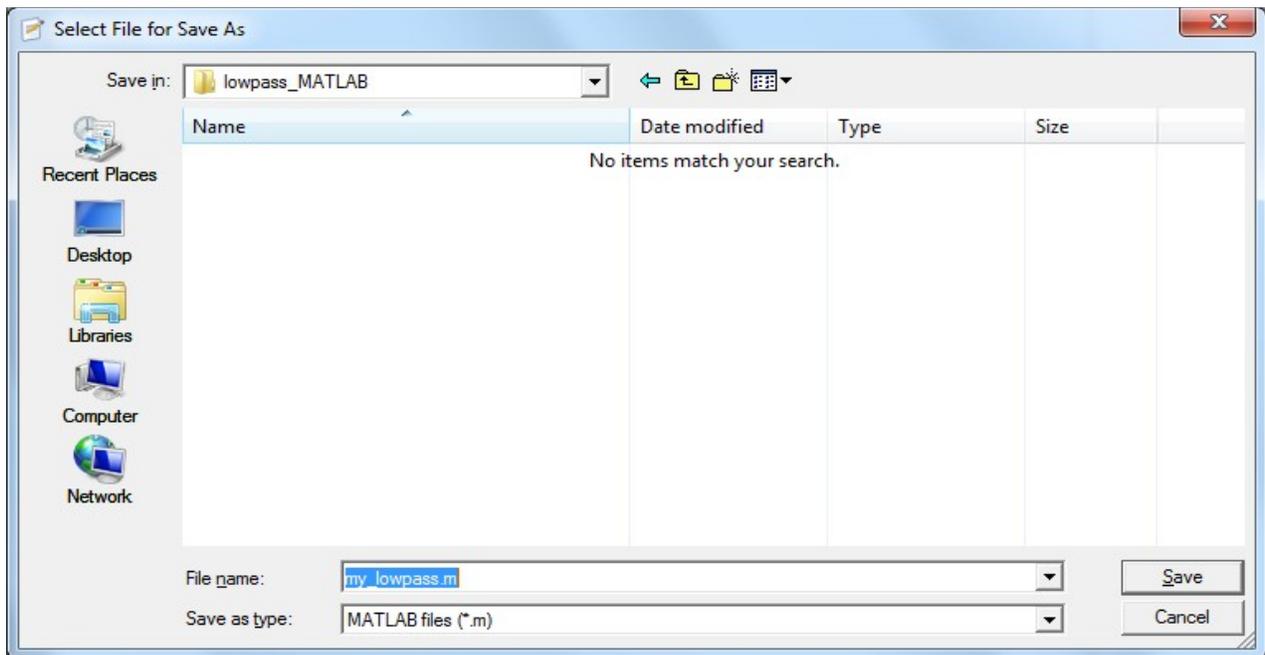


Figure 9: New M file for lowpass filter

Then copy-paste the code from (Appendix: MATLAB Filter Code) a slightly modified version of the butterworth example (This will create a lowpass filter with a sampling frequency of 52 kHz and a cutoff frequency of 1000 Hz, with 12 bit input/output widths and 11 fractional length):

Now click on the green play arrow and click 'Change Folder' if it asks:

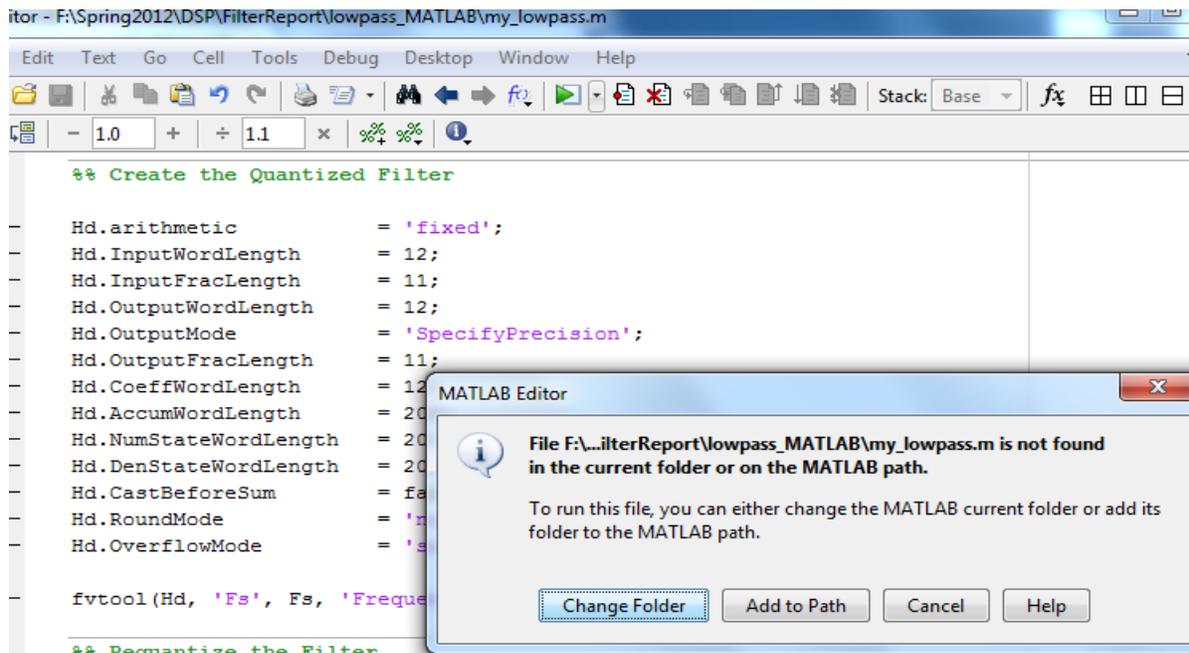


Figure 10: MATLAB Filter

Then MATLAB should create a testbench and a VHDL module for the filter we specified as shown in (figure 11):

```

### Starting VHDL code generation process for filter: hdlbutter
### Starting VHDL code generation process for filter: hdlbutter
### Generating: F:\Spring2012\DSP\FilterReport\lowpass_MATLAB\hdl_work\hdlb
### Starting generation of hdlbutter VHDL entity
### Starting generation of hdlbutter VHDL architecture
### First-order section, # 1
### Second-order section, # 2
### Second-order section, # 3
### HDL latency is 2 samples
### Successful completion of VHDL code generation process for filter: hdlbu

Warning: GENERATETB function is
deprecated. Please use GENERATEHDL
with 'GenerateHDLTestbench'
property set to 'on' instead.
> In dfilt.basefilter.generatetb at 48
   In my_lowpass at 87
### Starting generation of VHDL Test Bench
### Generating input stimulus
### Done generating input stimulus; length 3045 samples.
### Generating Test bench: F:\Spring2012\DSP\FilterReport\lowpass_MATLAB\hd
### Please wait ...Done
### Done generating VHDL Test Bench
>>

```

Figure 11: MATLAB Creates HDL

Now if we look into the 'lowpass_MATLAB' directory we created earlier, we should have a new directory called 'hdl_work', this is shown in (Figure 12). In this directory is the VHDL module and testbench for our lowpass filter. In the next section we will use the VHDL file that MATLAB created and instantiate it in our ADC/DAC module.

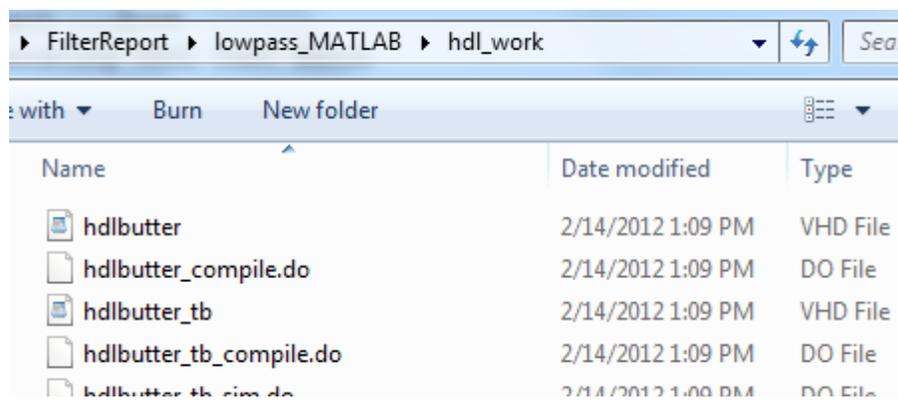


Figure 12: HDL Files

Implementing a Filter on an FPGA

After MATLAB creates the HDL files, we will instantiate them into our ADC/DAC circuit.

Go back to Xilinx ISE and right-click the project and click 'add source':

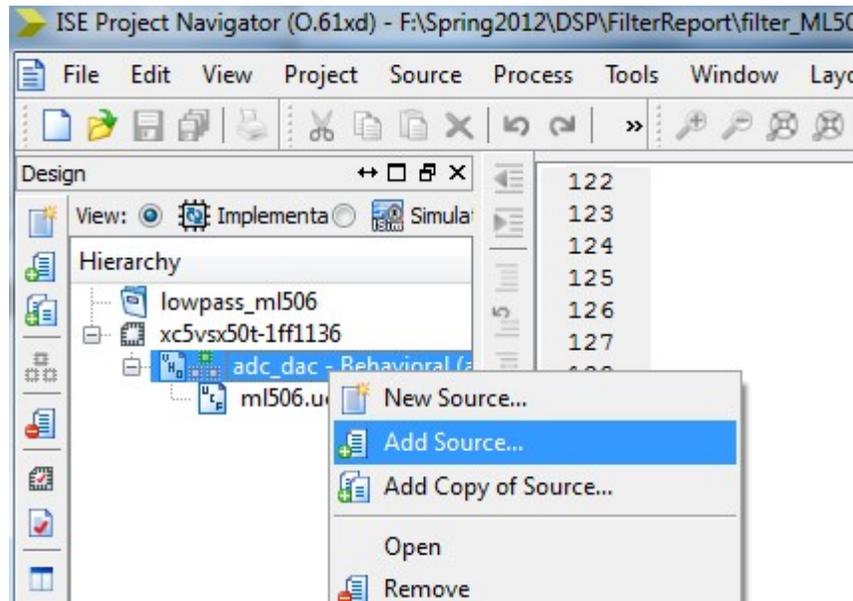


Figure 13: New source

Navigate to the hdlbutter.vhd file that MATLAB created and add it:

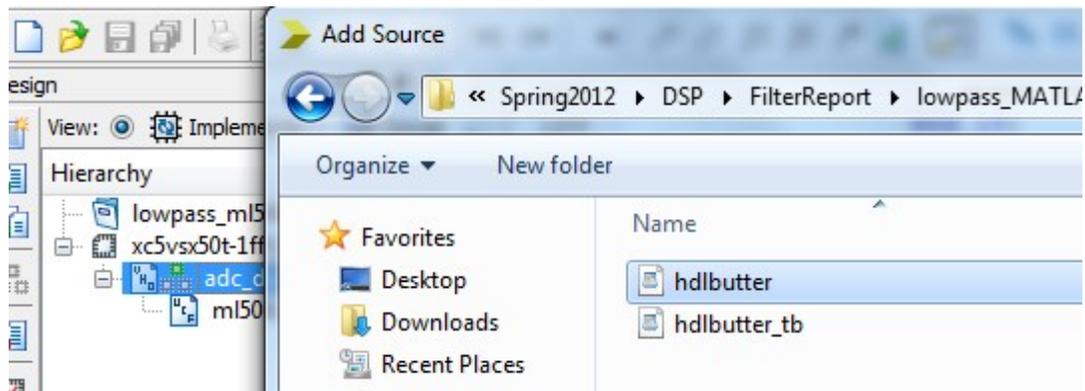


Figure 14: Adding HDL file to project

Now we must create a clock for the filter using DCM. Right-click the project and click 'New Source'. Click 'IP (CORE Generator & Architecture Wizard)' and name it 'clk_div':

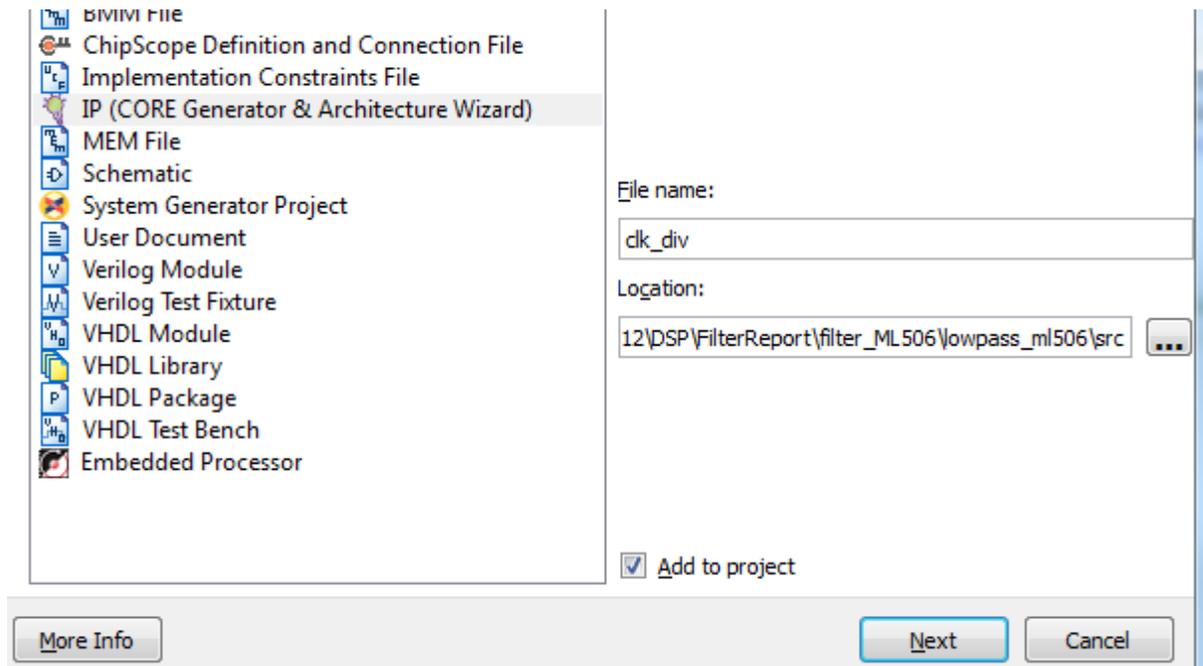


Figure 15: Xilinx IP Core

In the IP selection Wizard, navigate to FPGA Features and Design → Clocking → Virtex-5 → Choose wizard by basic function and click on 'Double clock frequency (DCM)' and click Next and Finish (this is shown in the following page on Figure 16):

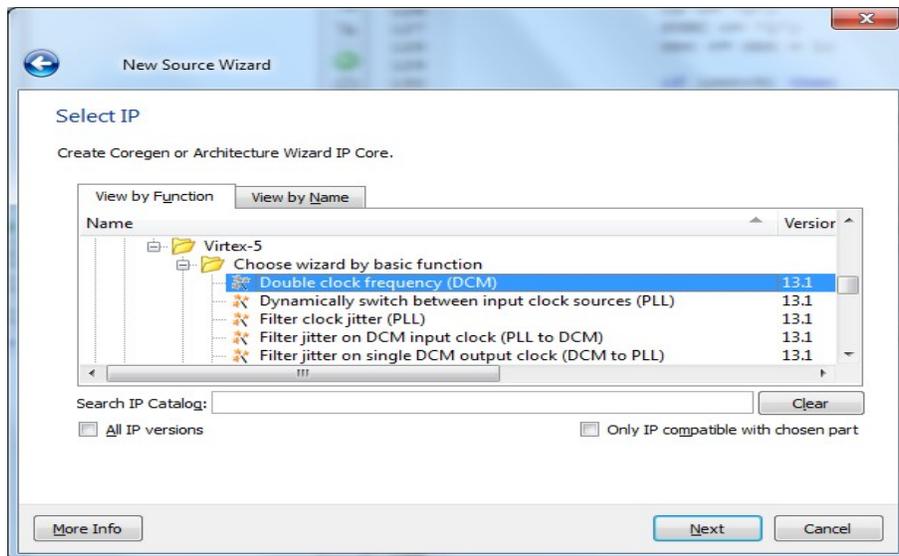


Figure 16: Add DCM

In the next window choose the following options in figure 17 and click OK:

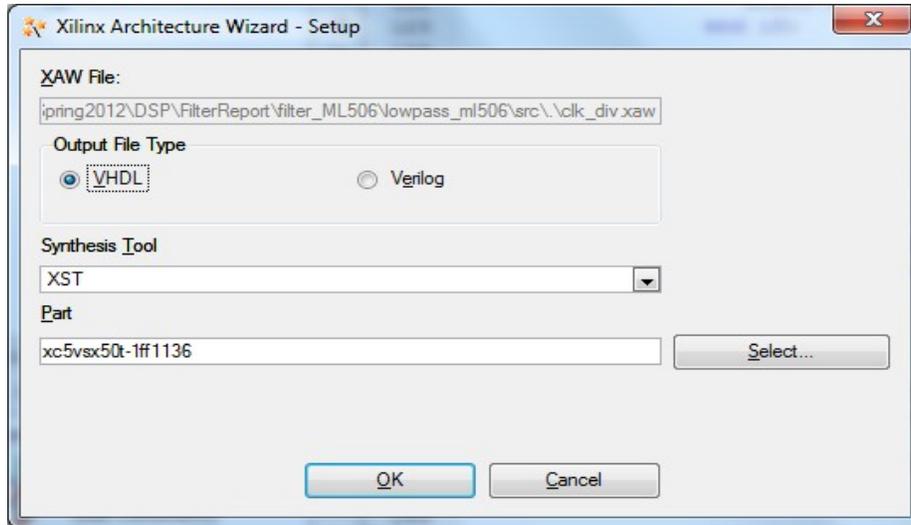


Figure 17: Choose VHDL as the file type

We will divide the 100MHz ML506 clock by 20 to get 5 MHz. Choose the following options shown in figure 18 :

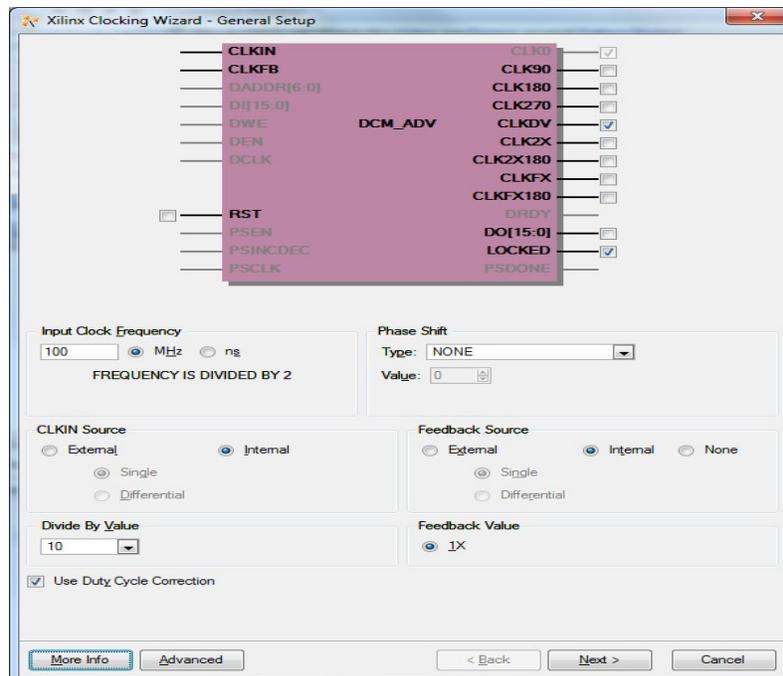


Figure 18: DCM Settings

Click the 'Advanced' button and check the 'Divide Input Clock By 2' option:

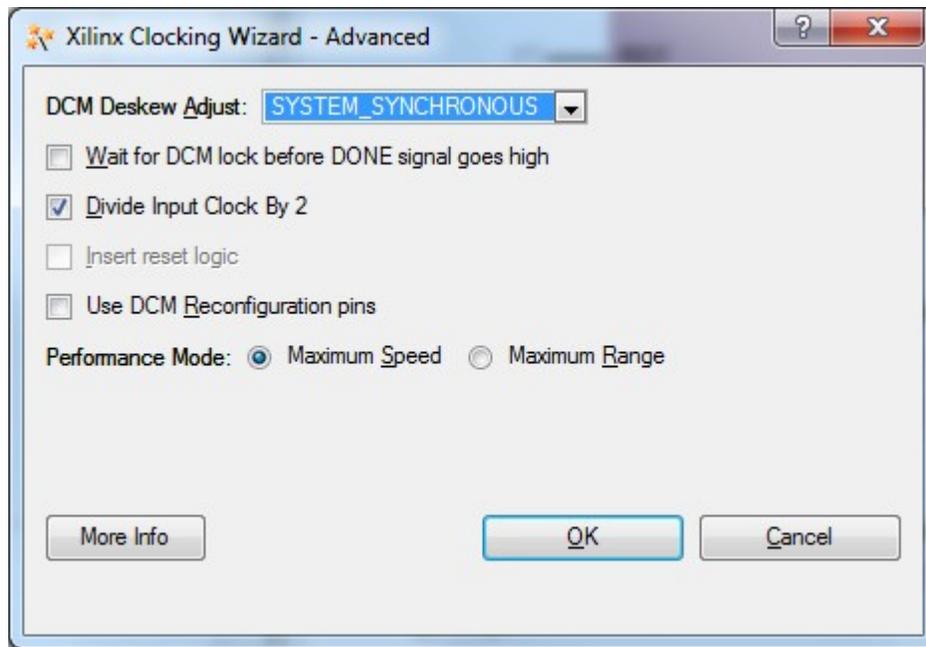


Figure 19: Divide by 2

Xilinx will now create a clock divider module that will divide our 100MHz clock into a 5MHz clock. Our next step is to instantiate the clock divider and the lowpass filter into our `adc_dac` module. But we must take care of one more issue before doing this. The filter we created only accepts signals that have zero DC offset, meaning that they have to be centered around 0. Thus, the DC offset we added to our signal earlier will not work with our filter. To fix this, we must subtract off the DC value in our VHDL code. There are many ways of doing this, for this example we will simply subtract `0x"800"` which is approximately 1.5V to every value of our signal. Then pass this through the filter and then add back `0x"800"` to the filter output and pass it to the DAC.

The new VHDL code that will instantiate the lowpass filter, DCM clock divider and take care of the DC offset is given in the Appendix titled Digital Filter Code (just copy and paste over the existing `adc_dac` module):

Then add these lines at the end of the `.ucf` file:

```
# dip switch 1
NET "sw0"          LOC = "U25";
```

After Implementing the design and programming the FPGA. We can use the function generator as the input with $\pm 1 V_{pp}$ and verify that our filter attenuates frequencies above 1kHz: (note: if your filter output has several spikes then most likely your ADC/DAC and filter clock are not synchronized make sure that they are both synchronized to the rising edge of the clock)

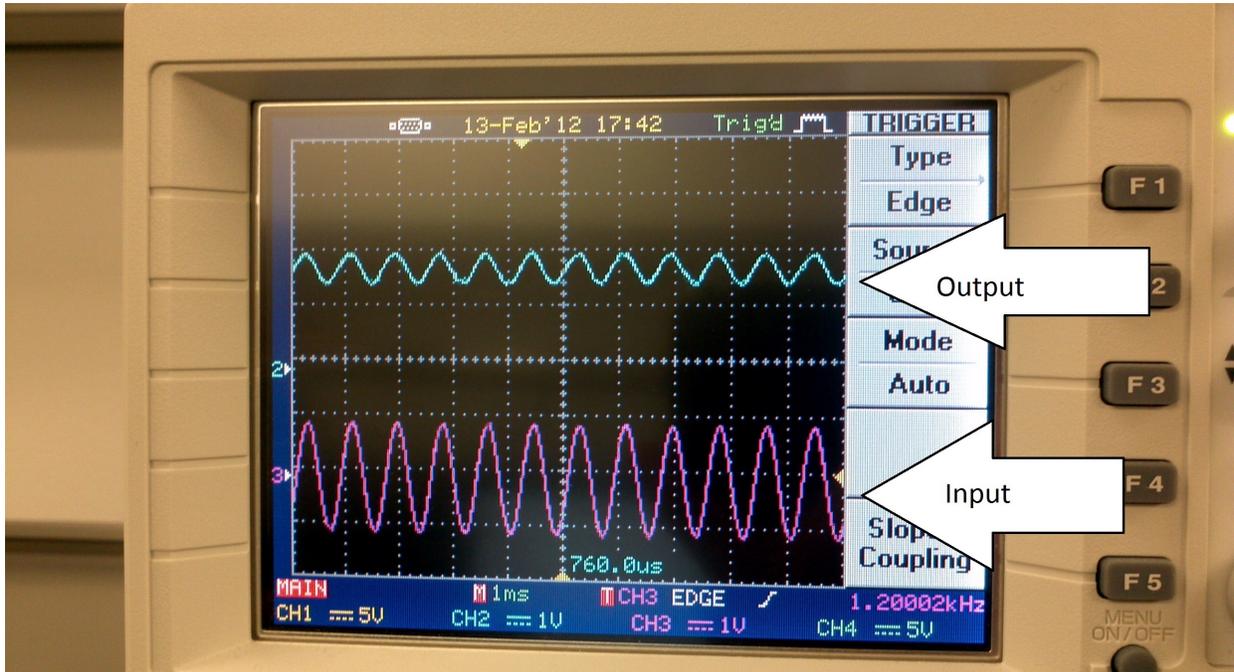


Figure 20: Attenuated output of Digital Lowpass Filter

Least Mean Square Adaptive Filter on FPGA

A least mean square adaptive filter can also be implemented using a similar process.

For this section we will use the MATLAB Simulink HDL Coder example of noise cancellation using the LMS Adaptive Filter.

This file can be found in the following path : Program Files → MATLAB → R2011a → toolbox → hdlcoder → hdlcoderdemos → hdlcoderlms.mdl

First copy-paste this file into a directory called 'NoiseRemove'.

Before generating the HDL code for this block, we must make the following changes:

- Change the sampling frequency to match our previous HDL code for the lowpass filter which is 52kHz
- The model is made for 16-bit fixed point but our ADC is 12-bit.

First change the sampling frequency of the lms filter. Open hdlcoderlms.mdl in MATLAB. Then change the sampling time from the default 1/8000 to 1/52000 by double clicking all the blocks and changing it. (This may sound tedious but you only need to do this for the input blocks as most of the other blocks inherit their sampling time from the previous block). Figure 21 shows a screenshot of how

to change the sampling time.

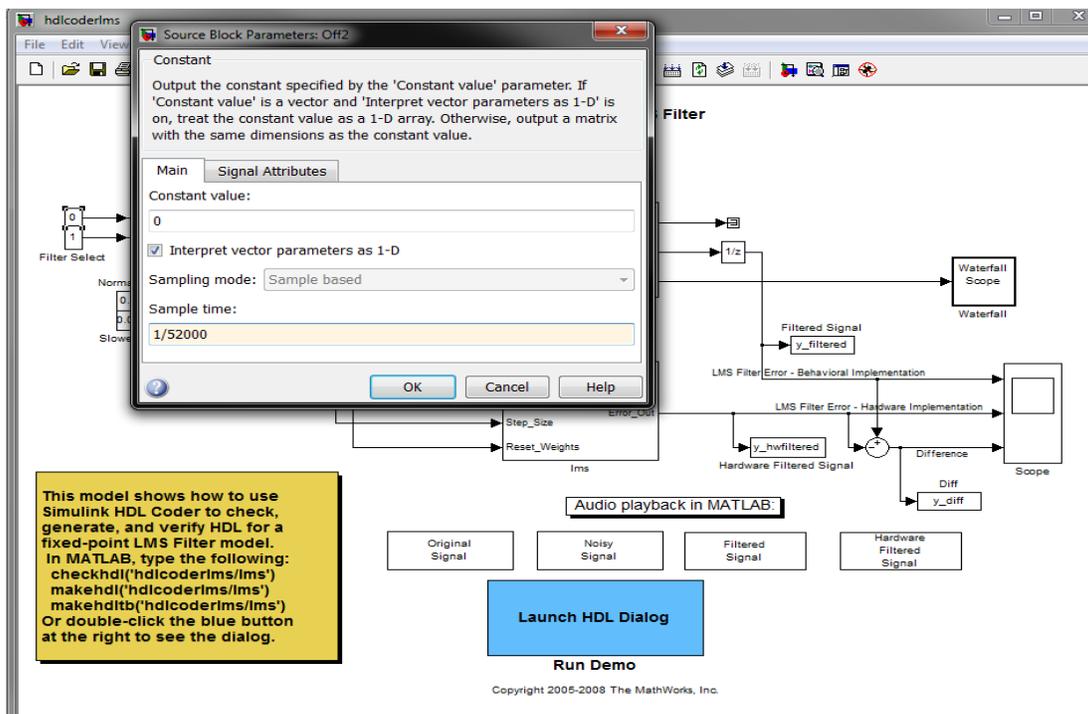


Figure 21: Change Sampling Time

Then we need to change the model to work with 12 bits instead of 16 bits.

Double-click the Acoustic Environment and change the Convert block to the data type specified below (Figure 22):

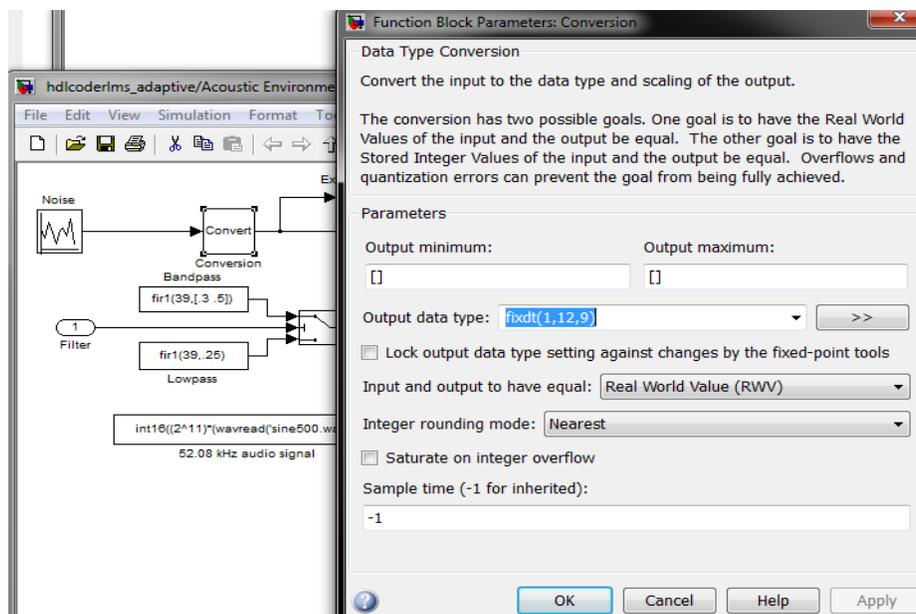


Figure 22: 12 Bits

Double-click the lms block and change the final summing node to have the following data type shown in figure 23:

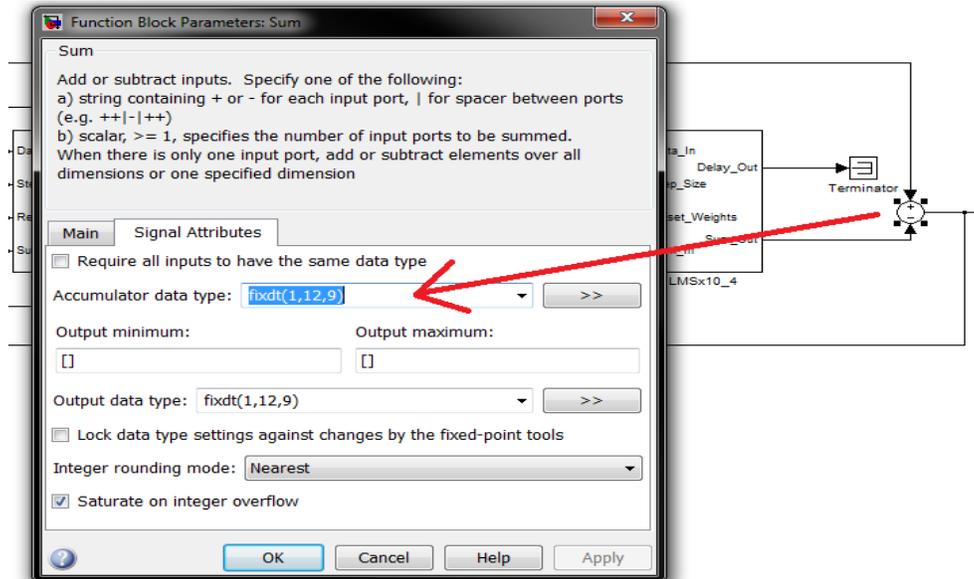


Figure 23: Summing node data type

Save the project and go to Tools → HDL Code Generation → Options.

Set the following options to generate VHDL code for the lms block and save it into a folder named 'lms_HDL' (This is shown in Figure 24)

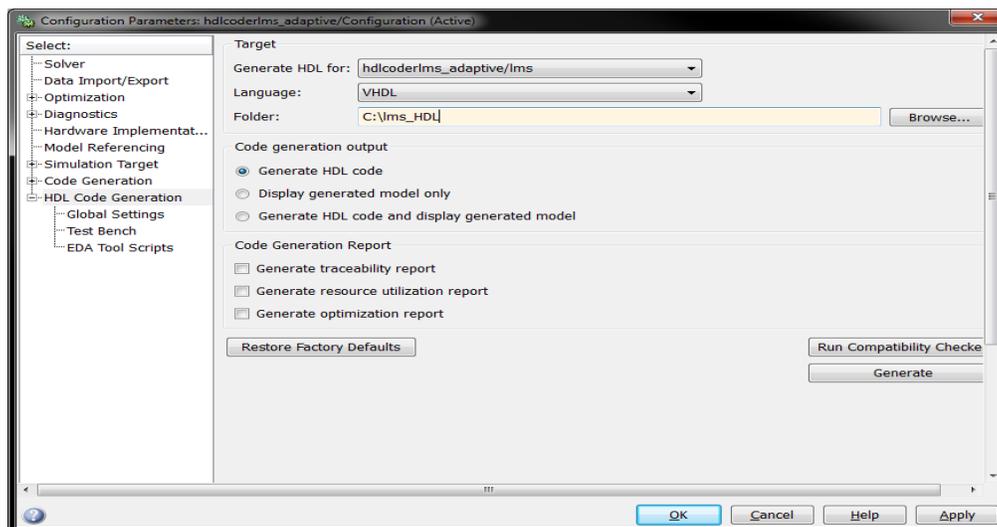


Figure 24: HDL Coder Settings

Click OK.

Then click Tools → HDL Code Generation → Generate HDL

After the HDL code generation completes, we follow the steps in the previous section for the lowpass filter but instead of instantiating the lowpass filter we would instantiate the lms adaptive filter.

The VHDL code instantiating the adaptive filter is given in the appendix title Adaptive Filter HDL

Here is the UCF file for interfacing with the ML506:

```
# ML506 100 MHz clock pin
NET "CLK"          LOC = "AH15";

# Pmod AD1 ADC Pins on J6 Expansion Header
NET "CS"           LOC = "H33";
NET "DIN"          LOC = "F34"; # noise
NET "DIN2"         LOC = "H34"; # desired, pilots mic # (noise + clean signal)
NET "SCLK"         LOC = "G33";

# Pmod DA2 DAC Pins on J6 Expansion Header
NET "SYNC"         LOC = "J34";
NET "DOUT"         LOC = "L33";
NET "SCLK2"        LOC = "P34";

# gpio switch 1
NET "sw0"          LOC = "U25";
```

After programming the FPGA, we created two inputs using MATLAB. One is noise and the other is a 500Hz signal + noise. The noise is stored in the left channel of a .wav file and the signal+noise is stored in the right channel.

We then input these two signals simultaneously into the adaptive filter. Below are pictures of the input and output. (Figures 25 and 26)

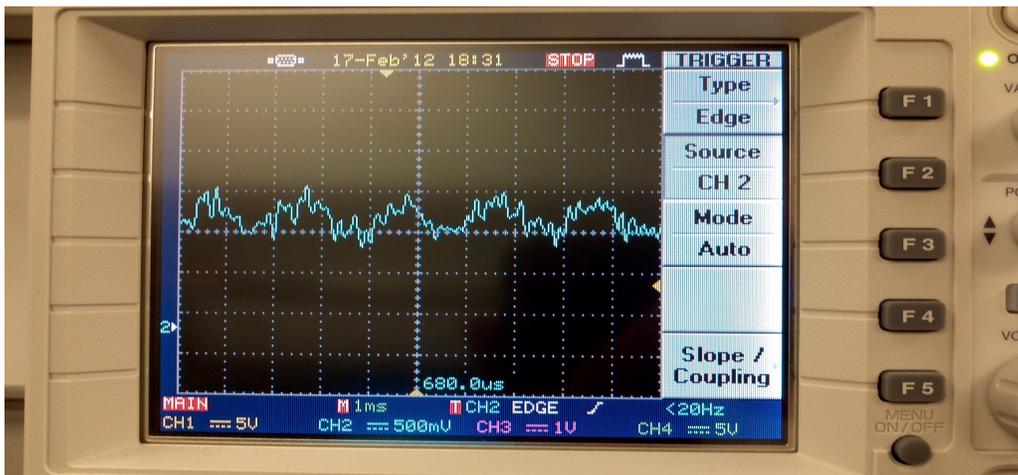


Figure 25: Signal with noise

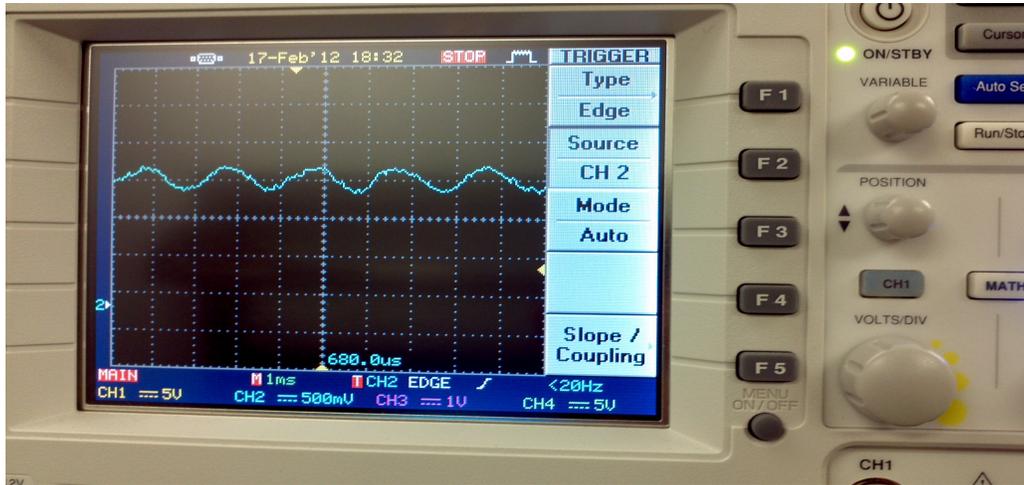


Figure 26: Filtered Signal

Appendix

Analog to Digital Conversion Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;

entity adc_dac is
  port(
    CLK : in std_logic;          -- ML506 100MHz clock

    CS : out std_logic;          -- chip select for ADC(active low)
    SYNC : out std_logic;        -- SYNC for DAC
    DIN : in std_logic;          -- ADC
    DOUT : out std_logic;        -- DAC

    -- These two clock signals will clock the ADC/DAC
    SCLK : out std_logic;        -- ADC
    SCLK2 : out std_logic;       -- DAC
  );
end adc_dac;

architecture Behavioral of adc_dac is

  -- FSM states
  type state_type is (IDLE, READ_DATA, FUNC, WRITE_DATA);

  -- initial state
  signal state : state_type := READ_DATA;

  -- data from the ADC
  signal data : std_logic_vector(11 downto 0);

  -- counter variable
  signal cnt : integer range 0 to 20 := 0;

  -- counter for clock division
  signal clkdiv : integer range 0 to 6;

  -- new clock from division
  signal newclk : std_logic := '0';
  signal risingedge : std_logic := '1';
```

```

-- reset signal
signal reset          : std_logic := '0';

begin

-- drive the ADC and DAC clock pins
-- The ADC and DAC can be clock up to 20 MHz
SCLK <= newclk;
SCLK2 <= newclk;

-- divide the ML506 clock to 20 MHz
clock_divide : process(CLK, reset)
begin
    if (reset = '1') then
    elsif (rising_edge(CLK)) then
        if (clkdiv = 5) then          -- divide 100MHz by 5
            risingedge <= risingedge xor '1';
            newclk <= newclk xor '1';
            clkdiv <= 0;
        else
            clkdiv <= clkdiv + 1;
        end if;
    end if;
end process clock_divide;

-- main process FSM
main : process (CLK, reset)

begin
    if (reset = '1') then
    elsif (rising_edge(CLK)) then
        if (clkdiv = 5 and risingedge = '1') then

            case state is

                -- each state takes 16 clock cycles to finish
                -- except FUNC which only takes 1

                when IDLE =>
                    CS <= '1';
                    SYNC <= '1';
                    if (cnt = 16) then
                        cnt <= 0;
                        state <= READ_DATA;
                    else
                        cnt <= cnt + 1;
                    end if;
                end case;
            end if;
        end if;
    end process main;
end;

```

```

        state <= IDLE;
    end if;

when READ_DATA =>
    CS <= '0';
    SYNC <= '1';
    cnt <= cnt + 1;

    if (cnt < 4) then
        cnt <= cnt + 1;
        state <= READ_DATA;
    elsif (cnt > 3 and cnt < 16) then
        cnt <= cnt + 1;

        -- the first 4 bits are 0000 only read the last 12
        data(15-cnt) <= DIN;
        state <= READ_DATA;
    elsif (cnt = 16) then
        cnt <= 0;
        state <= FUNC;
    end if;

-- signal processing would go in this state
-- but for now we don't do anything in here
when FUNC =>
    CS <= '1';
    SYNC <= '1';
    cnt <= 0;

    state <= WRITE_DATA;

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

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

        DOUT <= data(15 - cnt);
    end if;

```

```

        state <= WRITE_DATA;
    elseif (cnt = 16) then
        cnt <= 0;
        state <= IDLE;
    end if;

    end case;

    end if;
end if;
end process main;

```

end Behavioral;

MATLAB Filter Code

```

%% Design a lowpass filter to filter out some high frequency sounds in an
% audio file.

%% Create the filter
close all;
clear all;

Fs      = 52000;    % Sampling freq = 52 kHz
Fn      = Fs/2;
F3db    = 1000;    % Cutoff frequency

% Design the lowpass filter
% filter order is 5, with a 3dB point at F3db = 500Hz
filtdes = fdesign.lowpass('n,f3db', 5, F3db, Fs);
Hd = design(filtdes, 'butter');

% Convert it from the default structure (DF2SOS) to the desired structure,
% DF1SOS
Hd = convert(Hd, 'df1sos');

% Examine the response
fvtool(Hd, 'Fs', Fs, 'FrequencyScale', 'log');

%% Create the Quantized Filter

Hd.arithmetic      = 'fixed';
Hd.InputWordLength = 12;
Hd.InputFracLength = 11;
Hd.OutputWordLength = 12;
Hd.OutputMode      = 'SpecifyPrecision';
Hd.OutputFracLength = 11;
Hd.CoeffWordLength = 12;
Hd.AccumWordLength = 200;
Hd.NumStateWordLength = 20;

```

```

Hd.DenStateWordLength = 20;
Hd.CastBeforeSum      = false;
Hd.RoundMode          = 'nearest';
Hd.OverflowMode       = 'saturate';

fvtool(Hd, 'Fs', Fs, 'FrequencyScale', 'log');

%% Requantize the Filter
% In the previous plot, fvtool shows that the quantized passband is
% approximatley 2dB lower than the desired response. Adjust the
% coefficient word length from 12 to 16 to get the quantized response
% closer to the reference double-precision response and zoom in on the
% passband response.

Hd.CoeffWordLength = 16;
fvtool(Hd, 'Fs', Fs, 'FrequencyScale', 'log');
axis([0 1.0 -1 1]);

%% Examine the Scale Values

scales = Hd.scalevalues .* 2^Hd.InputFracLength
% Now scale the filter using the frequency domain infinity norm.
scale(Hd, 'Linf');
% After scaling, the scale values are all one in this case
scales = Hd.scalevalues

%% Generate HDL Code from the Quantized Filter

% Create a temporary work directory
workingdir = 'hdl_work';
generatehdl(Hd, 'Name', 'hdlbutter', 'TargetLanguage', 'VHDL', ...
    'TargetDirectory', workingdir);
edit(fullfile(workingdir, 'hdlbutter.vhd'));

%% Generate a Test Bench from the Quantized Filter

% Since the passband of this filter is so low relative to the sampling
% rate, a custom input stimulus is a better way to test the filter
% implementation. Build the test input with one cycle of each of 50 to 300
% Hz in 50 Hz steps.

% Generate a VHDL test bench to verify that the results match the MATLAB
% results exactly.

% After generating the test bench, open the generated file in the editor

userstim = [];
for n = [50, 100, 150, 200, 250, 300, 500, 600, 700, 800, ...
    1000, 1200, 1500, 2000]
    userstim = [userstim, sin(2*pi*n/Fs*(0:Fs/n))];
end

generatetb(Hd, 'VHDL', 'TestBenchName', 'hdlbutter_tb', ...
    'TestBenchStimulus', [], ...
    'TestBenchUserStimulus', userstim, ...
    'TargetDirectory', workingdir);

```

```

edit(fullfile(workingdir, 'hdlbutter_tb.vhd'));

%% Simulation

% After Creating the VHDL and VHDL test bench run it in ModelSim and
% compare the results with the MATLAB simulation, describe below:

% Plot the input
xrange = (0:length(userstim) - 1);
y = filter(Hd, userstim);
subplot(2,1,1); plot(xrange, userstim);
%axis([0 length(userstim) -1.1 1.1]);
title('HDL Butterworth filter in Stimulus');
xlabel('Sample #');

% Plot the output of the filter
% Notice how it attenuates the frequencies higher than 500 Hz
subplot(2,1,2); plot(xrange, y);
%axis([0 length(userstim) -1.1 1.1]);
title('HDL Butterworth filter out Response');
xlabel('Sample #');

```

Digital Filter Code

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;

entity adc_dac is
    port(
        CLK      : in std_logic;          -- ML506 clock
        CS       : out std_logic;         -- chip select for ADC(active low)
        SYNC     : out std_logic;         -- SYNC for DAC
        DIN      : in std_logic;         -- ADC
        DOUT     : out std_logic;         -- DAC

        -- turn low pass filter on or off with a SW0
        sw0      : in std_logic;

        -- These two clock signals will clock the ADC/DAC
        SCLK     : out std_logic;         -- ADC
        SCLK2    : out std_logic;        -- DAC
    );
end adc_dac;

architecture Behavioral of adc_dac is

    -- FSM states
    type state_type is (IDLE, READ_DATA, FUNC, WRITE_DATA);
    signal state      : state_type := READ_DATA;

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

    signal reset      : std_logic := '0';

```

```

-- filter clock
signal newclk_filter      : std_logic := '0';
signal filter_cnt        : integer range 0 to 100 := 0;

-- 5MHz clock from DCM
signal clk_5e6           : std_logic;

COMPONENT clk_div
PORT(
    CLKIN_IN : IN std_logic;
    CLKDV_OUT : OUT std_logic;
    CLK0_OUT : OUT std_logic;
    LOCKED_OUT : OUT std_logic
);
END COMPONENT;

-- lowpass filter signals
signal clk_enable : std_logic := '1';
signal filter_in  : std_logic_vector(11 downto 0);
signal filter_out : std_logic_vector(11 downto 0);

component hdlbutter
port (
    clk           : in std_logic;
    clk_enable    : in std_logic;
    reset         : in std_logic;
    filter_in     : in std_logic_vector(11 downto 0);
    filter_out    : out std_logic_vector(11 downto 0)
);
end component;

-- constant to convert the ADC values to signed
constant one_five : std_logic_vector(11 downto 0) := X"800";    -- 1.5V

```

begin

```

-- Instantiate the Butterworth filter
butter1 : hdlbutter port map(
    clk           => newclk_filter,
    clk_enable    => clk_enable,
    reset         => reset,
    filter_in     => filter_in,
    filter_out    => filter_out
);

-- Instantiate the DCM clock divider
Inst_clk_div: clk_div PORT MAP(
    CLKIN_IN => CLK,
    CLKDV_OUT => clk_5e6,
    CLK0_OUT => open,
    LOCKED_OUT => open
);

-- drive the ADC and DAC clock pins
SCLK <= newclk;
SCLK2 <= newclk;

-- Each of the states take 16 clock cycles long
-- The DCM takes the 100MHz FPGA clock and divides it:
-- 100Mhz/ (2*10) = 5 MHz
-- But it takes 3*16 clock cycles to sample and output the sound

```

```

-- so 5 MHz/ (3*16) = 104.17 kHz sampling frequency

-- But on each rising edge of the 104.17 kHz clock we increment
-- clkdiv by 1. So newclock is 104.17 kHz / 2 = 52.08 kHz
-- which is just above the CD sampling rate of 44.1kHz
clock_divide : process(clk_5e6, reset)
begin
    if (reset = '1') then
    elsif (rising_edge(clk_5e6)) then
        if (clkdiv = 1) then
            risingedge <= risingedge xor '1';
            newclk <= newclk xor '1';
            clkdiv <= 0;
        else
            clkdiv <= clkdiv + 1;
        end if;
    end if;
end process clock_divide;

-- Change the clock frequency of the filter clk to 52.08 kHz
filter_clock : process(clk_5e6)
begin
    if (rising_edge(clk_5e6)) then
        if (filter_cnt = 48) then
            if (clkdiv = 1 and risingedge = '1') then
                newclk_filter <= newclk_filter xor '1';
                filter_cnt <= 0;
            end if;
        else
            filter_cnt <= filter_cnt + 1;
        end if;
    end if;
end process filter_clock;

-- main process FSM
main : process (clk_5e6, reset, sw0, filter_out)

    -- Convert to signed
    variable DIN_temp                : std_logic_vector(12 downto 0);
    variable one_five_temp           : std_logic_vector(12 downto 0);
    variable DIN_signed_temp         : signed(12 downto 0);
    variable DIN_signed_temp2       : signed(12 downto 0);
    variable DIN_signed              : signed(11 downto 0);
    variable one_five_signed        : signed(12 downto 0);

    -- Convert to unsigned
    variable data_out_temp           : signed(12 downto 0);
    variable data_out_unsigned_temp  : signed(12 downto 0);
    variable data_out_unsigned_temp2 : signed(11 downto 0);
    variable data_out_unsigned      : unsigned(11 downto 0);

begin
    if (reset = '1') then
    elsif (rising_edge(clk_5e6)) then
        if (clkdiv = 1 and risingedge = '1') then

            case state is

                when IDLE =>
                    CS <= '1';
                    SYNC <= '1';
                    if (cnt = 16) then

```

```

        cnt <= 0;
        state <= READ_DATA;
    else
        cnt <= cnt + 1;
        state <= IDLE;
    end if;

when READ_DATA =>
    CS <= '0';
    SYNC <= '1';
    cnt <= cnt + 1;

    if (cnt < 4) then
        cnt <= cnt + 1;
        state <= READ_DATA;
    elsif (cnt > 3 and cnt < 16) then
        cnt <= cnt + 1;
        data(15-cnt) <= DIN;
        state <= READ_DATA;
    elsif (cnt = 16) then
        cnt <= 0;
        state <= FUNC;
    end if;

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

    -- remove the offset of 1.5V and convert to signed
    -----
    DIN_temp := '0' & data;
    one_five_temp := '0' & one_five;

    -- convert to signed
    DIN_signed_temp := signed(DIN_temp);
    one_five_signed := signed(one_five_temp);

    DIN_signed_temp2 := DIN_signed_temp - one_five_signed;
    DIN_signed := DIN_signed_temp2(11 downto 0);
    -----

    filter_in <= std_logic_vector(DIN_signed);

    state <= WRITE_DATA;

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

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

```

```

-- convert back to unsigned and
-- add the offset of 1.5V = 0x800~0x7FF
data_out_temp := '0' & signed(filter_out);
data_out_unsigned_temp := data_out_temp + one_five_signed;

data_out_unsigned_temp2 := data_out_unsigned_temp(11 downto 0);
data_out_unsigned := unsigned(data_out_unsigned_temp2);

if (sw0 = '1') then
    DOUT <= data_out_unsigned(15-cnt);
else
    DOUT <= data(15-cnt);
end if;

state <= WRITE_DATA;

elsif (cnt = 16) then
    cnt <= 0;
    state <= IDLE;
end if;

end case;

end if;
end process main;

```

end Behavioral;

Adaptive Filter HDL

```

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

entity adaptive_filter is
    port(
        CLK      : in std_logic;          -- ML506 Clock

        -- ADC and DAC Pins
        CS       : out std_logic;        -- chip select for ADC(active low)
        SYNC     : out std_logic;        -- SYNC for DAC
        DIN      : in std_logic;        -- ADC
        DIN2     : in std_logic;        -- ADC
        DOUT     : out std_logic;        -- DAC

        -- turn on filter with a switch
        sw0      : in std_logic;

        -- These two clock signals will clock the ADC/DAC
        SCLK     : out std_logic;        -- ADC
        SCLK2    : out std_logic;        -- DAC
    );
end adaptive_filter;

```

architecture Behavioral of adaptive_filter is

```
-- FSM states
type state_type is (IDLE, READ_DATA, FUNC, WRITE_DATA);
signal state      : state_type := READ_DATA;

-- ADC/DAC Temporary Signals
signal data      : std_logic_vector(11 downto 0);
signal data2    : std_logic_vector(11 downto 0);
signal cnt      : integer range 0 to 20 := 0;
signal lms_cnt  : integer range 0 to 150 := 0;
signal clkdiv   : integer range 0 to 11;
signal newclk   : std_logic := '0';
signal risingedge : std_logic := '1';

signal rst      : std_logic := '0';

-- LMS Temporary Signals
signal reset    : std_logic := '0';
signal clk_enable : std_logic := '1';

-- 0.002 000000000100
signal Step_Size : std_logic_vector(11 downto 0) := "000000000100";
signal Reset_Weights : std_logic := '0';
signal ce_out    : std_logic;
signal Error_Out : std_logic_vector(11 downto 0);

signal Input    : std_logic_vector(11 downto 0);
signal Desired  : std_logic_vector(11 downto 0);

-- Adaptive filter clock
signal newclk_filter : std_logic := '0';
signal filter_cnt    : integer range 0 to 100 := 0;

-- 5MHz clock from DCM
signal clk_5e6 : std_logic;

-- Declare the DCM clock divider
COMPONENT clk_div
PORT(
    CLKIN_IN : IN std_logic;
    CLKDV_OUT : OUT std_logic;
    CLK0_OUT : OUT std_logic;
    LOCKED_OUT : OUT std_logic
);
END COMPONENT;

-- Declare the LMS MATLAB Component
component lms
port(
    clk      : in std_logic;
    reset    : in std_logic;
    clk_enable : in std_logic;
    Input    : in std_logic_vector(11 downto 0);
    Desired  : in std_logic_vector(11 downto 0);
    Step_Size : in std_logic_vector(11 downto 0);
    Reset_Weights : in std_logic;
    ce_out   : out std_logic;
    Error_Out : out std_logic_vector(11 downto 0)
);
end component;
```

begin

```
-- Instantiate the LMS MATLAB Block
LMS_Block      :      lms
    port map(
        clk => newclk_filter,
        reset => reset,
        clk_enable => clk_enable,
        Input => Input,
        Desired => Desired,
        Step_Size => Step_Size,
        Reset_Weights => Reset_Weights,
        ce_out => ce_out,
        Error_Out => Error_Out
    );

-- Instantiate the DCM clock divider
Inst_clk_div: clk_div PORT MAP(
    CLKIN_IN => CLK,
    CLKDV_OUT => clk_5e6,
    CLK0_OUT => open,
    LOCKED_OUT => open
);

-- drive the ADC and DAC clock pins
SCLK <= newclk;
SCLK2 <= newclk;

-- Each of the states take 16 clock cycles long
-- The DCM takes the 100MHz FPGA clock and divides it:
-- 100Mhz/ (2*10) = 5 MHz
-- But it takes 3*16 clock cycles to sample and output the sound
-- so 5 MHz/ (3*16) = 104.17 kHz sampling frequency

-- But on each rising edge of the 104.17 kHz clock we increment
-- clkdiv by 1. So newclock is 104.17 kHz / 2 ~ 52 kHz
-- which is just above the CD sampling rate of 44.1kHz
clock_divide : process(clk_5e6, rst)
begin
    if (rst = '1') then
    elsif (rising_edge(clk_5e6)) then
        if (clkdiv = 1) then
            risingedge <= risingedge xor '1';
            newclk <= newclk xor '1';
            clkdiv <= 0;
        else
            clkdiv <= clkdiv + 1;
        end if;
    end if;
end process clock_divide;

-- Change the clock frequency of the filter clk to 52 kHz
-- 5 MHz / (2*48) ~ 52 kHz
filter_clock : process(clk_5e6)
begin
    if (rising_edge(clk_5e6)) then
        if (filter_cnt = 48) then
            if (clkdiv = 1 and risingedge = '1') then
                newclk_filter <= newclk_filter xor '1';
                filter_cnt <= 0;
            end if;
        else

```

```

        filter_cnt <= filter_cnt + 1;
    end if;
end if;
end process filter_clock;

-- main process FSM
main : process (clk_5e6, sw0, Error_Out, rst)

    -- Convert to signed
    variable DIN_temp                : std_logic_vector(12 downto 0);
    variable DIN_temp2               : std_logic_vector(12 downto 0);
    variable one_five_temp           : std_logic_vector(12 downto 0);
    variable DIN_signed_temp         : signed(12 downto 0);
    variable DIN_signed_temp4        : signed(12 downto 0);
    variable DIN_signed_temp2        : signed(12 downto 0);
    variable DIN_signed_temp8        : signed(12 downto 0);
    variable DIN_signed               : signed(11 downto 0);
    variable DIN_signed2              : signed(11 downto 0);
    variable one_five_signed         : signed(12 downto 0);

    -- Convert to unsigned
    variable data_out_temp            : signed(12 downto 0);
    variable data_out_unsigned_temp   : signed(12 downto 0);
    variable data_out_unsigned_temp2  : signed(11 downto 0);
    variable data_out_unsigned        : unsigned(11 downto 0);

    -- constant to convert the ADC values to signed
    constant one_five                 : std_logic_vector(11 downto 0) := X"800";    -- 1.5V

begin
    if (rst = '1') then
    elsif (rising_edge(clk_5e6)) then
        if (clkdiv = 1 and risingedge = '1') then

            case state is

                when IDLE =>
                    CS <= '1';
                    SYNC <= '1';
                    if (cnt = 16) then
                        cnt <= 0;
                        state <= READ_DATA;
                    else
                        cnt <= cnt + 1;
                        state <= IDLE;
                    end if;

                when READ_DATA =>
                    CS <= '0';
                    SYNC <= '1';
                    cnt <= cnt + 1;

                    if (cnt < 4) then
                        cnt <= cnt + 1;
                        state <= READ_DATA;
                    elsif (cnt > 3 and cnt < 16) then
                        cnt <= cnt + 1;

                        -- the first 4 bits are 0000 only read the last 12
                        data(15-cnt) <= DIN;
                        data2(15-cnt) <= DIN2;
                        state <= READ_DATA;
                    end if;
                end case;
            end if;
        end if;
    end process;
end process;

```

```

    elsif (cnt = 16) then
        cnt <= 0;
        state <= FUNC;
    end if;

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

    -- remove the offset of 1.5V and convert to signed
    -----
    DIN_temp           := '0' & data;
    DIN_temp2          := '0' & data2;
    one_five_temp      := '0' & one_five;

    -- convert to signed
    DIN_signed_temp := signed(DIN_temp);
    DIN_signed_temp4 := signed(DIN_temp2);
    one_five_signed := signed(one_five_temp);

    DIN_signed_temp2 := DIN_signed_temp - one_five_signed;
    DIN_signed_temp8 := DIN_signed_temp4 - one_five_signed;
    DIN_signed       := DIN_signed_temp2(11 downto 0);
    DIN_signed2      := DIN_signed_temp8(11 downto 0);
    -----

    -- Noise
    Input <= std_logic_vector(DIN_signed);

    -- Signal + Noise
    Desired <= std_logic_vector(DIN_signed2);

    state <= WRITE_DATA;

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

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

    -- convert back to unsigned and add the offset of 1.5V = 0x800~0x7FF
    data_out_temp           := '0' & signed(Error_Out);
    data_out_unsigned_temp := data_out_temp + one_five_signed;
    data_out_unsigned_temp2 := data_out_unsigned_temp(11 downto 0);
    data_out_unsigned       := unsigned(data_out_unsigned_temp2);

    if (sw0 = '1') then
        DOUT <= data_out_unsigned(15 - cnt);
    else
        DOUT <= data2(15 - cnt);
    end if;

```

```
state <= WRITE_DATA;  
elsif (cnt = 16) then  
  cnt <= 0;  
  state <= IDLE;  
end if;
```

```
end case;
```

```
end if;
```

```
end process main;
```

```
end Behavioral;
```

References

"Cosmiac - Spartan 3E Tutorials - Configurable Space Microsystems Innovations & Applications Center." *Cosmiac*. Web. 21 Feb. 2012. <http://www.cosmiac.org/spartan3e_tutorials.html>.

"Clamper (electronics)." *Wikipedia*. Wikimedia Foundation, 02 Dec. 2012. Web. 21 Feb. 2012. <[http://en.wikipedia.org/wiki/Clamper_\(electronics\)](http://en.wikipedia.org/wiki/Clamper_(electronics))>.

"Forum for Electronics." *Forum for Electronics*. Web. 21 Feb. 2012. <<http://www.edaboard.com/>>.

"MATLAB Answers - MATLAB Central." *Document Moved*. Web. 21 Feb. 2012. <<http://www.mathworks.com/matlabcentral/answers/>>.