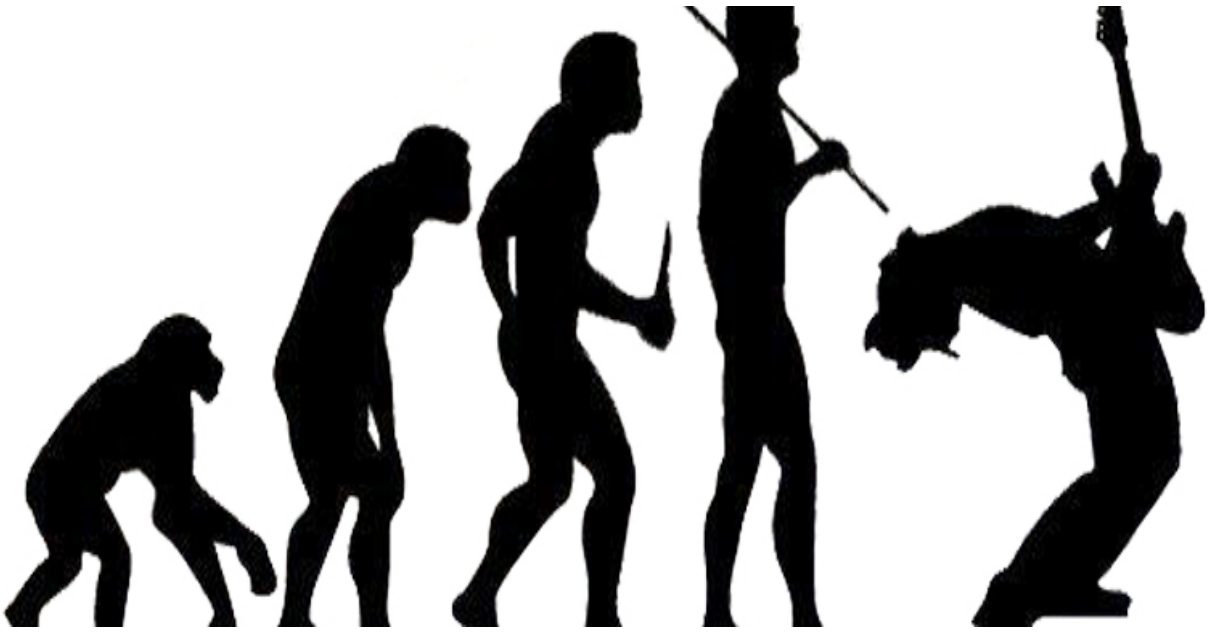
Introduction:

Guitar effects have been around for almost as long as the electric guitar itself. The first standalone effect unit (called "stomp-box", or "pedal") was built in 1948, and it was a Tremolo pedal. Since then countless effects were designed, created, re-created, modified and improved, taking the music industry by storm, creating a fertile ground for the development of different music genres, including the good old Rock 'n' Roll.
These stomp-boxes were (and many still are) analog – they consisted of discrete analog components, first vacuum-tubes and later transistors, and could create high quality audio effects, some of which we try to emulate to this very day by digital means.
In the 1980's the first digitized units started to appear, taking advantage of integrated circuits (IC's) and combining several effects units within a single unit. As digital technology increased in quality in the 1990's, better tools allowed more freedom and ease for engineers to experiment with different audio effects, eventually creating the all-in-one software based Multi-effect units which we can see today.



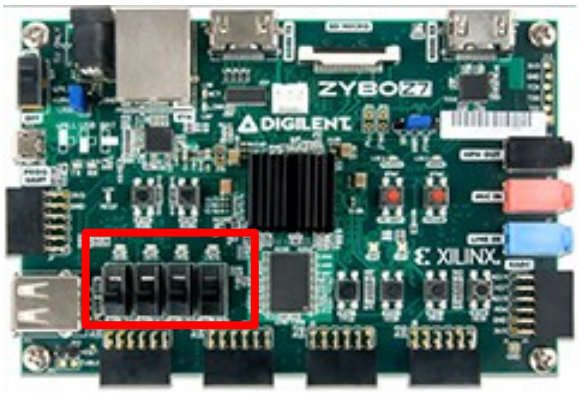
Project Goals :

In this project, we will experiment with the concept of implementing audio-processing algorithms on an FPGA, instead of the common CPU/DSP-based platforms.
This approach was inspired from a commercial vendor (Antelope Audio) products which integrated an FPGA logic into their audio effects platform .
Our main goals are:

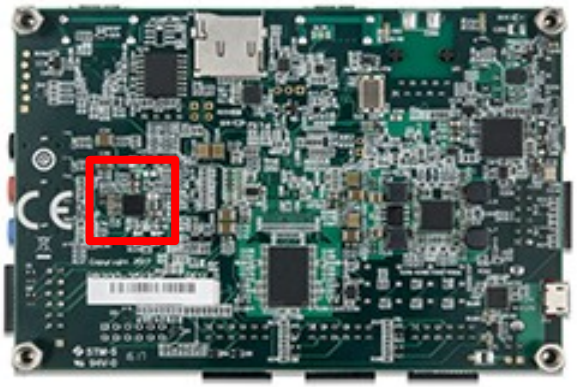
- Create an FPGA audio-processing platform and implement effects for the electric guitar, with a design resembling that of commercial physical Multi-effect pedals.
- Obtain knowledge and experience in VHDL and FPGA development in general, as this field grows in popularity in recent years.
- Put the students engineering skills and what we learned during the Zynq course, to create something practical which we will enjoy using.

Development platform :

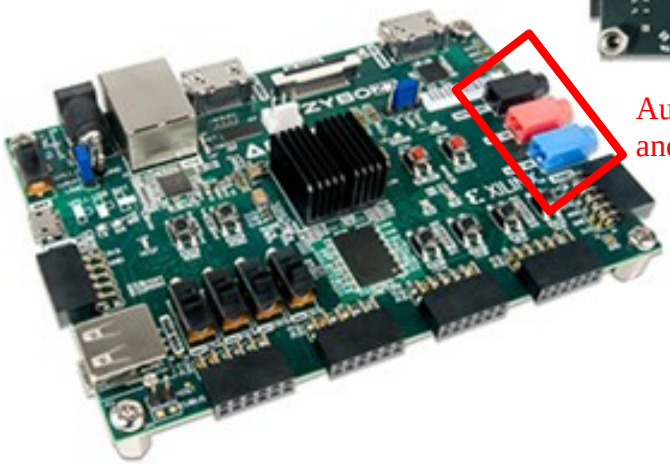
We will use the Zybo Zynq-7000 ARM/FPGA SoC Development Board - a development board for the Xilinx Zynq-7000 all programmable SoC (System-on-Chip), which comprises an ARM processing system (PS) and an FPGA programmable logic (PL). The board is equipped with a high-quality Audio-Codec, 3.5mm audio jacks and an abundance of buttons, switches, and LEDs, which serves all the needs of our project, in a single board.



SW and LEDs



Audio Codec



Audio input
and output

Protocols

Throughout the design, 3 well-known communication protocols will be used :

- AXI – for PS-PL interconnection
- I2C – to control the various registers of the on-board Audio-Codec
- I2S – to send digital audio to/from the Audio-Codec

AXI - (Advanced eXtensible Interface):

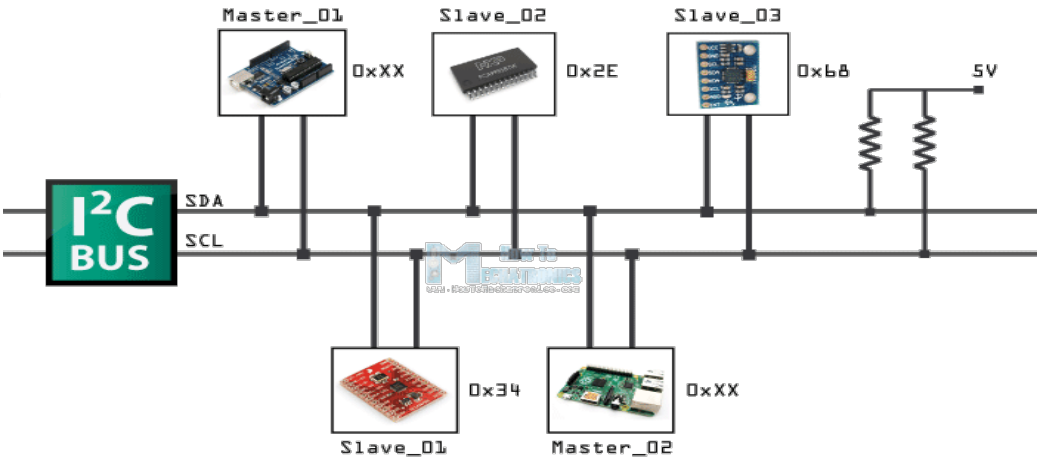
AXI is part of ARM AMBA (Advanced Microcontroller Bus Architecture), a family of micro controller buses first introduced in 1996. Many IP (Intellectual Property) providers support the AXI protocol, and a robust collection of third-party AXI tools is available, making it a standardized and convenient protocol for IP implementation.
There are three types of AXI4 interfaces:

AXI4—for high-performance memory-mapped requirements.
AXI4-Lite—for simple, low-throughput memory-mapped communication. It has a small logic footprint and is a simple interface to work with both in design and usage.
AXI4-Stream—for high-speed streaming data.

For the purpose of this project we will use the AXI4-Lite as it is simple and sufficient for our design. Each AXI IP has four 32-bit registers which can be written/read from the PS. The AXI protocol works in a master-slave mode, representing IP cores that exchange information with each other.

I2C - (Inter-integrated Circuit):

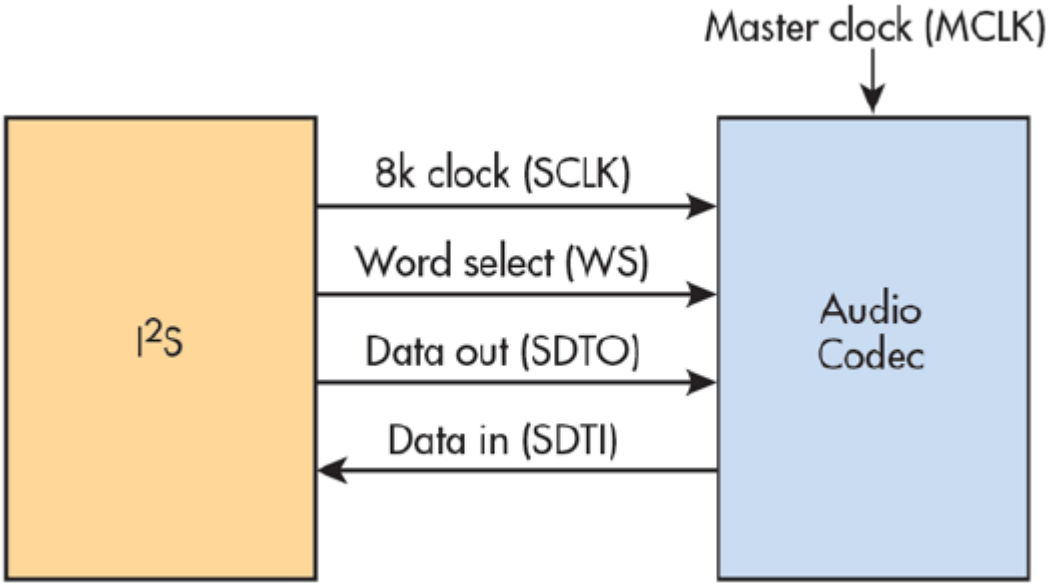
The Inter-Integrated Circuit (I2C) Protocol is a protocol intended to allow multiple “slave” digital integrated circuits to communicate with one or more “master” chips. I2C uses two bidirectional lines, Serial Data Line (SDA) and Serial Clock Line (SCL), and on the Zynq they are pulled up with 2kΩ resistors and a voltage of +3.3 V. We will use I2C to communicate with the Audio-Codec.



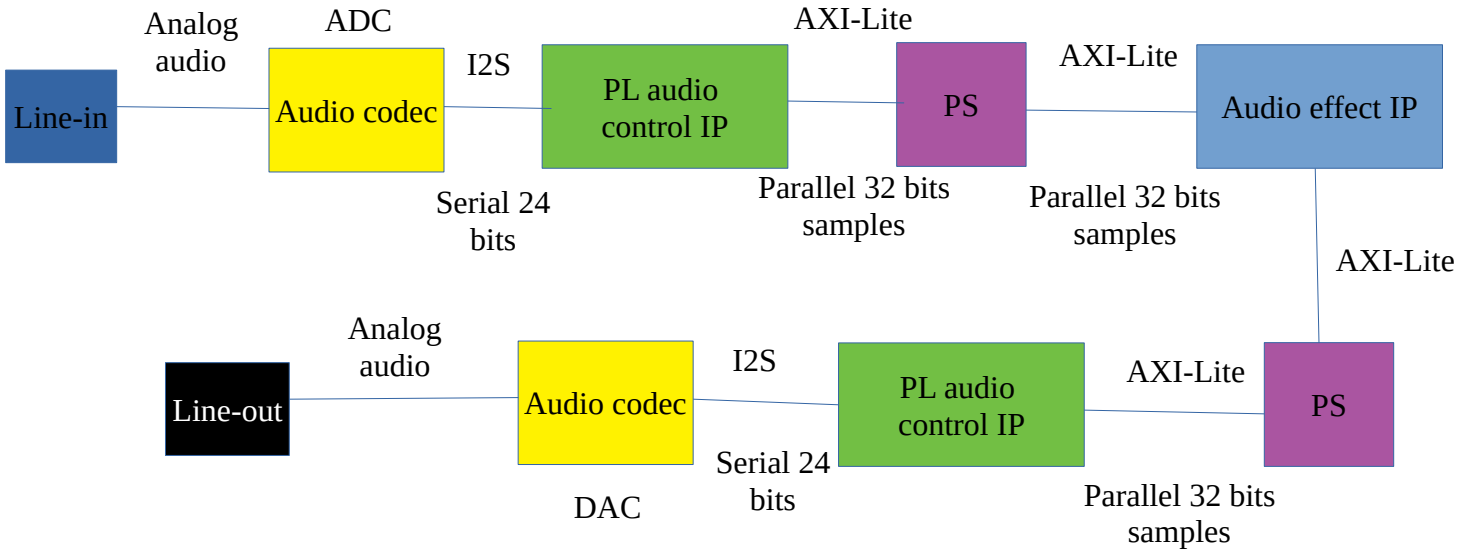
I2S - (Inter-IC Sound):

I2S (Inter-IC Sound), is an electrical serial bus interface standard used for connecting digital audio devices together. It is used to communicate PCM audio data between integrated circuits in an electronic device. The I2S bus separates clock and serial data signals, resulting in a lower jitter than is typical of communications systems that recover the clock from the data stream.
The bus consists of:

Bit clock line – officially called "continuous serial clock (SCK)" and labeled "bit clock" or BCLK.
Word clock line – officially called "word select (WS)" and labeled "left-right clock" or LRCLK.
Two multiplexed serial data lines – labeled SDATA_I and SDATA_O.
(Exact clock rates are described in the Audio-Codec section later).



Signal path :

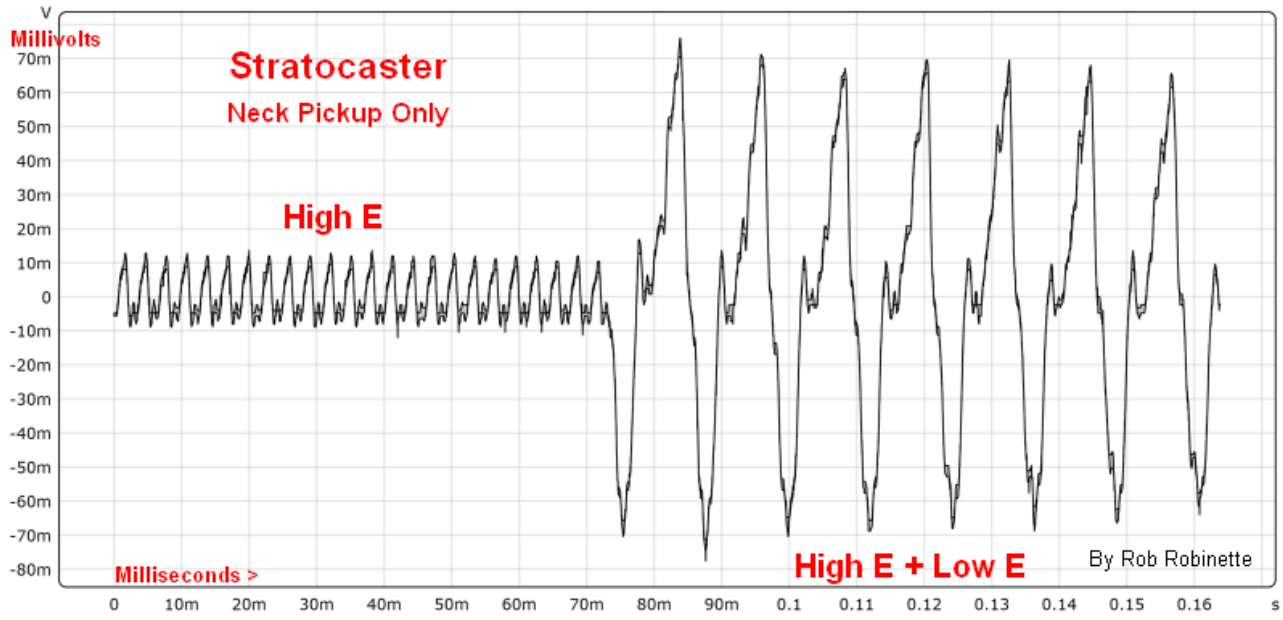


Generally the guitar pickups produces an output of 10-80mV, going into the Line-in 3.5mm jack. The analog audio is processed by the A/D on the Audio Codec at a sampling rate of 48.828Khz and 24 bits, outputting a serial stream of bits through the I2S protocol.

Then the serial data should be transfered Audio Control block, which creates parallel 32-bit samples (since the AXI peripherals in Vivado were designed for 32 bits), where the data is stored in the 24 LSB, padded from the left by 8 zeros, to complement the required 32-bit word. This IP also generates the LRCLK and BCLK required for the I2S protocol.
The 32-bit samples should be outputted via AXI to the Processing system(PS).

The PS should read the sample via AXI, and writes them back (via AXI), to the Audio effect IP, which should process the 32-bit audio sample and send them back to the PS.

The same thing that happened in the beginning should happens here in reverse - the sample goes from the PS to the Audio Control block where it's converted to serial, then via I2S to the codec's DAC, and out via the Line-out 3.5mm jack, to the guitar amplifier or headphones.



The design blocs :

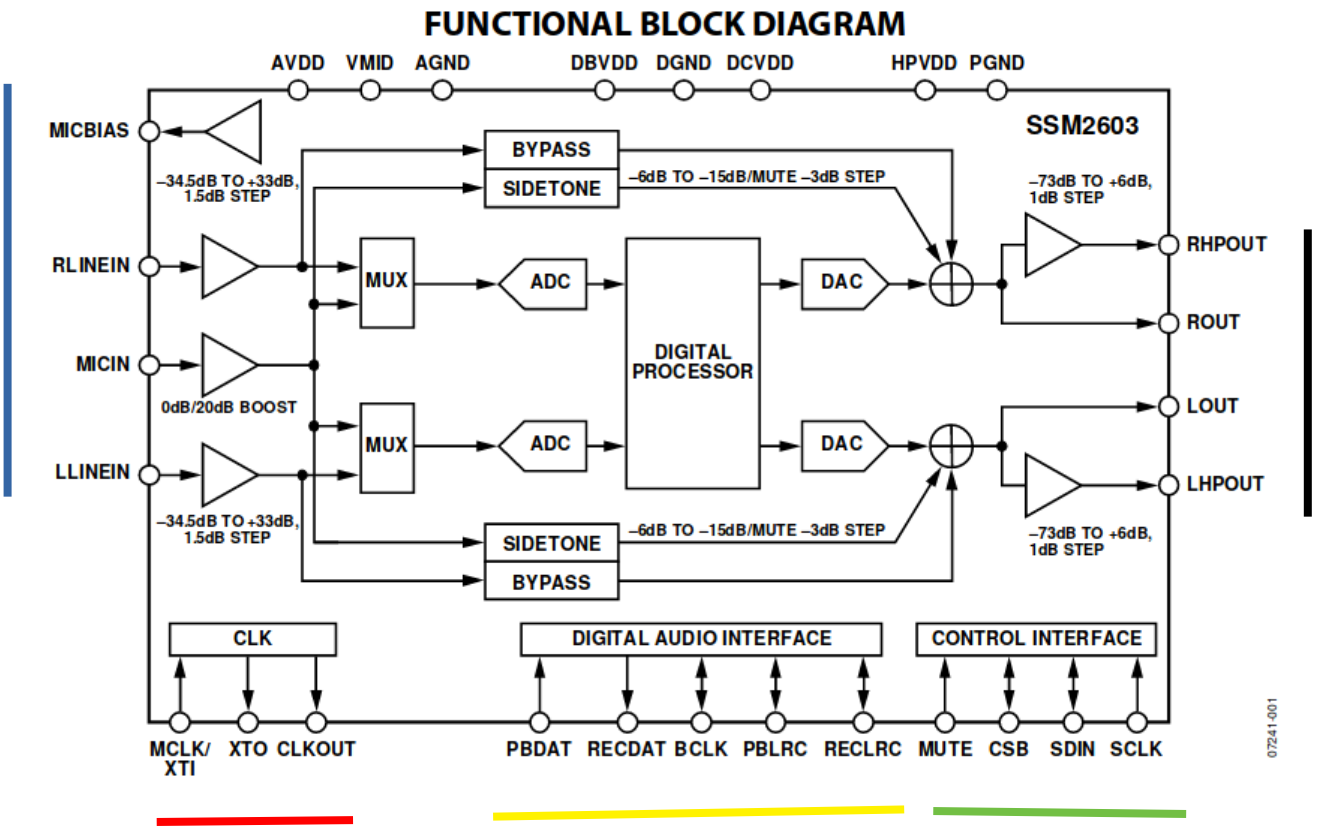
PL audio control IP :

this ip is delivered by digillient (the board manufucter), the is is responsible of interfacing our hardware audio codec to an AXI-lite interface the ip also genrates the required clock for our I2S interface that we're going to use to get our audio samples from the audio codec.

Analog-Devices SSM2603 Audio Codec

The Zybo board is equipped with an on-board Audio Codec, the SSM2603 by Analog Devices. It supports sampling rates from 8-96 Khz and 24-bit depth stereo audio ADC and DAC.
In our project, we will work at around 48Khz. It is more than twice the maximal human hearing limit of 20Khz, which is even better than CD quality (44.1Khz).
The Audio Codec is also the A/D and D/A, so a proper configuration is vital for good audio quality.

Below is a block diagram taken from the user manual, with some additional annotations:



Audio input

Audio output

Master clock source

I2S audio interface

I2C configuration interface

As can be seen in the block diagram above, the codec has various analog inputs and outputs, for the purpose of this project we will use only the left channel of the physical Line-in port, and duplicate the signal to left and right channels of the physical Line-out port, for a faster processing time.

At the bottom, there are 3 clock inputs - MCLK, RECLRC and BCLK, which are used by I2S for digital audio communication, and two serial ports – RECDAT(ADC) which is the output of the codec and the input of our system, and PBDAT(DAC) which is the input on the other end of the codec and the output of our system.
In addition, there are 2 ports for I2C – SCLK and SDIN which are the standard I2C lines.

Clocking:

As mentioned before, the I2S protocol requires the codec to work with 2 different clocks (BCLK and RECLRC), and the codec also requires a Master-clock input (MCLK).
Clocks for the converters and the serial ports are derived from the core clock. The core clock can be derived directly from MCLK or it can be generated by the an internal PLL.
The PLL can be bypassed or used, resulting in two different approaches to clock management .

The master clock formula is also register-based (with 4 predefined values) and we chose the highest multiplier of MCLK = 1024 x fs which reults fs= 50MHz/1024=48,828 kHz.

From the timing diagram in the user manual, it is possible to infer the required rates of BCLK and RECLRC :

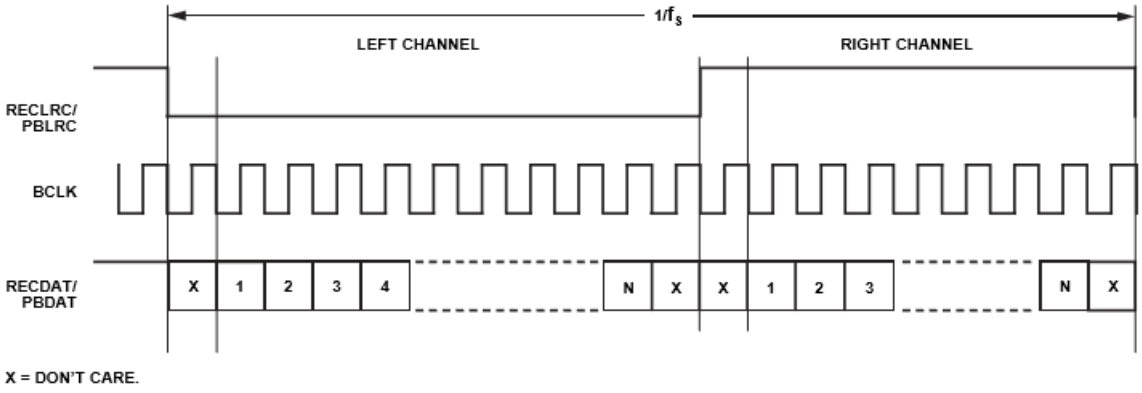


Figure 25. I²S Audio Input Mode

The bit clock pulses once for each discrete bit of data on the data lines. The number of BCLK cycles per audio frame is also configurable, and it can be 32 or 64. We will set it to 64, resulting in two 32-bit windows (for left and right channels), where the actual data is in 24 of the 32 bits, and the rest is zero-padding. Since RECLRC defines the audio frame, this will be our sampling rate, so - LRCLK = 48.828Khz. Now we need to fit 64 BCLK cycles into one RECLRC cycle, so – BCLK = 64 x RECLRC = 3.125Mhz.

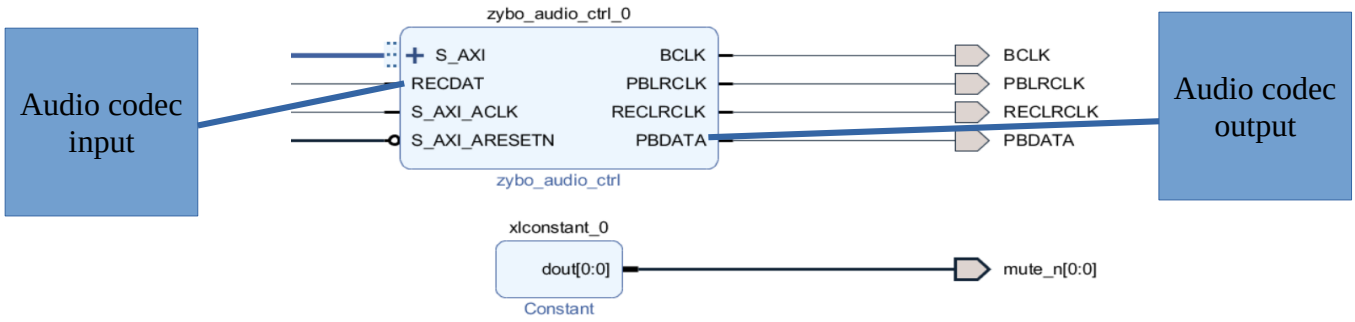
MCLK	50 MHz
RECLRC(LRCLK) / PBLRCLK	48,828 kHz
BCLK	3,125 MHz

TO DO List 1:

- Create a new project.

- Create a new block design and add the zybo audio control ip to your design (Check the wiki for the ip file).

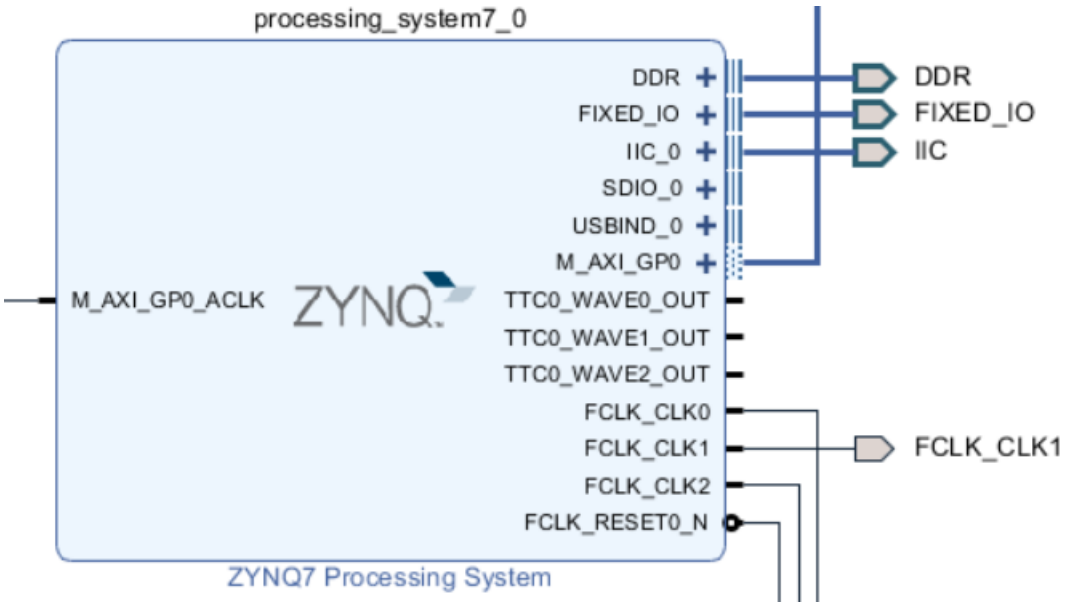
- Make RECDAT, BCLK, PBLRCLK, RECLRCLK, and PBDATA as external ports.



- Add the processing system to your design.

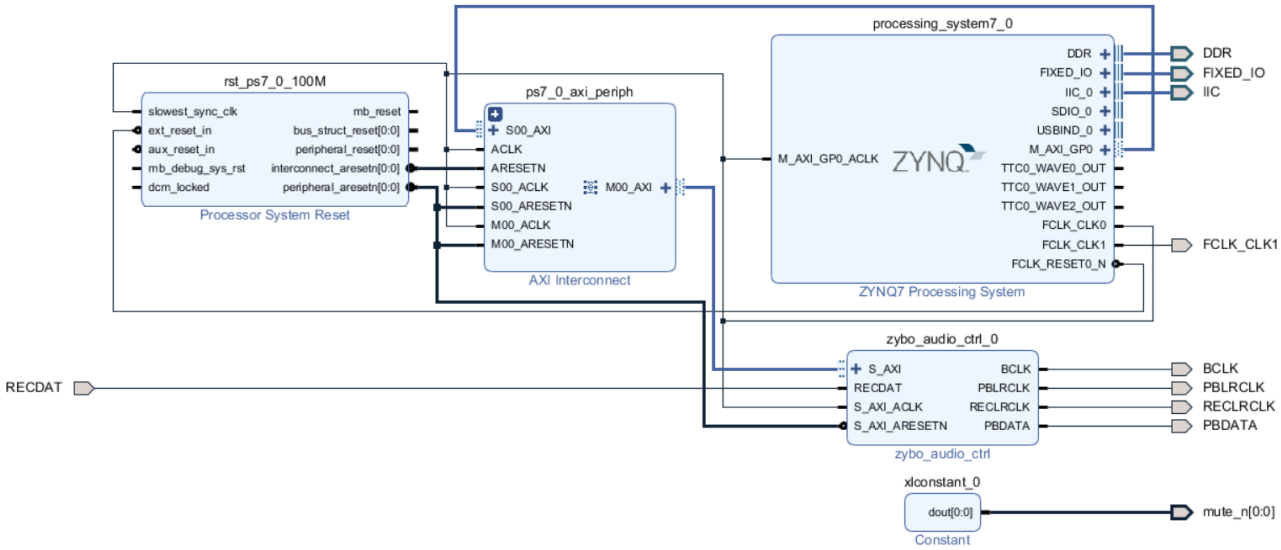
- Feed your MCLK with 50 MHz clock .

- Add the



constraints file (check the wiki for the file).

(Your final design should look something like this)



- Run synthesis and implementation.

- Export your hardware and lunch the SDK.

- Add the header file and the audio drivers to your project (Check the wiki for the files).

You may use some of the following functions in your software application:

Configure the I2C data structure:
IicConfig(XPAR_XIICPS_0_DEVICE_ID);

Configure the Audio Codec :
AudioPllConfig();

configure the UART registers:

```
u32 CntrlRegister;

CntrlRegister = XUartPs_ReadReg(UART_BASEADDR, XUARTPS_CR_OFFSET);

XUartPs_WriteReg(UART_BASEADDR, XUARTPS_CR_OFFSET,
                        ((CntrlRegister & ~XUARTPS_CR_EN_DIS_MASK) |
                        XUARTPS_CR_TX_EN | XUARTPS_CR_RX_EN));
```

function to send data through UART :

```
xil_printf("\r\n\r\n");
```

function to get data received through UART :

```
XUartPs_ReadReg(UART_BASEADDR, XUARTPS_FIFO_OFFSET);
```

Check if UART is receiving data :
XuartPs_IsReceiveData(UART_BASEADDR)

Read audio input from codec

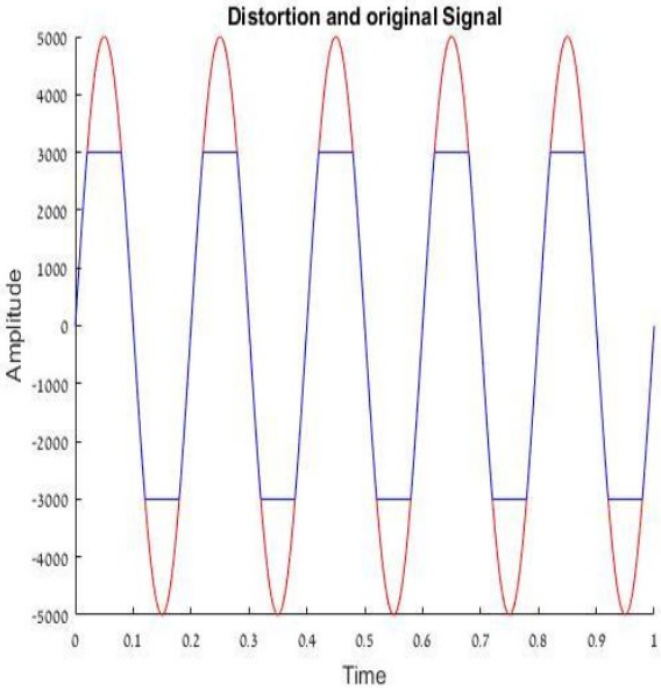
```
Xil_In32(I2S_DATA_RX_L_REG);
Xil_In32(I2S_DATA_RX_R_REG);
```

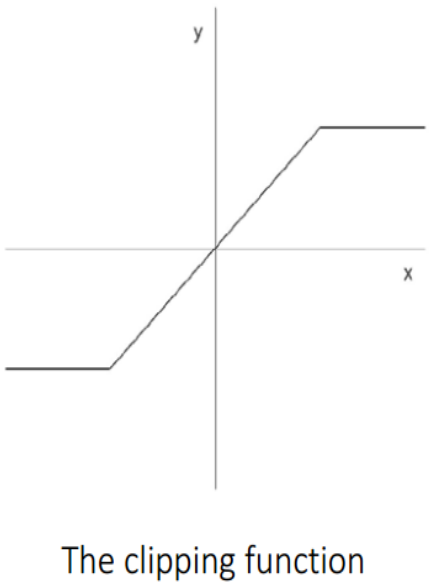
Write audio input to codec

```
Xil_Out32(I2S_DATA_TX_L_REG, Data);
Xil_Out32(I2S_DATA_TX_R_REG, Data);
```

Overdrive effect IP:

Overdrive effect (OD) belong to the family of non-linear effects.
Overdrive is a type of distortion, in which an amplifier is saturated (Overdrived) because of high-gain input (since both transistors and valves has a saturation point beyond a certain input gain). Digital OD wishes to emulate this process, by artificially creating a saturated wave-shape, by clipping the input signal above a certain threshold in the positive peaks, and below a certain threshold in the negative peaks. The level of clipping affects how distorted the sound is.
The signal is compered against 2 thresholds a positive and negative, and if the value exceeds one of them, it is clipped to the threshold's value. Otherwise the signal is passed unaffected. Below is a a graph that shows Overdrive effect on the signal:





The clipping function

The overdrive block should work with a faster clock than our acquisition clock so for this purpose let's use one of the PS CLK function to generate a **48 MHz** clock to feed our ip with it.

Depending on guitar signal these threshold was chosen

Effect type	Threshold	Operation
Weak overdrive	70000 and -70000	If in>= 70000 out = 70000 if in<= -70000 out = -70000
Regular overdrive	70000 and -70000	If in>= 70000 out = 50000 if in<= -70000 out = -50000
Strong overdrive	70000 and -70000	If in>= 70000 out = 90000 if in<= -70000 out = -90000

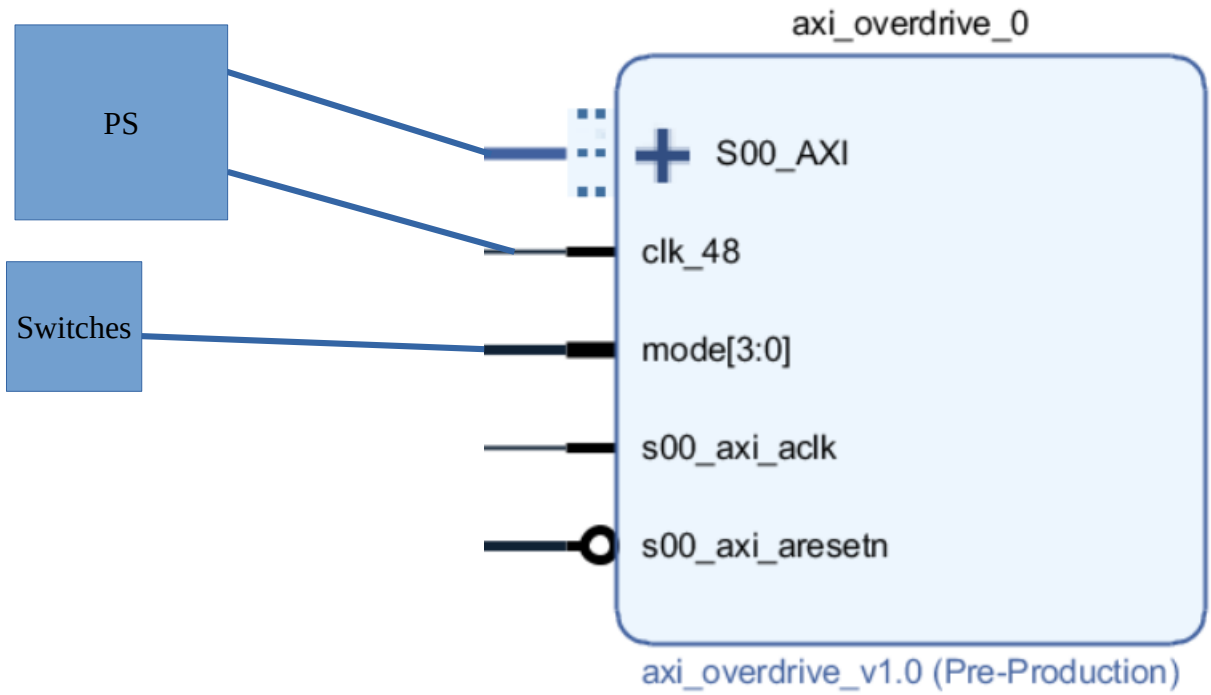
Rq: from the 32 bits we use only the first 24 bits because ou data is represented just in 24 bits, the rest of the bit is just used for axi-lite compatibility (the rest of the bits are null),

Example : (don’t forget to use use IEEE.NUMERIC_STD.ALL; library).

```
-----------------------------------------------------------------
--Weak overdrive
-----------------------------------------------------------------
if mode="001" then --weak overdrive
   if signed(audio_in(23 downto 0)) >= 70000 then
      audio_out<=std_logic_vector(to_signed(70000,32));
   elsif signed(audio_in(23 downto 0)) <= -70000 then
      audio_out<=std_logic_vector(to_signed(-70000,32));
   else
     audio_out<=audio_in;
end if;
   end if;
```

before interfacing your module through AXI Lite interface it should have the following ports:

audio_in	in STD_LOGIC_VECTOR(31 downto 0)	Sound samples input
audio_out	out STD_LOGIC_VECTOR(31 downto 0)	Sound samples output
clk_48	In STD_LOGIC	Clock source
mode	in STD_LOGIC_VECTOR(3 downto 0)	Select between strong weak and regular overdrive through the board switches
en	In std_logic	Bit to enable the effect bloc



TO DO List 2:

- Create and test the overdrive effect hdl bloc.

- use simulation to validate your bloc behavior.

- Use the ip creator tool to interface your module with AXI Lite interface.

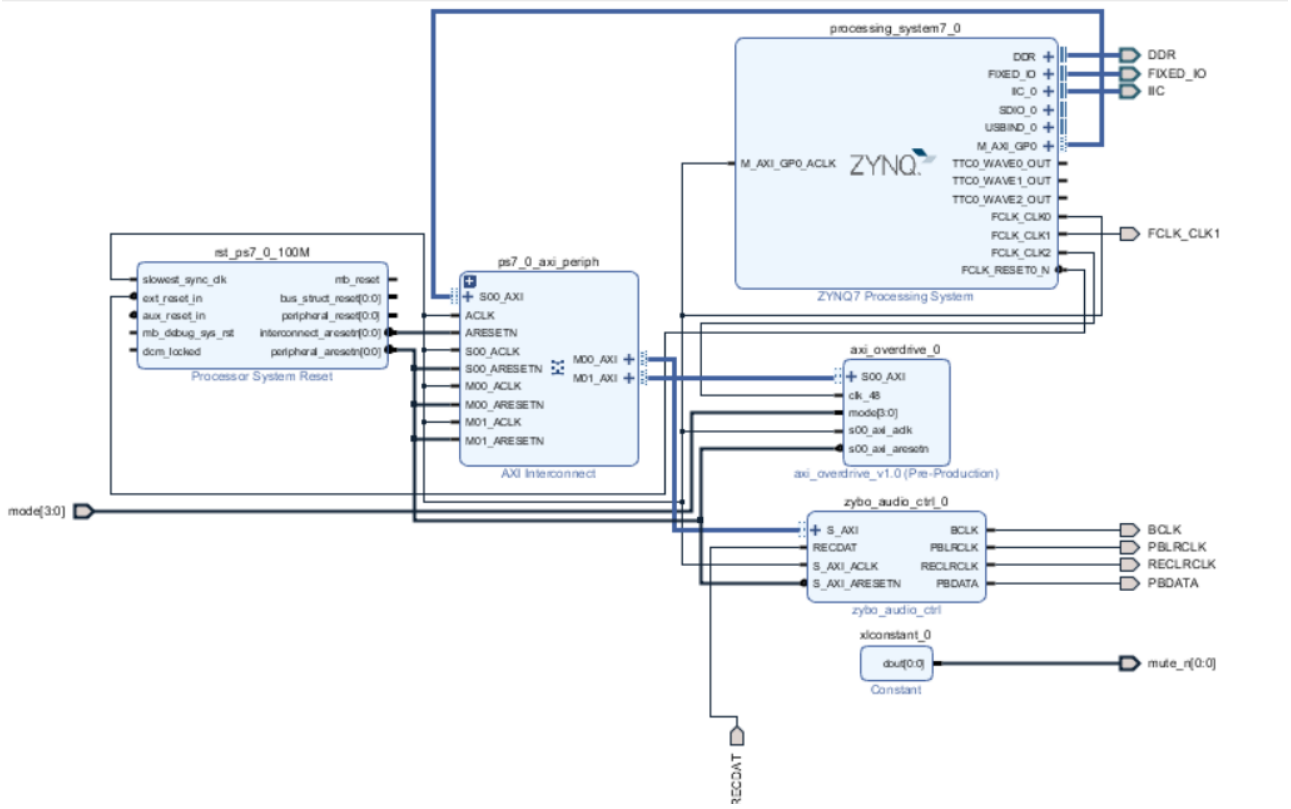
- add Your developed IP to the previous design and make the necessary connections for the IP.

- Run synthesis and implementation.

- Export your design and lunch the SDK.

- Write a software application.

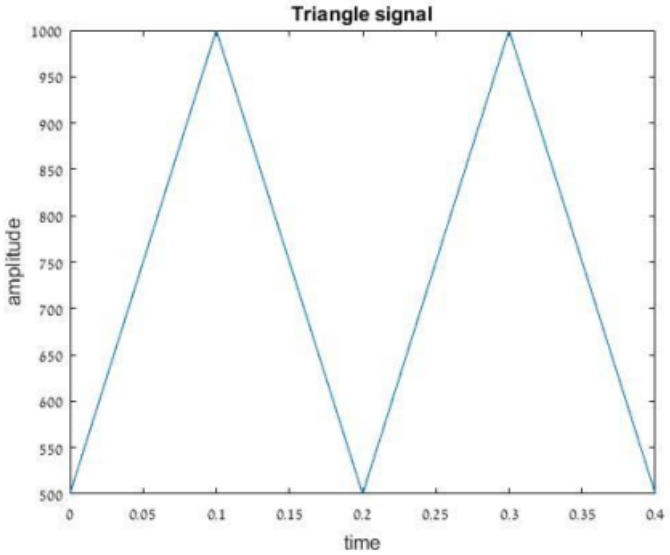
(your final design should look something like this)



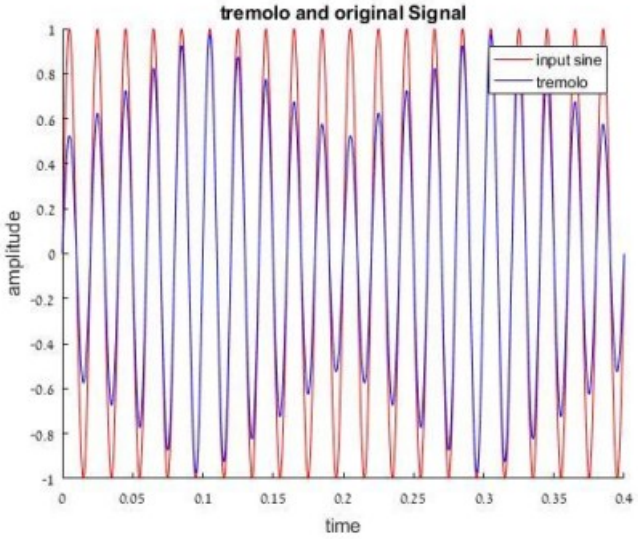
Optional parts:

Tremolo effect:

Tremolo is basically Amplitude Modulation (AM). The effect is generated by multiplying the audio with a modulating signal, which changes the envelope of the modulated signal. In VHDL the easiest way is to create a triangular wave, and multiply it by the input signal. The result is a triangular envelope on the input signal.

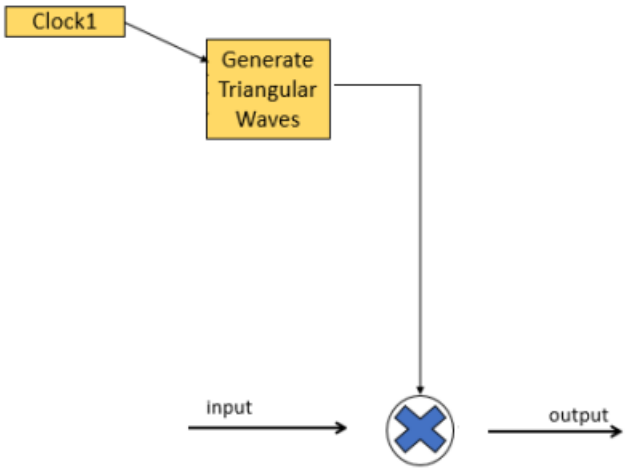


Triangle wave



Modulated sine wave

The triangle wave is implemented by a counter, which counts to a certain value, and then back down. Setting the count value to 30, a full period means 60 clock cycles. So, by calculating how many periods are required for the triangular wave (the tremolo's frequency), it is easy to determine which clock is required to produce this frequency.



Example:

Clock frequency in Hz	Tremolo frequency Hz
48	48/60 = 0,8
190	190/60 = 3,2
380	380/60 = 6,35

The MCLK clock source at 50Mhz is used for the codec, we could use it in our module and divide it into slower clocks.
The division could be done by powers of 2, with a 32-bits counter which counts at 50Mhz, and we can generate various clocks by accessing different bits in the counter.

Example: clock generation

50 MHz / 2^{17} = 381 Hz

```
count: process(clk_in)

begin

   if clk_in='1' and clk_in'event then

      clk_cntr <= clk_cntr+1;

   end if;

end process;


--****** output clock  **********

clk_380hz <= clk_cntr(16); --380Hz clock

clk_190hz <= clk_cntr(17); --190Hz clock

clk_48hz <= clk_cntr(19); --48Hz clock

--*****************************
```

Example: triangular wave generation

```
process(count_trem) --tremolo frequency 380hz

begin

   if (count_trem=30) then

      direction <= '1';

   end if;
```

```
    if (count_trem=1) then
        direction <= '0';
    end if;
end process;
process(direction, clk_380hz)
begin
if rising_edge(clk_380hz) then
    if (direction='0') then
    count_trem <= count_trem+1;
    end if;
    if (direction='1') then
    count_trem <= count_trem-1;
end if;
end if;
end process;
--**************************************
```

At the end you should assign the value of the multiplication between the input signal and with each value of the triangular wave.

```
y <= std_logic_vector((signed(x))*count_trem);
```

TO DO List 3:

- Create and test the tremolo effect hdl bloc.

- Use simulation to validate your bloc behavior.

- Use the ip creator tool to interface your module with AXI Lite interface.

- Add Your developed IP to the previous design and make the necessary connections for the IP.

- Run synthesis and implementation.

- Export your design and lunch the SDK.

- Write a software application to test your design.

User interface:

The user interface part consists of creating a menu that will be printed through the UART to choose between the a function mode :

- stream the pure audio.
- apply the overdrive effect on audio input.
- apply the tremolo effect on audio input.

Character ‘s’	Stream pure audio
Character ‘d’	Apply overdrive effect
Character ‘t’	Apply tremolo effect
Character ‘q’	Quit the mode and back to the main menu

Also a module to control leds should be used for visibility.

- light LED[0] when the system is running.
- light LED[1] when streaming pure audio.
- light LED[2] when applying overdrive effect.
- light LED[3] when applying Tremolo effect.

The user interface should looks something like this on a serial terminal: