ELE748-01
Architecture des systèmes ordinés et VHDL

Rapport final projet #1 :

Pédale à effet pour guitare

Par :
Vincent Gosselin GOSV16129208
Carl Trudeau TRUC28029008

Présenté à :

Simon Pichette

Fait le : 30 Juillet 2017

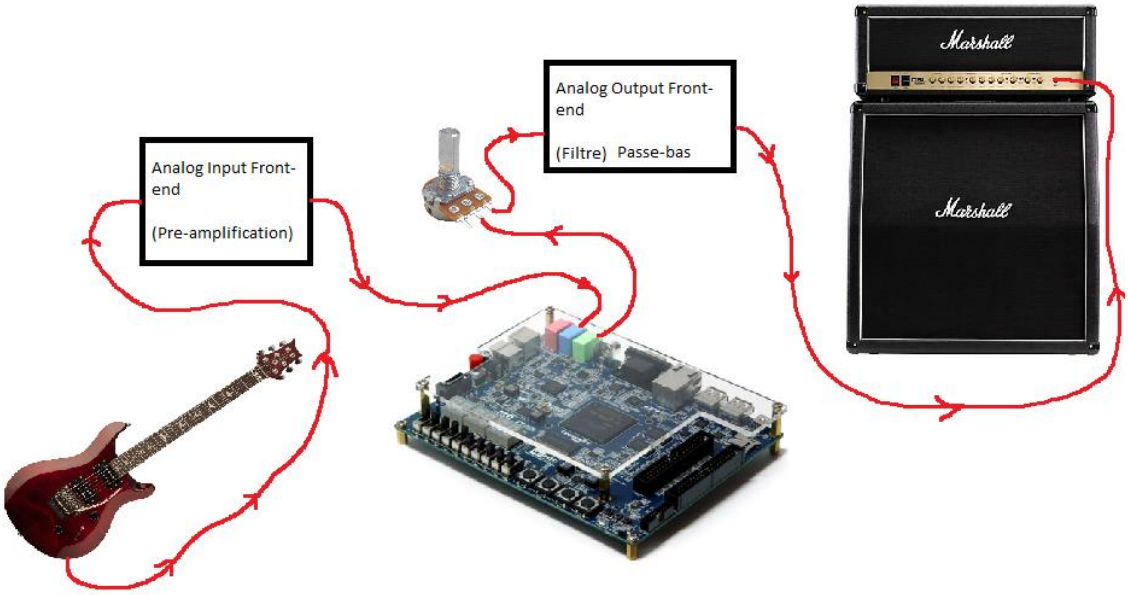
Introduction

L'équipe a décidé de s'aventurer dans la conception d'une pédale de guitare à l'aide de la carte DE1-SoC. Le projet consiste à rentrer un signal de guitare (pré-amplifier) dans le port ADC de la puce audio Wolfson WM8731, effectuer le traitement de signal dans le FPGA à l'aide du Nios2 et ensuite de faire sortir le signal traité par la sortie DAC de la même puce audio. Le traitement de signal se fait en langage C à l'aide de Eclipse Software Building Tools for Nios2.

Présentation du projet

La réalisation de la pédale de guitare se fait sur trois différents niveau : la partie analogique (front-end input/output), la partie hardware dans le FPGA et la partie software en C dans le Nios II. Tous les effets ont été produit en software et non avec des accélérateurs matériels.

Vue d'ensemble



Note : La *pré-amplification* devra avoir un gain de 20 pour être dans le range possible minimum de l'ADC du chip audio puisque le signal de sortie de la guitare est de 80mV. Le chip audio a aussi la possibilité d'amplifier le signal sur le Line Input mais le gain maximal est de 12db soit une multiplication du voltage par 4, ce qui ne sera pas assez. Le *master volume* consiste à un potentiomètre qui réduit l'amplitude du signal vers l'analogue output. Il sert de protection pour ne pas faire sauter l'ampli si jamais le signal sortant de la carte DE1-SoC est trop élevé en amplitude.

Liste des fonctionnalités

Il y aura 4 canaux différents sur la pédale. Le canal 0 étant 'aucun effet'. Le signal est seulement entré par le port ADC et sortie tel quel sur le port DAC de la puce audio. Aucun traitement numérique ce fait sur ce canal. Le canal 1 est la distorsion. Le canal 2 est pour le délai et le canal 3 est pour l'octavier. Le contrôle des effets se fait l'aide des 4 boutons de la carte. L'affichage du canal actuel se fait par l'entremise des 7-segments disponibles sur la carte.

Description du fonctionnement

Une fois la carte DE1-SoC programmé, l'effet présent par défaut est 'aucun effet'. L'utilisateur n'a qu'à appuyer sur les différents boutons de la carte pour changer entre les différents éffets.

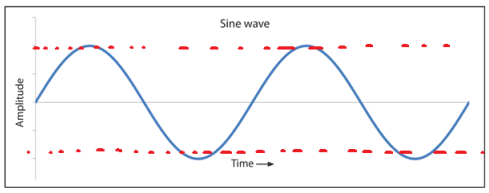
Description détaillée de votre démarche de conception

L'équipe a commencé par la conception au niveau hardware pour connecter le FPGA à la puce audio WM8731 sur la carte. L'objectif #1 du projet était de faire fonctionner le mode 'aucun effet'. Pour le faire fonctionner, il fallait donc que l'architecture hardware dans Qsys et qu'un code en C soit correct. Nous avons découvert qu'il y avait déjà des IP core pour communiquer avec la puce audio : Audio IP core, Audio and Video Config, Audio PLL.

D'autre part, l'équipe s'est aventuré dans le design électrique d'un front end analogue pour interfacé avec la guitare électrique et l'amplificateur de guitare. Avec des recherches avancés, l'équipe a choisi d'utiliser le front end *PedalShield Due* [1]pour l'interface avec la guitare et l'amplificateur. Le *PedalShield Due* contient un analogue input front-end et un analogue output front-end. Le côté analogue input front-end se charge d'amplifier le signal par un gain de 20 et fait sortir le signal 0-3.3V sur une pin (prêt pour être injecté dans un ADC). Le côté analogue output front-end s'agit de prendre le signal (venant d'un DAC) et d'appliquer un filtre passe-bas sur celui-ci. Il y a deux connecteurs ¼ pouces pour interfacer avec des câbles de guitare.

Description détaillée de votre démarche de réalisation

L'équipe a débuté par le mode 'aucun effet'. La plupart du temps de debugging du projet était de faire fonctionner ce mode, c'était la barrière principale de ce laboratoire. Après avoir réussi, l'équipe avait initialement envisagé de faire l'effet de distorsion en coupant les pointes du signal :



Il a donc décidé que l'effet de distorsion allait avoir un accélérateur matériel pour effectuer ce traitement de signal. Après des tests au niveau software, cette solution de couper le signal avec une limite inférieure et une limite supérieur donnait des résultats médiocres quand le son était émis par des écouteurs (présence de *'pop'* dangereux pour les oreilles). Une deuxième solution a été envisagé en software mais n'a pas été implémenté en accélérateur matériel. Après avoir eu la distorsion fonctionnelle, l'équipe à poursuivit avec le délai et ensuite l'octavier en software. Quand tous les effets ont été fonctionnel, le circuit électrique global a été réaliser pour connecter l'analogue front-end à la carte DE1-SoC.

Division des tâches au sein de l'équipe

Vincent Gosselin : Qsys, VHDL top level, code en C Nios II pour 'aucun-effet', distorsion, délai, octavier, programme principale, conception/réalisation électrique.

Carl Trudeau : Accélérateur matériel pour distorsion.

---

[1] http://www.electrosmash.com/pedalshield

Architecture du système matériel

Une implémentation dans Qsys a déjà été réalisé pour ce qui est de l'architecture du système matériel. Il est possible que celle-ci change d'ici la fin du projet. Le texte suivant décrit l'architecture actuelle du système matériel dans Qsys ainsi que du top-level en VHDL du système.

Description détaillée du système

Le système actuel contient les modules suivant : *Clock Source, On-Chip Memory RAM, Nios II processor, System ID Peripheral, 3x PIO (Parallel I/O), Audio, Audio and Video Config, 1x Audio Clock for DE-series Boards, Clock-Bridge* et enfin un *JTAG UART*. Le module *Clock Source* sert à donner une horloge de 100MHz au système. La mémoire *On-Chip* a une taille de 300KBytes pour emmagasiner les instructions compilées à l'aide de Eclipse Software Building Tools for Nios2. En addition, cette mémoire sera utilisée pour le traitement de signal comme l'effet de délai qui est demandant en mémoire. Le cœur du système est encore une fois le *Nios II processor*. D'autre part, le *System Id Peripheral* sert à donner une authentification unique du hardware pour que la programmation en C du NIOS 2 soit compatible. Ensuite, *1x PIO (Parallel I/O)* sert pour les boutons afin de sélectionner les effets et *2x PIO (Parallel I/O)* sert pour afficher l'effet actuelle sur les 7-segments disponibles sur la carte. Le module *Audio* sert à communiquer avec *l'interface audio* de la puce audio, WM8731. Ce module *Audio* est physiquement connecté aux pins *ADCDAT, ADCLRCK, BCLK, DACDAT* et *DACLRCK* du WM8731. La transmission des données audio passe entièrement par ce module. Ce module nécessite une horloge de 12.288MHz afin que l'acquisition de signal par le puce audio ce fait à 32KHz. D'autre part, le module *Audio and Video Config* sert à configurer le WM8731 via la communication sériel I^2C. Il y a donc 2 pins de connecté sur le WM8731 : le signal SDA et le signal SCLK. Présentement, ce module donne une configuration de base à la puce audio. La configuration de base est : *LINE In to ADC, Audio Out – Line In Bypass, Data Format Left Justified, Bit lenght : 16* et *Sampling Rate : 32KHz.* Cette configuration initiale permet de tester qu'un sinus entre et sort de la carte DE1-SoC sans toucher l'ADC et le DAC. Il est à noter que *Audio and Video Config* prend une horloge de 100MHz. Ensuite, *2x Audio Clock for DE-series Boards* sont utilisé pour générer les horloges à 12.288MHz. Étrangement, le module *Audio* n'avait pas de sortie pour le signal MCLK/XCK (Master Clock) pour la puce audio. C'est pour cette raison que notre architecture contient 1 horloge de 12.288Mhz ainsi qu'un *Clock-Bridge* pour le module *Audio* et pour la pin MCLK. Puis enfin, le module *JTAG UART* est utilisé pour afficher des messages (à l'aide de la fonction *printf*) sur la console Nios 2.

Le Audio IP core d'Altera fonctionne avec le principe de FIFO (first in/first out). C'est avec cet IP core qu'on communique avec le puce audio. Il y a 2 types de FIFO : des FIFO entrantes et des FIFO sortantes. Les FIFO ont une taille de 128 Bytes chacune. Il y a des FIFO pour le canal gauche et droit afin d'obtenir un son en stereo.

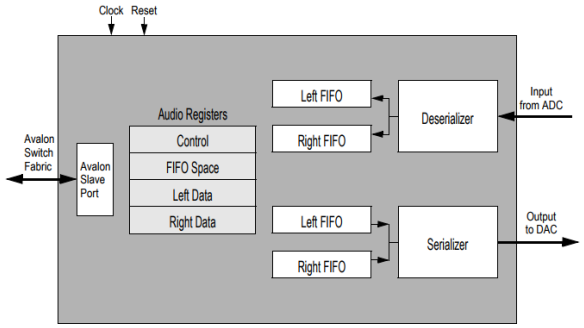
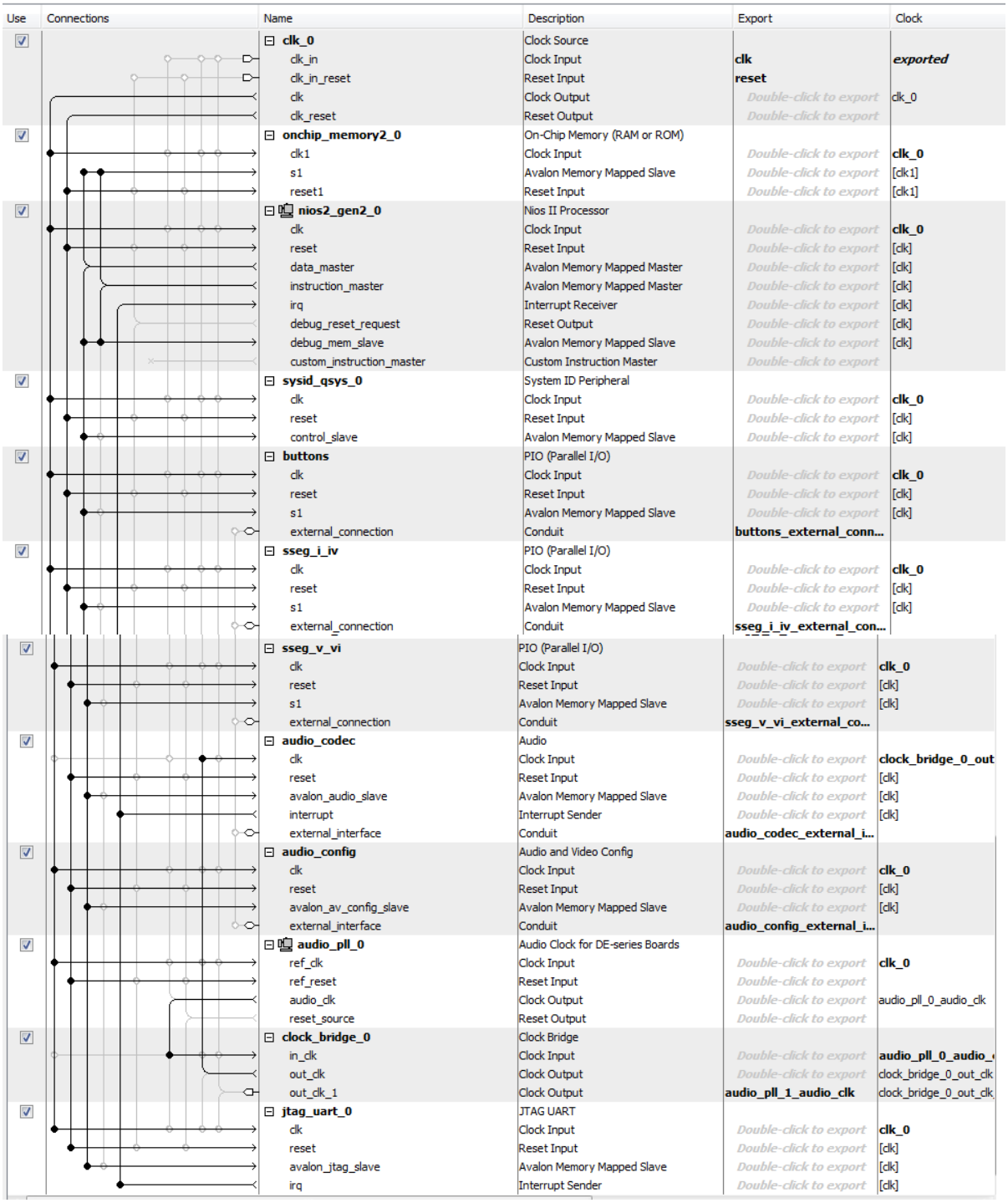


Figure 1. Block diagram for Audio core with Memory-Mapped Interface

Configuration présente dans Qsys



Use	Connections	Name	Description	Export	Clock
☑		**clk_0**	Clock Source		
		clk_in	Clock Input	**clk**	***exported***
		clk_in_reset	Reset Input	**reset**	
		clk	Clock Output	*Double-click to export*	clk_0
		clk_reset	Reset Output	*Double-click to export*	
☑		**onchip_memory2_0**	On-Chip Memory (RAM or ROM)		
		clk1	Clock Input	*Double-click to export*	**clk_0**
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]
		reset1	Reset Input	*Double-click to export*	[clk1]
☑		**nios2_gen2_0**	Nios II Processor		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		data_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]
		instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]
		irq	Interrupt Receiver	*Double-click to export*	[clk]
		debug_reset_request	Reset Output	*Double-click to export*	[clk]
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
		custom_instruction_master	Custom Instruction Master	*Double-click to export*	
☑		**sysid_qsys_0**	System ID Peripheral		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		control_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
☑		**buttons**	PIO (Parallel I/O)		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
		external_connection	Conduit	**buttons_external_conn...**	
☑		**sseg_i_iv**	PIO (Parallel I/O)		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
		external_connection	Conduit	**sseg_i_iv_external_con...**	
☑		**sseg_v_vi**	PIO (Parallel I/O)		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
		external_connection	Conduit	**sseg_v_vi_external_co...**	
☑		**audio_codec**	Audio		
		clk	Clock Input	*Double-click to export*	**clock_bridge_0_out**
		reset	Reset Input	*Double-click to export*	[clk]
		avalon_audio_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
		interrupt	Interrupt Sender	*Double-click to export*	[clk]
		external_interface	Conduit	**audio_codec_external_i...**	
☑		**audio_config**	Audio and Video Config		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		avalon_av_config_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
		external_interface	Conduit	**audio_config_external_i...**	
☑		**audio_pll_0**	Audio Clock for DE-series Boards		
		ref_clk	Clock Input	*Double-click to export*	**clk_0**
		ref_reset	Reset Input	*Double-click to export*	
		audio_clk	Clock Output	*Double-click to export*	audio_pll_0_audio_clk
		reset_source	Reset Output	*Double-click to export*	
☑		**clock_bridge_0**	Clock Bridge		
		in_clk	Clock Input	*Double-click to export*	**audio_pll_0_audio_**
		out_clk	Clock Output	*Double-click to export*	clock_bridge_0_out_clk
		out_clk_1	Clock Output	**audio_pll_1_audio_clk**	clock_bridge_0_out_clk
☑		**jtag_uart_0**	JTAG UART		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
		irq	Interrupt Sender	*Double-click to export*	[clk]

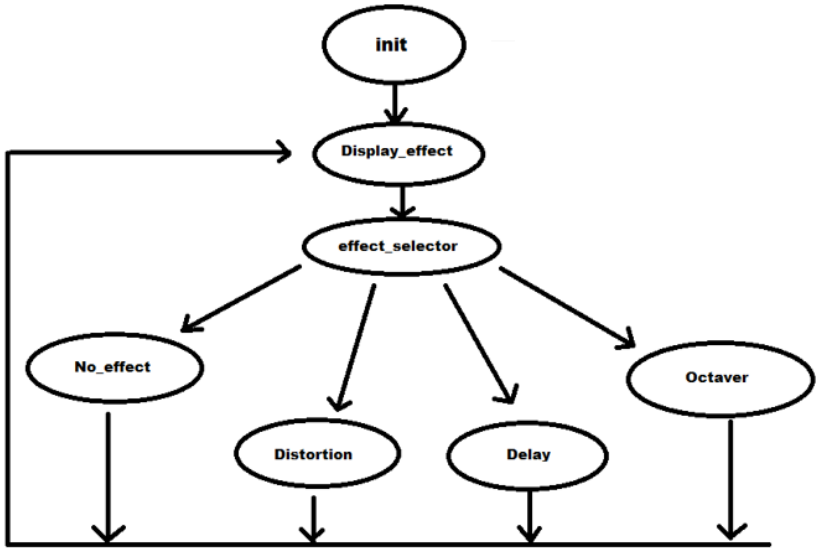
Code VHDL top-level

```vhdl
--top level
-- Par Vincent Gosselin


library ieee;
use ieee.std_logic_1164.all;

entity projet_1 is
    port(clock_50 : in std_logic;
         hex0  : out std_logic_vector(6 downto 0);
         hex1  : out std_logic_vector(6 downto 0);
         hex2  : out std_logic_vector(6 downto 0);
         hex3  : out std_logic_vector(6 downto 0);
         hex4  : out std_logic_vector(6 downto 0);
         hex5  : out std_logic_vector(6 downto 0);
         key : in std_logic_vector(3 downto 0);

         --for audio codec
         --Where is AUD_XCK??
         AUD_XCK : out std_logic;

         AUD_ADCDAT    : in std_logic;
         AUD_ADCLRCK   : in std_logic;
         AUD_BCLK      : in std_logic;
         AUD_DACDAT    : out std_logic;
         AUD_DACLRCK  : in std_logic;
         FPGA_I2C_SDAT : inout std_logic; -- SDA connected to Audio chip.
         FPGA_I2C_SCLK : out std_logic -- SCLK connected to Audio chip.


        );
    end projet_1;

architecture structural of projet_1 is

component system is
        port (
            audio_codec_external_interface_ADCDAT  : in    std_logic                     := 'X';             -- ADCDAT
            audio_codec_external_interface_ADCLRCK : in    std_logic                     := 'X';             -- ADCLRCK
            audio_codec_external_interface_BCLK    : in    std_logic                     := 'X';             -- BCLK
            audio_codec_external_interface_DACDAT  : out   std_logic;                                        -- DACDAT
            audio_codec_external_interface_DACLRCK : in    std_logic                     := 'X';             -- DACLRCK
            audio_config_external_interface_SDAT   : inout std_logic                     := 'X';             -- SDAT
            audio_config_external_interface_SCLK   : out   std_logic;                                        -- SCLK
            audio_pll_1_audio_clk_clk              : out   std_logic;                                        -- clk
            buttons_external_connection_export     : in    std_logic_vector(3 downto 0)  := (others => 'X'); -- export
            clk_clk                                : in    std_logic                     := 'X';             -- clk
            reset_reset_n                          : in    std_logic                     := '1';             -- reset_n
            sseg_i_iv_external_connection_export   : out   std_logic_vector(31 downto 0);                    -- export
            sseg_v_vi_external_connection_export   : out   std_logic_vector(31 downto 0)                     -- export
        );
    end component system;

begin
    nios_system : system
    port map ( clk_clk => clock_50,
                  -- pour Aud_xck, non fourni par Audio Core ...
                audio_pll_1_audio_clk_clk => AUD_XCK,

                audio_codec_external_interface_ADCDAT => AUD_ADCDAT,
                audio_codec_external_interface_ADCLRCK => AUD_ADCLRCK,
                audio_codec_external_interface_BCLK => AUD_BCLK,
                audio_codec_external_interface_DACDAT => AUD_DACDAT,
                audio_codec_external_interface_DACLRCK => AUD_DACLRCK,
                audio_config_external_interface_SDAT => FPGA_I2C_SDAT,
                audio_config_external_interface_SCLK => FPGA_I2C_SCLK,

                buttons_external_connection_export => key(3 downto 0),
                sseg_i_iv_external_connection_export(6 downto 0) =>  hex0,
                sseg_i_iv_external_connection_export(14 downto 8) => hex1,
                sseg_i_iv_external_connection_export(22 downto 16) => hex2,
                sseg_i_iv_external_connection_export(30 downto 24) => hex3,
                sseg_v_vi_external_connection_export(6 downto 0) =>  hex4,
                sseg_v_vi_external_connection_export(14 downto 8) => hex5
               );

end structural;
```

Architecture logicielle

Description du logicielle

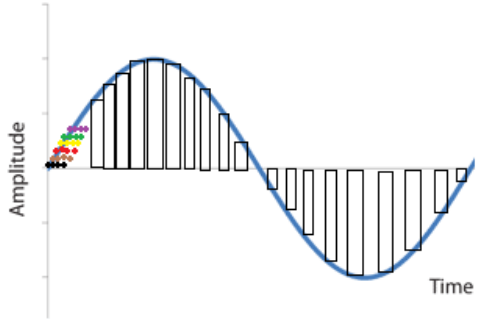
Le logicielle contient les fichiers suivant : *main.c, audio_driver.h, button_reader.h, sseg_driver.h*. Le fichier *main.c* contient la boucle infinie qui est le programme principal. Voici la machine à état qui compose le programme principal :



L'état *init* permet d'initialiser la puce audio pour la configuration de base. Par défaut, c'est l'effet *'aucun-effet'* qui est sélectionné de base. Ensuite, l'effet actuellement choisi est affiché sur les 7-segments de la carte. Puis, la sélection du prochain effet se fait dans l'état *effect_selector*. Cet état va lire le registre EDGE capture des boutons pour déterminer le prochain effet. Le fichier *button_reader.h* servira pour interfacer avec les boutons de la carte tandis que *sseg_driver.h* sera pour afficher l'effet actuelle sur les 7-segments. C'est dans le fichier *audio.driver.h* que tous les effets seront implémentés ainsi que l'initialisation de la puce WM8731. L'effet *'aucun-effet'* n'a pas de traitement de signal associé, il suffit de convertir le signal entrant avec l'ADC et de l'émettre sur le DAC.

Stratégie de l'effet de Distorsion software

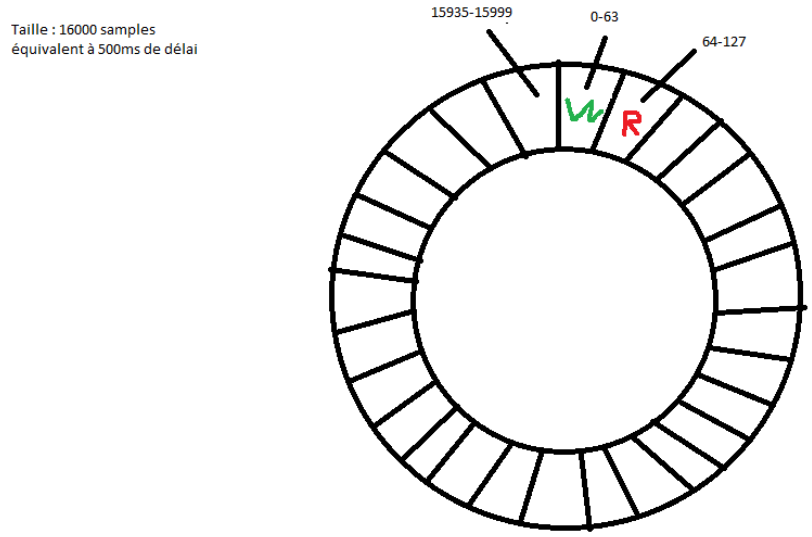
Chaque point échantillonné par l'ADC du WM8731 est écrit 4 fois dans les FIFO sortantes. Cela a comme effet d'introduire de la distorsion au son sans aucun danger pour les oreilles.



Stratégie de l'effet de Délai software

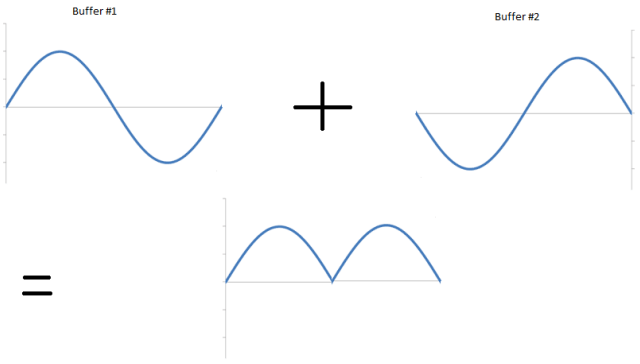
La stratégie est d'enregistrer 16000 échantillons (words) à coup de 64 words dans un buffeur circulaire en ayant toujours l'**index du reading** vers l'output FIFO en avance de l'**index du writing** de l'input FIFO. Comme ça, le nombre d'échantillons qui sépare le writing du reading est égale à 16000.

Puisqu'on échantillonne à 32kHz (0.03125ms par échantillon),
16000 échantillons * 0.03125ms/échantillon = 500ms.



Stratégie de l'effet de l'octave software

La stratégie pour faire un effet d'octave est d'avoir 2 buffers circulaire très petit (64 échantillons maximum). Buffer #1 a une phase de 0°, tandis que le buffer #2 a une phase de 180° entre **les indexes de reading**. Après que les buffers soient lus, leurs valeurs sont additionnées et moyennées pour être acheminé vers l'output FIFO.



Code du Logiciel

main.c

```c
/*
 * main.c
 *
 *  Created on: 2017-06-30
 *      Author: VINCENT GOSSELIN
 */


/* C Library */
#include <stdio.h>      // printf()
#include <string.h>     // sprintf()

#include "io.h"
#include "alt_types.h"
#include "system.h"

//for 7-segments
#include "sseg_driver.h"

//for buttons
#include "button_reader.h"
#define BUTTONS_BASE 0x101020

//for audio
#include "audio_driver.h"


//There are 4 different state : 0 -> no-effect, 1 -> distortion, 2 -> delay, 3 -> octave
//default is no effect
int state = 0;
//State selection
void state_select(void);

int main(void)
{
    //FALLING Edge selected in Qsys.
    clear_all_edge_capture;

    //Configure audio chip. By default, Line IN is connected to Line OUT by the BYPASS scheme.
    audio_init();

    while(1)
    {

       //Guitar Pedal.

        display_int_to_sseg(state); //0 for no-effect
        state_select();
        switch (state){
        case 0 :
            audio_no_effect();
            break;

        case 1 :
            audio_distortion();
            break;

        case 2 :
            audio_delay();
            break;

        case 3 :
            audio_octave();
            break;

        }
    }
    return 0;
}

void state_select(void){
    if(edge_capture & key0_pressed){
        state = 0;
        //clear_key0;
    } else if(edge_capture & key1_pressed){
        state = 1;
        //clear_key1;
    } else if(edge_capture & key2_pressed){
        state = 2;
        //clear_key2;
    } else if(edge_capture & key3_pressed){
        state = 3;
        //clear_key3;
    }

    clear_all_edge_capture;
}
```

Audio_driver.h

```
/*
 * audio_driver.h
 *
 *  Created on: 2017-06-29
 *      Author: VINCENT GOSSELIN
 */

#ifndef AUDIO_DRIVER_H_
#define AUDIO_DRIVER_H_

//audio config
#include "altera_up_avalon_audio_and_video_config.h"

//audio codec
#include "altera_up_avalon_audio.h"

//math for dsp
//#include "math.h"

//audio config device
alt_up_av_config_dev *av_config_dev;
//audio codec device
alt_up_audio_dev * audio_dev;

//Each Fifo of the audio codec can store up to 128 words of 32-bits wide.
//So, our Audio buffer will be able to store 64 words.
//#define AUDIO_BUFFER_LEN 16
//#define AUDIO_BUFFER_LEN 32
#define AUDIO_BUFFER_LEN 64 //CANNOT CHANGE ANYMORE.
//#define AUDIO_BUFFER_LEN 100
//#define AUDIO_BUFFER_LEN 120 <- does not work.

/* used for audio record/playback */
unsigned int l_buf[AUDIO_BUFFER_LEN];
unsigned int r_buf[AUDIO_BUFFER_LEN];

//For 500ms Echo.
//Sample rate is 32kHz, 31.25usec between each samples
//Echo will be at 500ms. So we need to store 16000samples.
#define DELAY_AUDIO_BUFFER_LEN 16000
unsigned int delay_audio_buffer[DELAY_AUDIO_BUFFER_LEN] = {0};
//int delay_audio_buffer_index = 0;
unsigned int storing_audio_index = 0;
unsigned int reading_audio_index = 1;
unsigned int outgoing_buffer[AUDIO_BUFFER_LEN];

//For Octave, Higher pitch use here.
#define OCTAVE_AUDIO_BUFFER_LEN AUDIO_BUFFER_LEN //best fit!
//#define OCTAVE_AUDIO_BUFFER_LEN 1024
unsigned int octave_buffer1 [ OCTAVE_AUDIO_BUFFER_LEN] = {0};
unsigned int octave_buffer2 [ OCTAVE_AUDIO_BUFFER_LEN] = {0};
unsigned int octave_reading_index1 = 0;
unsigned int octave_reading_index2 = ( OCTAVE_AUDIO_BUFFER_LEN/2)-1;//halfway through
unsigned int octave_storing_index = 0;
unsigned int octave_outputbuffer[AUDIO_BUFFER_LEN] = {0};




//Audio init, connects
void audio_init(void){

    //open the Audio codec device
    audio_dev = alt_up_audio_open_dev (AUDIO_CODEC_NAME);
    if ( audio_dev == NULL){
    printf("Error: could not open audio codec device \n");
    } else {
    printf("Opened audio codec device \n");
    }

    // open the Audio config device
    av_config_dev = alt_up_av_config_open_dev(AUDIO_CONFIG_NAME);
    if ( av_config_dev == NULL){
    printf("Error: could not open audio config device \n\r");
    } else {
    printf("Opened audio config device \n\r");
    }

    //Reset Audio Codec chip (WM8731) to empty Fifos + hardware init config.
    alt_up_audio_reset_audio_core(audio_dev);
    alt_up_av_config_reset(av_config_dev);

    //Audio Chip ready for I2C transfer?
    av_config_dev = alt_up_av_config_open_dev(AUDIO_CONFIG_NAME);
    if(alt_up_av_config_read_ready(av_config_dev)){
        printf("Audio chip ready for new I2c transfer \n\r");
    }

    //Writing a new config for the Audio chip.
    //Enable BYPASS, should received audio now. Audio Path control Register = 0x04, Data to send = 0x0A
    //alt_up_av_config_write_audio_cfg_register(av_config_dev, 0x04, 0x0A);

    //Enabling Left Line Input
    //0db, Disable mute.
    alt_up_av_config_write_audio_cfg_register(av_config_dev, 0x00, 0x17);

    //NO right channel since my cheap audio cable has only left channel working...
```

```c
    //Enabling Right Line Input
    //0db, Disable mute.
    alt_up_av_config_write_audio_cfg_register(av_config_dev, 0x01, 0x17);
    //alt_up_av_config_write_audio_cfg_register(av_config_dev, 0x01, 0x80);

    //Left Headphone Out.
    //0db, with Zero Crossing Enable
    alt_up_av_config_write_audio_cfg_register(av_config_dev, 0x2, 0xF9);
    //Right Headphone Out.
    //0db, with Zero Crossing Enable
    alt_up_av_config_write_audio_cfg_register(av_config_dev, 0x3, 0xF9);

    //Analogue Audio Path Control
    //Select DAC, Disable Bypass to be output to RHPout/LHPout.
    alt_up_av_config_write_audio_cfg_register(av_config_dev, 0x04, 0x12);

    //Digital Audio Path Control
    //Disable DAC Soft Mute Control
    alt_up_av_config_write_audio_cfg_register(av_config_dev, 0x5, 0x06);

    //Power Down Control
    //everything ON except MICPD, OSCPD, CLKOUTPD. Not using MIC input.
    alt_up_av_config_write_audio_cfg_register(av_config_dev, 0x6, 0x02);

    //The rest of control registers are initialized by Hardware.
}

//No effect, this simply reads value from ADC and outputs them to DAC.
// NOTE : with my cheap audio cable, only the LEFT input works.
void audio_no_effect(void){

    //Select DAC, Disable Bypass to be output to RHPout/LHPout.
    alt_up_av_config_write_audio_cfg_register(av_config_dev, 0x04, 0x12);

    //EVEN BETTER, best one.

    int fifospace = alt_up_audio_read_fifo_avail (audio_dev, ALT_UP_AUDIO_LEFT);
    if(fifospace>AUDIO_BUFFER_LEN){

        alt_up_audio_read_fifo (audio_dev, r_buf, AUDIO_BUFFER_LEN, ALT_UP_AUDIO_RIGHT);
        alt_up_audio_read_fifo (audio_dev, l_buf, AUDIO_BUFFER_LEN, ALT_UP_AUDIO_LEFT);

        //AROUND 32767,
        int i;
        for(i=0;i<AUDIO_BUFFER_LEN;i++){
            if(l_buf[i]>40000){
                l_buf[i] = l_buf[i] - 0x7fff;
            } else {
                 l_buf[i] = l_buf[i] + 0x7fff;
             }
         }

         alt_up_audio_write_fifo (audio_dev, l_buf, AUDIO_BUFFER_LEN, ALT_UP_AUDIO_RIGHT);
         alt_up_audio_write_fifo (audio_dev, l_buf, AUDIO_BUFFER_LEN, ALT_UP_AUDIO_LEFT);
    }

}

void audio_distortion(void){
    //Not the expected distortion algorithm but it works.
    //This one works with frequency cancellations and bouncing I think.

    //Old working code but it works. This is the distortion.
    //WORKING CODE

    unsigned int distortion_l_buf;
    unsigned int distortion_r_buf;

    // read audio buffer
    alt_up_audio_read_fifo (audio_dev, &(distortion_r_buf), 1, ALT_UP_AUDIO_RIGHT);
    alt_up_audio_read_fifo (audio_dev, &(distortion_l_buf), 1, ALT_UP_AUDIO_LEFT);

    if(distortion_l_buf>40000){
        distortion_l_buf = distortion_l_buf - 0x7fff ;

            } else {
                distortion_l_buf = distortion_l_buf + 0x7fff ;

            }

    //Write audio buffer
    alt_up_audio_write_fifo (audio_dev, &(distortion_l_buf), 1, ALT_UP_AUDIO_RIGHT);
    alt_up_audio_write_fifo (audio_dev, &(distortion_l_buf), 1, ALT_UP_AUDIO_LEFT);
    alt_up_audio_write_fifo (audio_dev, &(distortion_l_buf), 1, ALT_UP_AUDIO_RIGHT);
    alt_up_audio_write_fifo (audio_dev, &(distortion_l_buf), 1, ALT_UP_AUDIO_LEFT);
    alt_up_audio_write_fifo (audio_dev, &(distortion_l_buf), 1, ALT_UP_AUDIO_RIGHT);
    alt_up_audio_write_fifo (audio_dev, &(distortion_l_buf), 1, ALT_UP_AUDIO_LEFT);
    alt_up_audio_write_fifo (audio_dev, &(distortion_l_buf), 1, ALT_UP_AUDIO_RIGHT);
    alt_up_audio_write_fifo (audio_dev, &(distortion_l_buf), 1, ALT_UP_AUDIO_LEFT);
}

void audio_delay(void){
    //Sample rate is 32kHz, 31.25usec between each samples

    //echo will be at 500ms. So we need to store 16000samples.

    //Active bypass + dac in audio path
    alt_up_av_config_write_audio_cfg_register(av_config_dev, 0x04, 0x1A);

    //Aquiring/Writing to Audio Chip
    int input_fifo = alt_up_audio_read_fifo_avail(audio_dev, ALT_UP_AUDIO_LEFT);
    //For synchronizing writing/reading from audio chip.
    if(input_fifo>AUDIO_BUFFER_LEN){

        //Circular buffer of 16000
        //storing index
        storing_audio_index = storing_audio_index % (DELAY_AUDIO_BUFFER_LEN);
        //reading index
        reading_audio_index = reading_audio_index % (DELAY_AUDIO_BUFFER_LEN-64);//SOLVE THE TOCTOC

        //Reading input fifo.
        alt_up_audio_read_fifo (audio_dev, r_buf, AUDIO_BUFFER_LEN, ALT_UP_AUDIO_RIGHT);
        alt_up_audio_read_fifo (audio_dev, l_buf, AUDIO_BUFFER_LEN, ALT_UP_AUDIO_LEFT);
        //printf("words_read = %d\n\r",words_read);
```

```
212
213         //for 100 samples.
214         int i;
215         for(i=0;i<AUDIO_BUFFER_LEN;i++){
216
217             //ADJUSTING TO AROUND 32767,
218             if(l_buf[i]>40000){
219                 l_buf[i] = l_buf[i] - 0x7fff;
220             } else {
221                 l_buf[i] = l_buf[i] + 0x7fff;
222             }
223
224             //storing in delay_buffer
225             delay_audio_buffer[storing_audio_index] = l_buf[i];
226             //retriving from buffer
227             outgoing_buffer[i] = delay_audio_buffer[reading_audio_index];
228
229             //Reading index is always 1 sample ahead of storing index.
230             storing_audio_index = storing_audio_index + 1;
231             reading_audio_index = reading_audio_index + 1;
232
233         }
234         //writing to output fifo
235         alt_up_audio_write_fifo (audio_dev, outgoing_buffer, AUDIO_BUFFER_LEN, ALT_UP_AUDIO_RIGHT);
236         alt_up_audio_write_fifo (audio_dev, outgoing_buffer, AUDIO_BUFFER_LEN, ALT_UP_AUDIO_LEFT);
237     }
238
239 }
240
241 void audio_octave(void){
242
243     //Not the best but prove of concept is done.
244     //Stragegy : buffer #1 is phase 0, buffer #2 is phase 180. Add both values and average.
245
246     //Not required.
247     //Active bypass + dac in audio path
248     //alt_up_av_config_write_audio_cfg_register(av_config_dev, 0x04, 0x1A);
249
250     //Dual buffer scheme use.
251     int fifospace = alt_up_audio_read_fifo_avail (audio_dev, ALT_UP_AUDIO_LEFT);
252         if(fifospace>AUDIO_BUFFER_LEN){
253
254             //2 circular buffer phased shift by 180degrees.
255             octave_storing_index = octave_storing_index %  OCTAVE_AUDIO_BUFFER_LEN;
256             octave_reading_index1 = octave_reading_index1 %  OCTAVE_AUDIO_BUFFER_LEN;
257             octave_reading_index2 = octave_reading_index2 %  OCTAVE_AUDIO_BUFFER_LEN;
258
259             alt_up_audio_read_fifo (audio_dev, r_buf, AUDIO_BUFFER_LEN, ALT_UP_AUDIO_RIGHT);
260             alt_up_audio_read_fifo (audio_dev, l_buf, AUDIO_BUFFER_LEN, ALT_UP_AUDIO_LEFT);
261
262
263             int i;
264             for(i=0;i<AUDIO_BUFFER_LEN;i++){
265                 //AROUND 32767
266                 if(l_buf[i]>40000){
267                     l_buf[i] = l_buf[i] - 0x7fff;
268                 } else {
269                     l_buf[i] = l_buf[i] + 0x7fff;
270                 }
271
272                 //storing in octave_buffer1
273                 octave_buffer1[octave_storing_index] = l_buf[i];
274                 //storing in octave_buffer2
275                 octave_buffer2[octave_storing_index] = l_buf[i];
276                 //reading from octave_buffer1 into a temp1 value. Will be further averaged.
277                 unsigned int temp1,temp2;
278                 temp1 = octave_buffer1[octave_reading_index1];
279                 temp2 = octave_buffer2[octave_reading_index2];
280                 //Average is taken
281                 unsigned int average;
282                 average = (temp1+temp2)/2;
283                 //Average is put into octave_outputbuffer
284                 octave_outputbuffer[i] =  average;
285                 //printf("average is %d\n\r", average);
286
287                 //Best fit.
288                 octave_reading_index1 = octave_reading_index1 + 1;
289                 octave_reading_index2 = octave_reading_index2 + 1;
290                 octave_storing_index = octave_storing_index + 1;
291
292             }
293             //writing to output fifo
294             alt_up_audio_write_fifo (audio_dev, octave_outputbuffer, AUDIO_BUFFER_LEN, ALT_UP_AUDIO_RIGHT);
295             alt_up_audio_write_fifo (audio_dev, octave_outputbuffer, AUDIO_BUFFER_LEN, ALT_UP_AUDIO_LEFT);
296         }
297
298 }
299
300 #endif /* AUDIO_DRIVER_H_ */
```

Button_reader.h

```
/*
 * button_reader.h
 *
 *  Created on: 2017-05-25
 *      Author: Vincent Gosselin, Carl Trudeau
 */

#ifndef BUTTON_READER_H_
#define BUTTON_READER_H_

#define buttons_register IORD(BUTTONS_BASE,0) //sur 4 bits.
#define key0 0xe
#define key1 0xd


#define edge_capture IORD(BUTTONS_BASE,3) //sur 4 bits.
#define key0_pressed 0x1
#define key1_pressed 0x2
#define key2_pressed 0x4
#define key3_pressed 0x8

#define clear_all_edge_capture IOWR(BUTTONS_BASE,3, 0xf)
#define clear_key0 IOWR(BUTTONS_BASE,3, 0x1)
#define clear_key1 IOWR(BUTTONS_BASE,3, 0x2)
#define clear_key2 IOWR(BUTTONS_BASE,3, 0x4)
#define clear_key3 IOWR(BUTTONS_BASE,3, 0x8)

//if(buttons_register==key1)//key0 pressed becomes LOW.

#endif /* BUTTON READER H  */
```

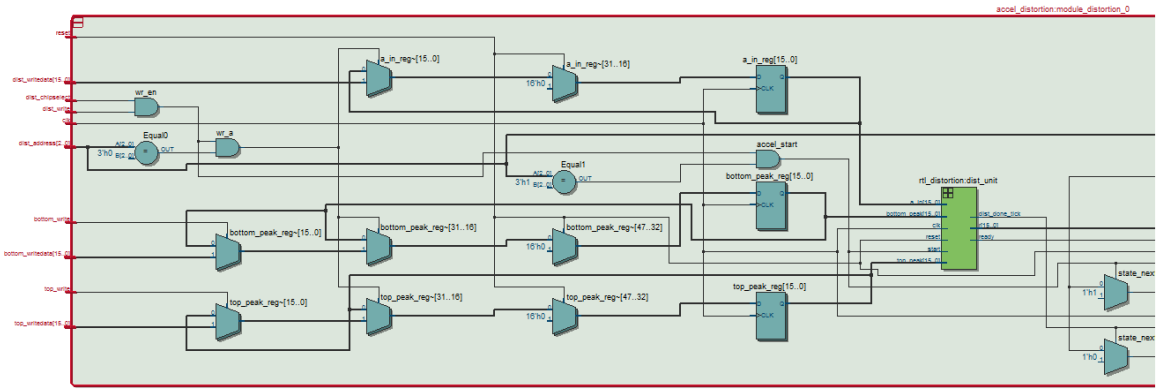
sseg_driver.h

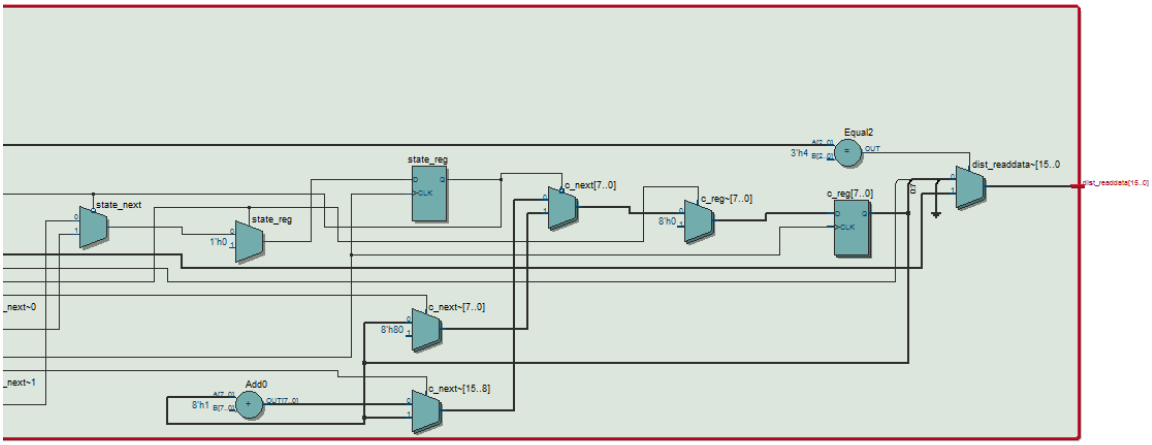
```c
/*
 * sseg_driver.h
 *
 *  Created on: 2017-05-25
 *      Author: Vincent Gosselin, Carl Trudeau
 */

#ifndef SSEG_DRIVER_H_
#define SSEG_DRIVER_H_


#define PIO_DATA_REG_OFT 0  //offset du registre Data
#define pio_write(base,data) IOWR(base, PIO_DATA_REG_OFT, data)

/*

 exemple d'utilisation affichant "12EF" sur les 7 seg

    alt_u8 message[4];

    message[0] = sseg_conv_hex(0x01);
    message[1] = sseg_conv_hex(0x0b);
    message[2] = sseg_conv_hex(0x00);
    message[3] = sseg_conv_hex(0x0a);
    sseg_disp_4_digit(SSEG_1_4_BASE, message); //equivalent message[0]
*/


alt_u8 sseg_conv_hex(int hex)
{
    /* patron hexadecimaux pour afficheur 7seg active-low (0-9, a-f)
     * le msb est ignore */
    static const alt_u8 SSEG_HEX_TABLE[16] = {
            0x40, 0x79, 0x24, 0x30, 0x19, 0x92, 0x02, 0x78, 0x00, 0x10, // 0-9
            0x88, 0x03, 0x46, 0x21, 0x06, 0x0E};                        // a-f

    alt_u8 pattern;

    if (hex < 16) {
        pattern = SSEG_HEX_TABLE[hex];
    } else {
        pattern = 0xff; //tous eteint
    }
    return (pattern);
}


void sseg_disp_4_digit(alt_u32 base, alt_u8 *digit)
{
    /* digit est l'adresse d'un tableau de 4 alt_u8 */

    alt_u32 sseg_data = 0;
    int i;

    /* assemblage de 4 donnees par OR et decalage pour former un 32 bit */
    for(i = 0; i < 4; i++){
        sseg_data = (sseg_data << 8) | *digit;
        digit++;
    }
    pio_write(base,sseg_data);
}

void display_int_to_sseg(int number)
{
    alt_u8 message[4];


    //Pour Display sur le hex3.
    int i = 0;
    while(number >= 1000)
    {
        i++;
        number -= 1000;

    }
    //display i  on hex3.
    message[0] = sseg_conv_hex(i);
    i = 0;
    while(number >= 100)
    {
        i++;
        number -= 100;
    }
    //display i  on hex2.
    message[1] = sseg_conv_hex(i);
    i = 0;
    while(number >= 10)
    {
        i++;
        number -= 10;
    }
    //display i  on hex1.
    message[2] = sseg_conv_hex(i);
    i = 0;
```

```
95      while(number >= 1)
96      {
97          i++;
98          number -= 1;
99      }
100     //display i  on hex0.
101     message[3] = sseg_conv_hex(i);
102     sseg_disp_4_digit(SSEG_I_IV_BASE, message); //equivalent message[0]
103
104     alt_u8 message1[4];
105
106     message1[0] = sseg_conv_hex(0x01);//no hex here...
107     message1[1] = sseg_conv_hex(0x02);//no hex here...
108     message1[2] = 0x40;//hex5 -> OFF
109     message1[3] = 0x40;//hex4 -> OFF
110     sseg_disp_4_digit(SSEG_V_VI_BASE, message1); //equivalent message[0]
111
112 }
113
114 void display_pause(void)
115 {
116     alt_u8 message[4];
117
118     message[0] = 0x41;//hex3 -> U
119     message[1] = 0x12;//hex2 -> S
120     message[2] = 0x06;//hex1 -> E
121     message[3] = 0xff;//hex0 -> OFF.
122     sseg_disp_4_digit(SSEG_I_IV_BASE, message); //equivalent message[0]
123
124     alt_u8 message1[4];
125
126     message1[0] = sseg_conv_hex(0x01);
127     message1[1] = sseg_conv_hex(0x02);
128     message1[2] = 0x0c;//hex5 -> P
129     message1[3] = 0x88;//hex4 -> A
130     sseg_disp_4_digit(SSEG_V_VI_BASE, message1); //equivalent message[0]
131 }
132
133 //usefull for debugging
134 void display_all_off(void)
135 {
136     alt_u8 message[4];
137
138         message[0] = 0xff;//OFF
139         message[1] = 0xff;
140         message[2] = 0xff;
141         message[3] = 0xff;
142         sseg_disp_4_digit(SSEG_I_IV_BASE, message); //equivalent message[0]
143
144         alt_u8 message1[4];
145
146         message1[0] = 0xff;
147         message1[1] = 0xff;
148         message1[2] = 0xff;
149         message1[3] = 0xff;
150         sseg_disp_4_digit(SSEG_V_VI_BASE, message1); //equivalent message[0]
151 }
152
153 #endif /* SSEG_DRIVER_H_ */
```

ARCHITECTURE DE L'ACCÉLÉRATEUR

Voici le schema RTL de l'accelerateur matériel lorsque compilé avec Quartus :





Fonctionnement de l'accélérateur

Le signal audio sort de l'ADC échantillonné sur 16 bits signés. Il est décalé par +/- 0x7FFF pour être positionner dans le milieu de l'échelle 16 bits. Après le décalage, la valeur signé est prête à être écrite dans l'accélérateur de distorsion. Le premier comparateur, à la sortie du registre, compare avec le niveau de référence de 1.65V (32768) afin de valider si nous sommes dans l'alternance positive ou négative du signal audio. Un bit de contrôle servira à actionner une des sorties du Mux. Ensuite on effectue une soustraction avec le niveau binaire "peak" de chacune des alternances, si le résultat est supérieur à 60000 ou inférieur à 5535, on achemine cette valeur plutôt que le niveau de l'échantillon à l'entrée. Ceci a pour but "d'éliminer" la portion crête positive et négative du signal audio reçu et ainsi créer un effet sonore de distorsion.

CODE VHDL DU DESIGN RTL DU MODULE DE DISTORTION

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity rtl_distortion is
    port(
        clk : in std_logic;
        reset : in std_logic;
        start : in std_logic;
        a_in : in std_logic_vector(15 downto 0);
        top_peak: in std_logic_vector(15 downto 0);
        bottom_peak : in std_logic_vector(15 downto 0);
        dist_done_tick : out std_logic;
        ready : out std_logic;
        r: out std_logic_vector(15 downto 0)
        );

end rtl_distortion;

architecture behaviour of rtl_distortion is
    type state_type is (idle, op);
    signal state_reg : state_type;
    signal state_next : state_type;
    signal a_reg : unsigned(15 downto 0);
    signal a_next : unsigned(15 downto 0);
    signal peak_plus : unsigned(15 downto 0);
    signal peak_moins: unsigned(15 downto 0);
    signal peak_plus_next: unsigned(15 downto 0);
    signal peak_moins_next: unsigned(15 downto 0);
    signal r_out : unsigned(15 downto 0);
    signal top_compare : unsigned(15 downto 0);
    signal bottom_compare : unsigned(15 downto 0);
begin
    process(clk)
    begin
        if(clk 'event and clk = '1') then  -- on défini l'état idle en initialisant tout les registre à 0
                                           -- avec un reset synchrone
            if(reset = '1') then
                state_reg <= idle;
                a_reg <= (others=>'0');
                peak_plus <= (others=>'0');
                peak_moins <= (others=>'0');
            else
                state_reg <= state_next;    --si le reset n'est pas actionné, la valeur suivant dans la machine a état est prise
                                            --et assigné a leur registre respectifs
                a_reg <= a_next;
                peak_plus <= peak_plus_next;
                peak_moins <=peak_moins_next;
            end if;
        end if;
    end process;

    process(state_reg,a_reg,peak_plus,peak_moins,start,a_in,top_peak,bottom_peak)
    begin
        -- définition des différents états de l'accélérateur matériel du module du distortion
        a_next <= a_reg;
        state_next <= state_reg;
        dist_done_tick <= '0';
        case state_reg is
            when idle =>
                if start = '1' then
                    a_next <= unsigned(a_in);
                    peak_plus_next <= unsigned(top_peak);
                    peak_moins_next <= unsigned(bottom_peak);
                    state_next <= op;
                end if;

            when op =>   --ici on valide l'état en fonction de la sortie afin de savoir si le process s'est déja exécuter
                if(r_out > x"0000") then
                    state_next <= idle;
                    dist_done_tick <= '1';
                    r_out <= x"0000";
                 else
                    if(a_reg > x"8000") then --on détermine ici s'il s'agit d'une alternance positive ou négative du signal entrant
                        top_compare <= a_reg;
                    else
                        bottom_compare <= a_reg;
                    end if;
                end if;
                -- afin de "couper" le peak positif ou négatif du signal, une comparaison est effectuée afin de valider
                -- si le signal est supérieur ou inférieur à la limite que l'utilisateur défini
                if(top_compare > x"0000" and top_compare > peak_plus_next) then
                    r_out <= peak_plus_next;
                else
                    if(top_compare > x"0000" and top_compare < peak_plus_next) then
                        r_out <= peak_plus_next;
                    end if;
                end if;

                if(bottom_compare > x"0000" and bottom_compare < peak_moins_next) then
                        r_out <= peak_moins_next;
                else
                    if(bottom_compare > x"0000" and bottom_compare > peak_moins_next) then
                        r_out <= bottom_compare;
                    end if;
                end if;
            end case;
        end process;

        ready <= '1' when state_reg=idle else '0';
        r <= std_logic_vector(r_out);
    end behaviour;

```

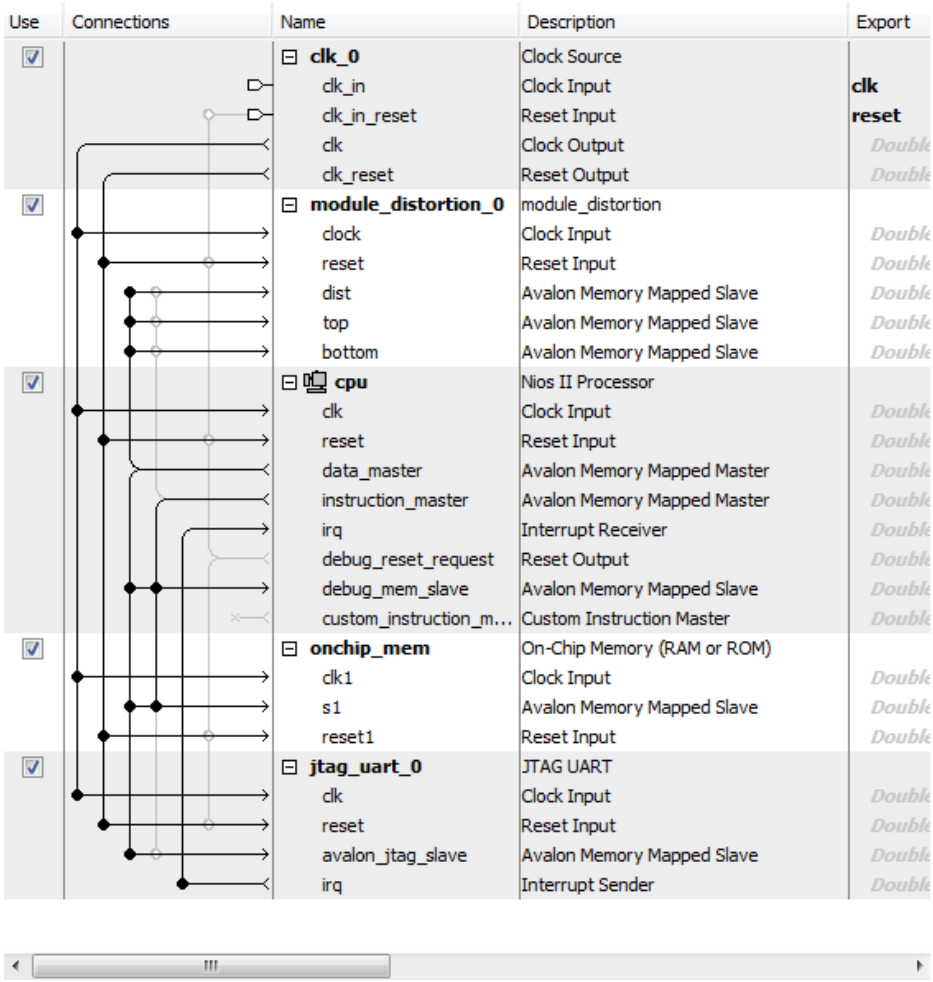
CODE VHDL DE L'ACCÉLÉRATEUR

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity accel_distortion is
    port (
        clk : in std_logic;
        reset : in std_logic;
        dist_address : in std_logic_vector(2 downto 0);
        dist_chipselect : in std_logic;
        dist_write : in std_logic;
        top_write : in std_logic;
        bottom_write : in std_logic;
        dist_writedata : in std_logic_vector(15 downto 0);
        top_writedata : in std_logic_vector(15 downto 0);
        bottom_writedata: in std_logic_vector(15 downto 0);
        dist_readdata : out std_logic_vector(15 downto 0)
         );

end accel_distortion;

architecture behaviour of accel_distortion is
    signal accel_start : std_logic;
    signal accel_done_tick : std_logic;
    signal a_in_reg : std_logic_vector(15 downto 0);
    signal top_peak_reg : std_logic_vector(15 downto 0);
    signal bottom_peak_reg : std_logic_vector(15 downto 0);
    signal r_out : std_logic_vector(15 downto 0);
    signal accel_ready : std_logic;
    signal wr_en : std_logic;
    signal wr_a : std_logic;
    type state_type is (idle,count);
    signal state_reg : state_type;
    signal state_next : state_type;
    signal c_reg : unsigned(7 downto 0);
    signal c_next : unsigned(7 downto 0);

component rtl_distortion
    port(
        clk : in std_logic;
        reset : in std_logic;
        start : in std_logic;
        a_in : in std_logic_vector(15 downto 0);
        top_peak: in std_logic_vector(15 downto 0);
        bottom_peak : in std_logic_vector(15 downto 0);
        dist_done_tick : out std_logic;
        ready : out std_logic;
        r: out std_logic_vector(15 downto 0)
        );

end component;
-- instantiation du composant module distortion
begin
    dist_unit : rtl_distortion port map (clk => clk, reset => reset, start => accel_start,
                                a_in => a_in_reg, top_peak => top_peak_reg,
                                bottom_peak => bottom_peak_reg,dist_done_tick => accel_done_tick,
                                ready => accel_ready, r => r_out);

process (clk)
    begin -- définition d'un reset synchrone
        if(clk 'event and clk = '1') then
            if(reset = '1') then
               a_in_reg <= (others => '0');
               top_peak_reg <= (others => '0');
               bottom_peak_reg <= (others => '0');
            else
            -- définition des bits d'écrture pour chacun des registres du module
               if wr_a = '1' then
                  a_in_reg <= dist_writedata;
                  if top_write = '1' then
                  top_peak_reg <= top_writedata;
                  end if;
                  if bottom_write = '1' then
                  bottom_peak_reg <= bottom_writedata;
                  end if;
               end if;
            end if;
        end if;
    end process;
    -- mapping nécessaire pour générer un module qsys
    wr_en <=
       '1' when dist_write ='1' and dist_chipselect = '1' else '0';

    wr_a <= '1' when dist_address = "000" and wr_en = '1' else '0';
    accel_start <= '1' when dist_address = "001" and wr_en = '1' else '0';

    dist_readdata <= r_out when dist_address = "100" else
                 accel_ready & "0000000" &
                 std_logic_vector(c_reg);


process (clk)
begin -- assignation d'un reset synchrone pour les différents états de l'accélérateur
    if(clk 'event and clk = '1') then
        if(reset = '1') then
            state_reg <= idle;
            c_reg <= (others=> '0');

        else
            state_reg <= state_next;
            c_reg <= c_next;
        end if;
    end if;
end process;

process(state_reg,c_reg,accel_start, accel_done_tick)
```

```
105
106 process(state_reg,c_reg,accel_start, accel_done_tick)
107 begin
108     -- on définit ici le compteur pour évaluer le nombre que l'accélérateur a effectué
109     c_next <= c_reg;
110     state_next <= state_reg;
111     case state_reg is
112         when idle =>
113             if (accel_start= '1') then
114                 c_next <= x"01" ;
115                 state_next <= count;
116             end if;
117         when count =>
118             if(accel_done_tick = '1') then
119                 state_next <= idle;
120             else
121                 c_next <= c_reg + 1;
122             end if;
123     end case;
124 end process;
125
126
127
128 end behaviour;
```

ANALYSE DE PERFORMANCE ET DISCUSSION

Le composant Qsys a été généré avec succès, cependant des bugs sont survenus lorsque fut venu le temps de valider le fonctionnement en générant un code pour le test ave Eclipse. Donc, pour ce qui est de l'analyse de performance, le temps d'exécution n'a pas pu être analysé. Voici tout de même un schéma détaillé du composant qsys final résultant de l'accélérateur matériel ci-haut :



Discussion

La barrière d'entré de ce projet était de faire fonctionner le 'aucun-effet'. Le problème principal était qu'il y avait un bruit statique mélangé à la musique injecté dans la carte DE1-SoC. Il s'est avéré que le bruit statique était présent à cause que le NIOS II lisait complètement les deux input FIFO du Audio IP core. Cela ne laissant pas le temps à la puce audio de générer d'autres échantillons, donc l'apparition de bruit statique. La solution était de limiter la lecture en fonction d'échantillons disponibles dans les inputs FIFO. En faisant cela, le 'aucun-effet' était fonctionnel. Après, est venu l'effet de distorsion. Le plan initial était de clipper le signal entrant avec une limite inférieure et supérieure pour générer de la distorsion. L'équipe a décidé de faire un accélérateur matériel pour effectuer ce traitement de signal. Pourtant, après des tests au niveau software, l'équipe s'est rendu compte que cette façon d'obtenir de la distorsion était dangereux pour les oreilles à cause de la présence de craquements et de 'pops' en sortie de la carte DE1-SoC. L'alternative a été découverte en conséquence des tests de 'aucun-effet'. Il a été découvert qu'en répétant un échantillon 4 fois dans les outputs buffers que cela causait de la distorsion audio confortable pour les oreilles. Ensuite, l'équipe à travailler sur l'effet de délai. Le problème principal était un 'buzzing' constant lors de la manipulation des données dans le buffer de 16000 échantillons. Il s'est avéré que l'horloge du système n'était pas assez vite. En passant de 50MHz à 100MHz, l'équipe a été capable de faire fonctionner l'effet de délai. Pour l'effet de l'octave, nous avons eu le choix de procédé avec un module FFT ou de faire de l'octave avec la superposition d'échantillons déphasés. La deuxième méthode était plus simple à implémenter donc l'équipe a procédé ainsi.

Pour ce qui est du design de la partie analogique, l'équipe avait déjà en sa possession l'analogue front-end par d'anciens projets audio. Il a été plus simple d'utiliser le module que d'en créer un et de faire le PCB nous-même.

Nous avons découvert qu'il y a une lacune importante dans le Audio IP core de Altera. La *Master Clock* alimentant la chip audio (WM8731) n'est pas connecté par défaut. L'équipe a dû inclure une horloge séparé pour alimenter la pin du FPGA connecté physiquement au *Master Clock* du WM8731. C'est pour cette raison que nous avons utilisé un *Clock Bridge* pour alimenter le audio core et la pin du *Master Clock.*

Conclusion

Il aurait été intéressant de réaliser nous-même une composante Qsys pour interfacer avec la puce audio pour combler le fait que le Audio IP core manque le signal de *Master Clock*. La décision de réaliser un accélérateur matériel pour l'effet de distorsion n'était pas la meilleure idée à cause craquements et de 'pops' dans le signal audio lorsque testé en software. D'autre part, l'effet d'octave aurait plus convenu à une implémentation en accélérateur matériel à cause que l'algorithme avec une FFT (fast fourier transform) est demandant en puissance de calcul. Bref, le projet de la pédale de guitare a été une réussite malgré que tous les effets ont été réaliser sans accélérateurs matériels.