# Embedded Systems Lab 5: Basic Audio and Filtering

## Introduction

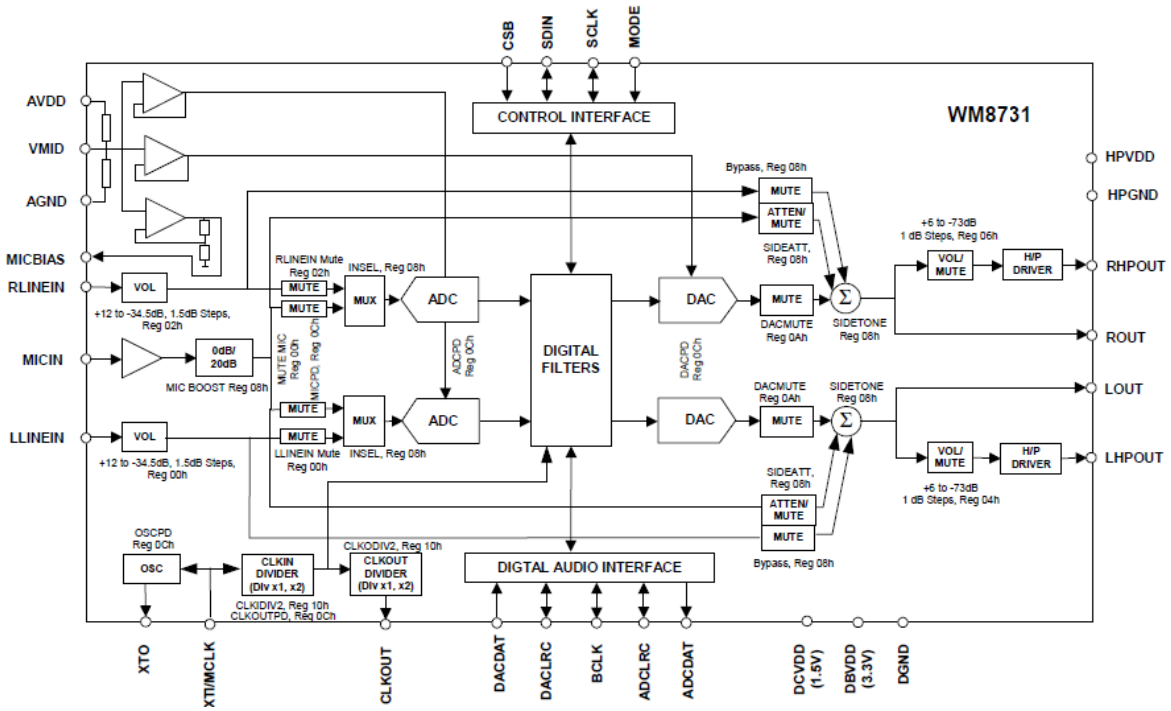
In this lab we will be using the FPGA to interface to an Audio Codec. The Audio Codec is an integrated circuit that is designed to connect a digital device such as a microcontroller or FPGA to headphones, speakers, microphones and other audio equipment. It contains a two-channel Analogue to Digital Converter (ADC), two-channel Digital to Analogue Converter (DAC) alongside amplifiers and buffers. In this lab, we will be using the DAC and amplifiers only; we will not do any audio input from the ADC.

This lab requires that speakers or headphones be connected to the DE2-115's Line Out port.

**NOTE:** If you are using headphones be very careful when adjusting the volume! You should remove your headphones when the board is being programmed and only put them on again once you're sure that you've got the volume set at suitable levels.

## The Codec

The Codec we're using is the Wolfson Microelectronics WM8731, a block diagram for which can be seen below.

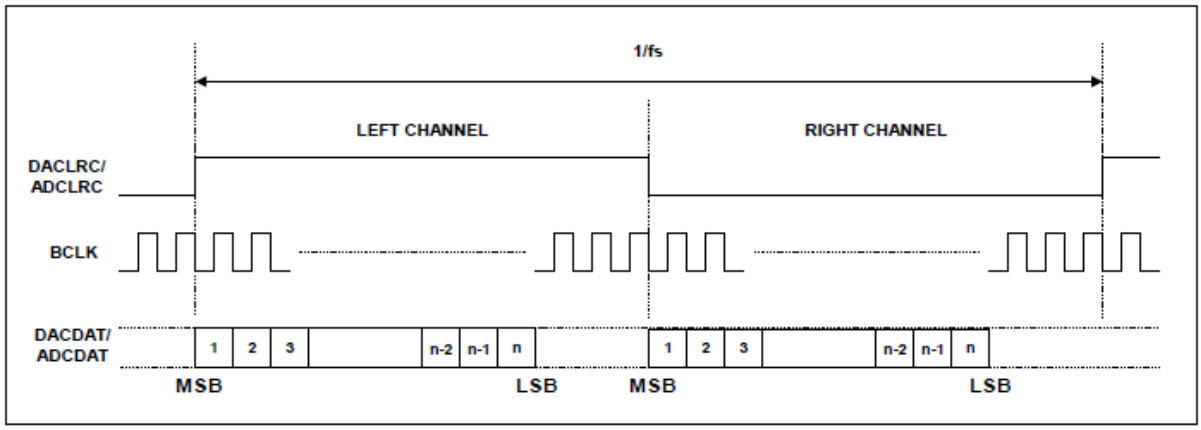



On this diagram, you can see the two channels each for the DAC and ADC, the blocks for filtering and you may note that there are signal paths that bypass the ADC/DACs entirely. These may be useful, for example, for using the microphone input to 'voice over' the normal DAC output.

Also to note on this diagram is that there are two interfaces; one for audio and one for control. The Audio interface uses a standard called "Inter-IC Sound" (I2S) where the control interface uses "Inter-IC Communications" (I2C). Despite the similar names, the protocols are quite different.

The files provided in the lab package use the control interface to configure the Codec for our use. We will modify the code a little bit to alter the volume of the output signal but otherwise don't have to understand the mechanics of the I2C bus as it's quite complex (we will cover it later in the course).

The I2C interface by contrast is quite simple. For the DAC path that we will be using, there are just three wires: The clock line, the serial data line and the "LRCLK", or left/right clock signal. A timing diagram can be seen below. As can be seen, to transfer signals to the DAC, one must select which channel you want by setting the LRCLK high or low then serially shifting the sample value data out the DACDAT line.




## Filtering

One of the most basic tasks in Digital Signal Processing is filtering. This involves taking an incoming signal and removing (or attenuating) components of that signal based on their frequency. A low-pass filter will 'turn down' high frequencies, a high-pass filter will 'turn down' low frequencies and a band-pass will only let through a range of frequencies in the middle.

A common type of simple filter for discrete time signal processing is called the Finite Impulse Response Filter (FIR). This type of filter keeps a history of the last several input values and provides an output that is a weighted sum of them all. The values of the weights determine whether the FIR is low-pass, high-pass or band-pass and at what frequency it operates. FIR filters are distinct from Infinite Impulse Response (IIR) filters, in that the latter also uses past output values in the output sum (not just input values).

A special case of the FIR filter comes about when all the coefficients are the same value. In this case, the FIR is known as a Moving Average Filter (MAV). The MAV is useful because the coefficient by which an input value is scaled is independent of how old it is. This means an implementation of the MAV is very simple: We can keep a 'running sum', each time step we add the newest value to the sum, remove the oldest value then multiply the whole sum by the appropriate common coefficient. Because of this structure, we don't need access to input values except the newest and oldest, which means that we can store the history if a First-In, First-Out (FIFO) buffer.

# Part 1: Wave Generation

In this section, we will start to generate some basic waveforms and pass them to the Codec. We will also do some basic configuration of the Codec to get the volume correct.

## Understanding the Code

The Lab5_Synthesizer project instantiates three modules at the top level: The 'I2C_AV_Config' module communicates with the configuration interface on the Codec, the 'audio' module contains all the code for interfacing with the I2S side of the Codec and finally a PLL has been created that generates the required clocks.

**NOTE:** There are a few references to Video, both in the PLL and inside the AV Config module. These are to do with the Video Decoder IC that is on the same I2C interface as the Codec. Despite the fact that we will not be using this decoder, we must still set it up in a basic fashion to stop it interfering.

The values to be written to the registers should be calculated from the Codec data sheet that is also part of the lab kit. The register definitions are found through the Device Description chapter.

Inside the AV Config module, you will find a look-up table that contains the configuration values that are to be written to the codec (and to the video decoder, ignore those ones). The most significant seven bits of each LUT entry are the register address for which they're destined and the next bit is whether to read (1) or write (0); for example, the value labelled A_PATH_CONTROL has the most significant 8 bits ob00001000 which should be a write to address ob0000100; checking the data sheet, we find that the "Analog Audio Path Control" register on page 33 has that address. The least-significant 8-bits of the look up table is the value to be written to that register.

Inside the 'audio' module, we find the 'adio_codec' module that is responsible for doing the low-level I2S interfacing and clock generation. The interface with this is simple: It provides a 'wave' input and 'wave_clock' output; at every negative edge of the 'wave_clock', external logic should set the 'wave' input to the next sample value to be output.

## Activities

Open the Lab5_Synthesizer project from the Lab kit.

We will start generating a simple square wave. In the 'audio' module, create a simple Verilog clock divider that divides the wave_clock by 64. Write a statement that sets 'wave' to be either 16'hFFFFC000 or 16'h3DFF depending on the value of that divided clock; these numbers will form the high and low values for our square wave.

**NOTE:** The first value is in fact a two's complement negative number.

Your code may look something like the following:

```
reg [6:0] square_gen;
always @(negedge wave_clock)
       square_gen <= square_gen + 1;

assign wave = square_gen[6] ? 16'hFFFFC000 : 16'h3DFF;
```

Compile the project and download it to the board. You should hear a very quiet tone out of the speakers. Next we will turn up the volume.

Open the Codec data sheet and find the register definitions for the headphone output controls on page 31. Use these definitions to come up with new 8-bit register values that should be written to the Codec, but this time set the output volume to -10dB (the other two bits in the register should be zero).

Find the old values in the AV Config module, replace them with the new, compile the program and program to the board. Confirm that the volume increase has worked. Remember that the number in the Config module's look up table is a 16-bit value where the first 8-bits are address and the last 8-bits are data, you should only change the data bits.

## Proper Waveforms

Now we will replace the fixed- frequency square wave with a nice Sine wave of variable frequency. A Wave Generator module has been written that can count through addresses at different rates depending on a frequency input. The address is then used to select a sample from a pre-written Sine wave look up table.

Add the 'wave_generator.v' file from the lab package to your project. Remove the square wave generator from above and instantiate this generator module inside the 'audio' module with the following connections:

1. Wave Generator's sample clock is the wave clock
2. Wave Generator's sound out is the wave
3. Wave Generator's reset is the audio module's reset
4. Wave Generator's instrument select is the audio module's instrument
5. Set 'freq' to 400 for now

Synthesize, download and listen to the new sound, it should be 'cleaner' than the old sound.

In the lab package, you'll find alternative look up tables that sound more like brass instruments, stringed instruments or saw-tooth "ramp" waveform. Pick one of these and instantiate it inside the Wave Generator module alongside the Sine wave lookup table. Use the 'instrument' input to select which of the two outputs get connected to the Wave Generator's 'sound out'. You will see in the top-level file that the instrument select has been wired to SW[0]. Download your new design to the board and use this switch to toggle between the different sounds.

## Playing a Tune

In the audio module, instantiate a 'staff' module and a 'demo_sound1' module as follows:

```
wire    [7:0]           key;
wire    [15:0]          freq;

demo_sound1 ds1 (
                        .clock(score_clock),
                        .key_code(key),
                        .k_tr(reset_n)
);
staff s1 (
                        .key(key),
                        .freq(freq)
);
```

Connect the staff's 'freq' output to the Wave Generator's freq input. The Score Clock is a simple clock generated at the top level to move between notes.

The 'demo_sound1' module here produces a sequence of output key codes; that is, numbers that correspond to different keys on a piano. The 'staff' module takes those keys and converts them to the appropriate frequency to be put in to the Wave Generator.

Synthesize the above, download it to the board and listen to the pre-programmed tune. You may still use the instrument switch to change how it sounds.

### Polyphonic!

Duplicate the instantiations of the Demo Sound, Staff and Wave Generator modules giving each copy their own 'key', 'freq' and 'wave' wires. Change the type of the second Demo Sound module from 'demo_sound1' to 'demo_sound2'. Change the output of the 'audio' module to be the sum of the 'wave' signals from each of the two Wave Generators.

## Part 2: Filtering

For the purposes of this lab, we will implement a special case of the Finite Impulse Response Filter, the Moving Average Filter.

### Activities

Create a new module called 'mav_filter' with inputs and outputs as shown:

```
module mav_filter(
        input   [15:0]  in,
        output [15:0]   out,
        input                           clk,
        input                           reset_n
        );
```

Use the MegaFunction Wizard to create a new FIFO module that is 16-bits wide, 64 elements long and has an 'almost full' signal that is asserted when there are 63 elements in the FIFO. All other settings may remain at their defaults.

To turn this in to a MAV, we may use code such as the following:

```
wire [15:0] fifo_out;
wire fifo_full;
reg [23:0]     running_count;

fifo f(
        .clock(clk & reset_n),
        .data(in),
        .rdreq(fifo_full),
        .wrreq(1'b1),
        .almost_full(fifo_full),
        .q(fifo_out)
        );

wire [23:0] in_ext = {{8{in[15]}}, in[15:0]};
wire [23:0] out_ext = {{8{fifo_out[15]}},fifo_out[15:0]};

always @(negedge clk or negedge reset_n) begin
        if (!reset_n)
                running_count <= 0;
        else
                running_count <= running_count + in_ext - out_ext;
end
assign out = running_count / 64;
```

Remember to change the type of the 'fifo' instantiation to match the name you gave to the Wizard.

Note that, because we're using signed numbers, we have to 'sign extend' the input and output numbers before we can add them all together. This means that in order to do a 24-bit operation with 16-bit inputs, we have to make 8 copies of the most significant bit of each number and pack that together with the original value.

Also note that we use the 'almost full' output directly as a signal to determine whether or not we should start reading from the FIFO yet. If we always read, then the FIFO would never fill up and we would never get the delay that is required.

Now that the filter is complete, instantiate it inside the 'audio' module between the wave generators and the adio_codec and have it selectively enabled by the 'filter' input. That is, you may have an instantiation something like

```
assign wave = (filter ? filt : wave_1 + wave_2);

mav_filter     f1 (
                       .clk(wave_clock),
                       .reset_n(reset_n),
                       .in(wave_1 + wave_2),
                       .out(filt)
               );
```

Synthesize this and download it to the board. Listen to the result, turning the filter on and off, for different instruments. The 'ramp' instrument is particularly good for showing the effect of a low-pass filter as it has a lot of high frequency harmonics that the filter can attenuate.

Try a few different lengths of FIFO to change the tuning of the filter. Remember you will have to change not only the FIFO length in the wizard, but also the 'almost full' threshold. Finally remember to change the scaling factor in the filter's output assign statement.