Università degli Studi di Salerno

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA E MATEMATICA APPLICATA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

PROGETTAZIONE ED IMPLEMENTAZIONE DI UN EQUALIZZATORE AUDIO SU SYSTEM ON CHIP

Gruppo

Andrea Scala - 0622702346 - a.scala31@studenti.unisa.it

Giuseppe Squitieri - 0622702339 - g.squitieri8@studenti.unisa.it

Ermanno Troisi - 0622702288 - e.troisi10@studenti.unisa.it

Anno Accademico 2024/2025

Contents

Chapter 1

Obiettivo del progetto

L'obiettivo di questo progetto è stato la realizzazione di un **equalizzatore audio digitale** a tre bande, implementato su una **scheda Zynq-7000 (Zynq7)**. L'equalizzatore consente di modificare la risposta in frequenza di un segnale audio in tempo reale, attivando o disattivando singolarmente tre filtri che operano su diverse bande di frequenza.
L'intero processo di sviluppo ha previsto diverse fasi, a partire dalla progettazione dei **filtri digitali** su **MATLAB** e successiva implementazione in **Vitis HLS**, fino all'implementazione su hardware programmabile tramite **Vivado** e alla realizzazione del software di controllo in linguaggio C attraverso l'ambiente **Vitis**.
Nello specifico, sono stati progettati tre filtri:

- un **filtro passa-basso** con frequenza di taglio a 300 Hz;
- un **filtro passa-banda** compreso tra 300 Hz e 3 kHz;
- un **filtro passa-banda** da 3 kHz a 20 kHz.

I coefficienti di questi filtri, ottenuti da MATLAB, sono stati esportati e successivamente integrati nel progetto hardware. L'equalizzatore permette, tramite tre switch, di attivare o disattivare ciascun filtro, influenzando così l'elaborazione del segnale audio in uscita.
Il progetto ha permesso di mettere in pratica competenze relative all'*elaborazione del segnale digitale*, alla *progettazione su FPGA* e all'*integrazione tra hardware e software*. L'utilizzo della piattaforma Zynq, che combina un processore ARM con un FPGA, ha fornito un ambiente flessibile e potente per la realizzazione di questo sistema embedded.

Chapter 2

Progettazione filtri

La prima fase del progetto ha riguardato la progettazione dei tre filtri digitali utilizzando il tool **Filter Designer** di **MATLAB**. Questo strumento permette di progettare, analizzare e salvare filtri digitali in modo interattivo, facilitando l'impostazione dei parametri e la visualizzazione della risposta in frequenza.
Sono stati realizzati i seguenti filtri:

- **Filtro passa-basso** con frequenza di taglio a 300 Hz;
- **Filtro passa-banda** con banda compresa tra 300 Hz e 3 kHz;
- **Filtro passa-banda** con banda compresa tra 3 kHz e 20 kHz;

Per ciascun filtro, una volta definite le specifiche (tipo di filtro, frequenze di taglio, ordine, risposta desiderata, ecc.), sono stati generati i coefficienti numerici che descrivono il comportamento del filtro nella forma discreta. Tali coefficienti sono stati esportati come file di testo, per poter essere successivamente utilizzati nelle fasi di implementazione hardware e software.
I coefficienti sono stati successivamente **formattati** in modo opportuno per l'integrazione nel codice C, e adattati alle specifiche dell'architettura di elaborazione, in particolare considerando il numero di bit e il tipo di rappresentazione, in particolare i coefficienti sono stati generati da MATLAB in formato floating point a singola precisione, e sono stati successivamente scalati a numeri a virgola fissa in formato Q1.15 con un semplice shift sinistro di 15, permettendoci così di implementare operazioni in virgola fissa invece che in virgola mobile, le quali risultano molto più lente.
Questa fase ha rappresentato un passaggio cruciale, poiché la qualità dell'equalizzazione finale dipende strettamente dalla corretta progettazione e implementazione dei filtri digitali.

2.1 Filtro passa-basso

Tutti i filtri hanno una frequenza di campionamento di 48KHz. Il filtro passa-basso ha queste caratteristiche:

- frequenza di taglio: 300Hz;
- frequenza di stop: 1kHz;
- attenuazione stopband: 60dB;
- densità: 20;
- tipo: equiripple.

Il filtro risultante ha 137 coefficienti; si è scelta una banda di stop di 1kHz poiché con una banda più stretta sarebbero stati generati troppi coefficienti.

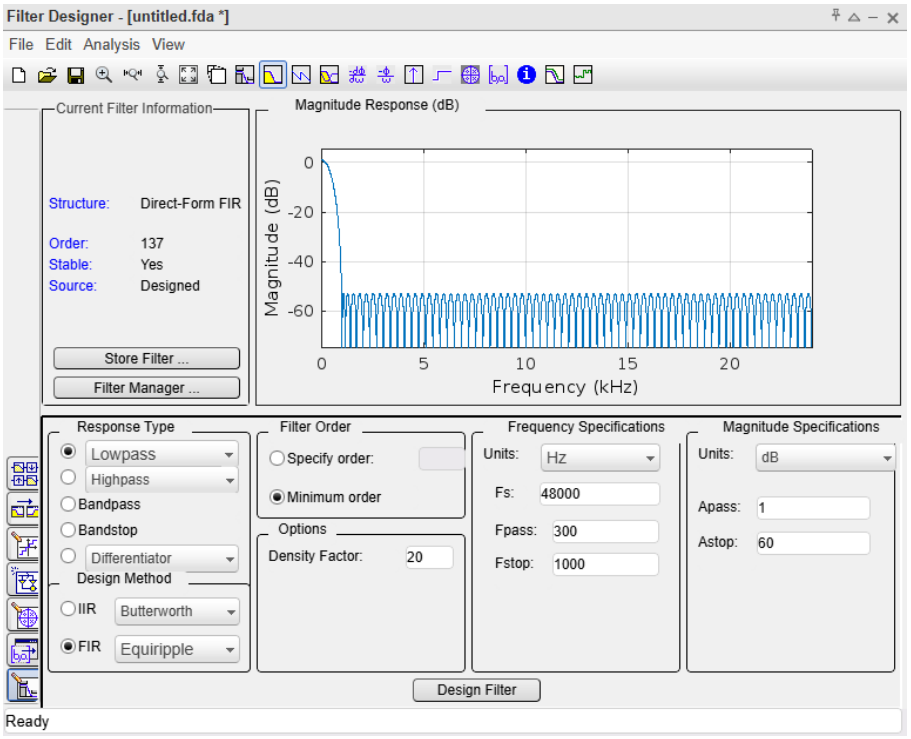

Figure 2.1: caratteristiche del filtro passa-basso

2.2 Filtro passa-banda a banda media

Il filtro passa-banda medio ha queste caratteristiche:

- frequenze di taglio: 300Hz-3kHz;
- tipo: Window;
- finestra: Hamming;
- ordine: 150;

si è deciso per un ordine specifico per evitare che il tool generi troppi coefficienti. Per il tipo di finestra sono state provate diverse finestre fin quando la risposta in frequenza non sembrava abbastanza appropriata.

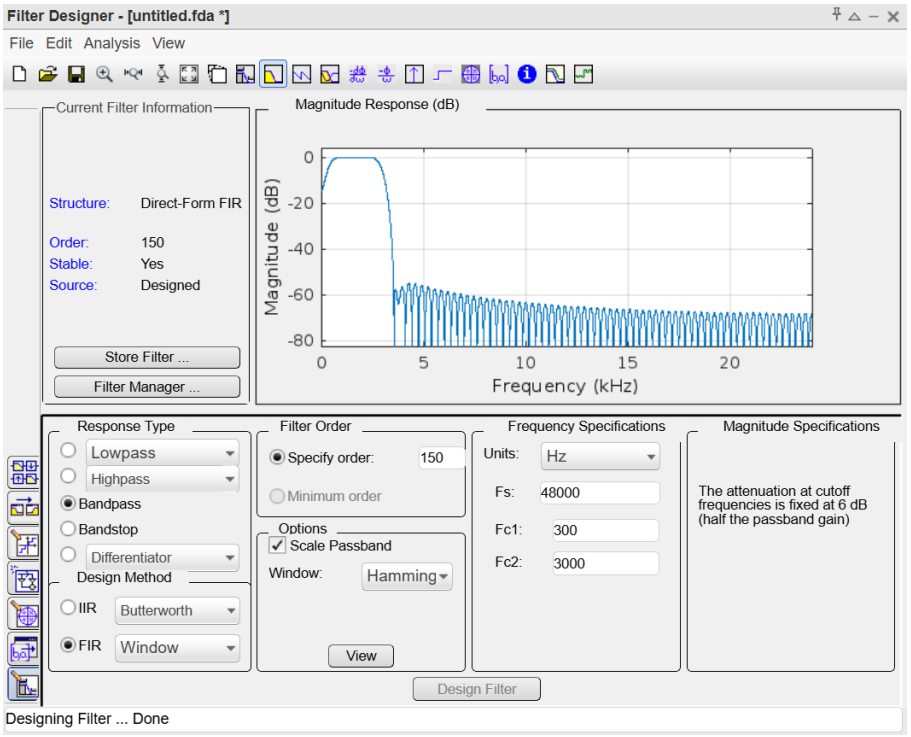

Figure 2.2: caratteristiche del filtro passa-banda medio

2.3 Filtro passa-banda a banda alta

Il filtro passa-banda alto ha queste caratteristiche:

- frequenze di taglio: 3kHz-20kHz;
- tipo: Window;
- finestra: Hamming;
- ordine: 150;

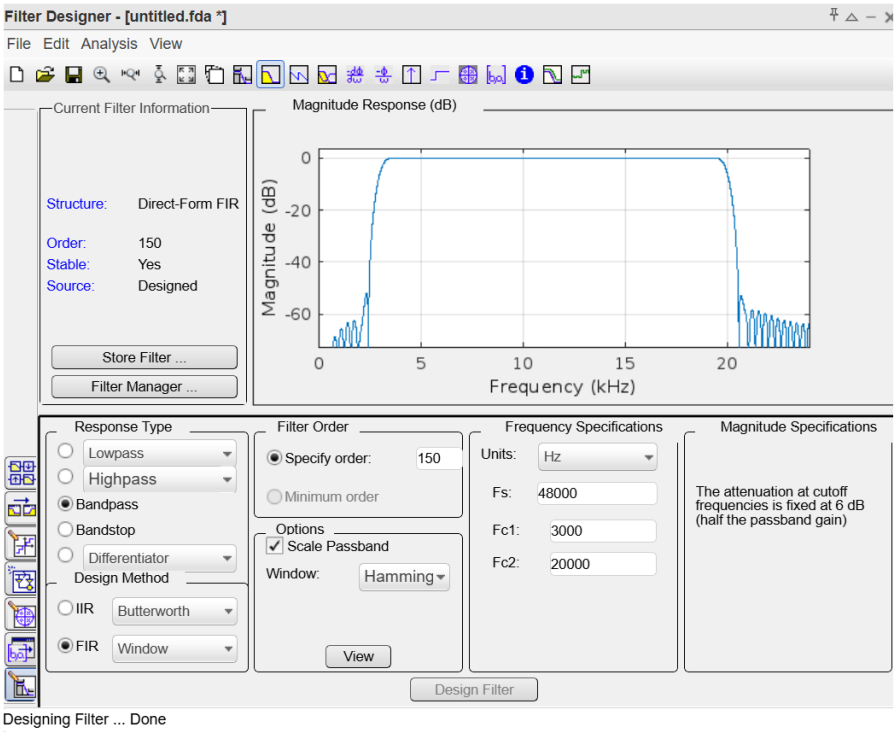

Figure 2.3: caratteristiche del filtro passa-banda alto

Per ogni filtro ne sono stati esportati i coefficienti e si è eseguito uno shift sinistro di 15 bit per convertirli in formato a virgola fissa. Questi coefficienti scalati sono poi stati salvati in tre diversi file chiamati **lowpass300hz.dat**, **bandpass_300_3kHz.dat** e **bandpass_3k_20kHz.dat**.

Chapter 3

Implementazione dei filtri FIR in Vitis HLS

Dopo aver progettato i filtri in MATLAB e ottenuto i relativi coefficienti, la fase successiva ha previsto l'implementazione dei filtri usando l'High Level Synthesis all'interno dell'ambiente **Vitis HLS**. Lo scopo è stato quello di generare i diversi blocchi hardware rappresentanti i tre diversi filtri FIR, utilizzabili successivamente in Vivado come IP personalizzati.
vedremo in particolare l'implementazione del filtro passa-basso, gli altri filtri seguono una struttura identica.

3.1 header "fir.h"

il file **fir.h** contiene i parametri necessari all'implementazione:

```
#ifndef _FIR_H_
#define _FIR_H_
#include "ap_cint.h"

// number of taps is N+1
#define N 136  //lowpass filter
//#define N 150 //300-3000Hz and 3000-20000Hz bandpass filter
#define TEST_SAMPLES N+10 // 10 more samples then N to run the
   impulse response test

        typedef short coef_t;
        typedef short data_t;
        typedef int25 acc_t; //enough for all our filters

#endif
```

- N : il numero di coefficienti meno 1 per ogni filtro; i filtri passa-banda risultano avere 151 coefficienti invece che 150;

- coef_t, data_t: tipi di dato per i nostri coefficienti e dati in ingresso, in questo caso definiti come short e quindi a 16 bit;
- acc_t: tipo di dato per l'accumulatore, è stato impostato a 25 bit bit poiché è il numero minimo di bit per contenere la somma di 150 interi a 16 bit con segno.

3.2 Funzione top-level

La funzione principale cambierà nome in base al filtro da implementare, nel caso del filtro passa-basso è:

```
void FIR_lowpass_300Hz (data_t *y, data_t x)
```

Questa funzione riceve in ingresso un campione del segnale audio (`x`) e restituisce in uscita il campione filtrato (`*y`).
Il nome della funzione top-level va cambiato per ogni IP, altrimenti verranno considerati tutti duplicati dello stesso IP.

3.3 Interfacciamento AXI

Le seguenti direttive HLS sono state inserite per generare un'interfaccia AXI-Lite compatibile con il sistema embedded:

```
#pragma HLS INTERFACE mode=s_axilite bundle=fir_io port=return
#pragma HLS INTERFACE mode=s_axilite bundle=fir_io port=y
#pragma HLS INTERFACE mode=s_axilite bundle=fir_io port=x
```

Questo permette di collegare il filtro al processore ARM tramite un'interfaccia AXI, consentendo al software di controllare il blocco hardware.

3.4 Integrazione dei coefficienti

I coefficienti del filtro, progettati su MATLAB, sono stati salvati in un file esterno (`ad esempio lowpass_300hz.dat`) e inclusi nel codice:

```
const coef_t c[N+1] = {
        #include "lowpass_300hz.dat"
        //#include "bandpass_300_3kHz.dat"
        //#include "bandpass_3k_20kHz.dat"
};
```

Questo array contiene i valori dei coefficienti per ogni FIR, utilizzati nel calcolo del filtro. Per implementare gli altri filtri basta commentare e decommentare le apposite istruzioni.

3.5 Shift register e MAC operation

Per eseguire il filtraggio, viene dichiarato un array che farà da `shift register`, che memorizza i campioni precedenti del segnale:

```
static data_t shift_reg[N];
```

è di tipo statico in modo da mantenere i valori al suo interno tra una chiamata di funzione e l'altra, risultando quindi in un registro a scorrimento in hardware.
La funzione esegue quindi una serie di operazioni **MAC (multiply and accumulate)**, applicando ciascun coefficiente al rispettivo campione ritardato:

```
acc = (acc_t)shift_reg[N-1] * (acc_t)c[N];
for (i = N-1; i != 0; i--) {
    acc += (acc_t)shift_reg[i-1] * (acc_t)c[i];
    shift_reg[i] = shift_reg[i-1];
}
acc += (acc_t)x * (acc_t)c[0];
shift_reg[0] = x;
```

Infine, l'output viene scalato con uno shift destro per riportare il valore nell'intervallo corretto (operazione di **normalizzazione** in virgola fissa):

```
*y = acc >> 15;
```

3.6 Risultato della sintesi

Una volta completato il codice e impostate le direttive si è proseguito con la sintesi di ogni filtro, con i seguenti risultati:

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
FIR_lowpass_300Hz				-	147	1,470E3	-	148	-	no	2	2	233	390	0
FIR_lowpass_300Hz_Pipeline_loop				-	140	1,400E3	-	140	-	no	1	1	115	114	0

Figure 3.1: caratteristiche del filtro passa-basso sintetizzato

Per il filtro passa-basso:

- latenza totale: 147 cicli di clock;
- blocchi di BRAM: 2;
- blocchi DSP: 2;
- flip flop: 233;

- look-up tables: 390.

il sintetizzatore è in grado di dedurre automaticamente quando è possibile applicare tecniche di pipelining, come fatto sopra, permettendo una velocità molto maggiore tra un'uscita e la seguente.

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
FIR_bandpass_300_3kHz				-	158	1,580E3	-	159	-	no	2	1	243	401	0
FIR_bandpass_300_3kHz_Pipeline_loop				-	154	1,540E3	-	154	-	no	1	1	115	114	0
loop				-	152	1,520E3	5	1	149	yes	-	-	-	-	-

Figure 3.2: caratteristiche del filtro passa-banda sintetizzato

Per entrambi i filtri passa-banda, poiché possiedono caratteristiche identiche e differiscono solo per i coefficienti:

- latenza totale: 153 cicli di clock;
- blocchi di BRAM: 2;
- blocchi DSP: 1;
- flip flop: 243;
- look-up tables: 401.

Una volta sintetizzati si procede all'esportazione RTL dei vari IP, che verranno poi usati in Vivado.

Chapter 4

Progettazione della piattaforma hardware in Vivado

Dopo aver sintetizzato i filtri FIR in Vitis HLS, la fase successiva del progetto ha previsto la creazione della piattaforma hardware all'interno dell'ambiente **Vivado**, utilizzando un approccio a blocchi tramite il tool `IP Integrator`.
L'obiettivo di questa fase è stato realizzare un sistema hardware completo e funzionante, compatibile con la scheda **Zybo Zynq-Z7-20**, che integrasse i blocchi logici necessari per l'elaborazione audio in tempo reale.

4.1 Descrizione dell'architettura

Nel diagramma del sistema (mostrato in figura), sono stati inseriti i seguenti blocchi principali:

- **ZYNQ7 Processing System**: rappresenta il processore ARM dual-core integrato nel chip Zynq. Questo componente funge da unità di controllo centrale, configurato per gestire la comunicazione AXI, il clock e le periferiche.

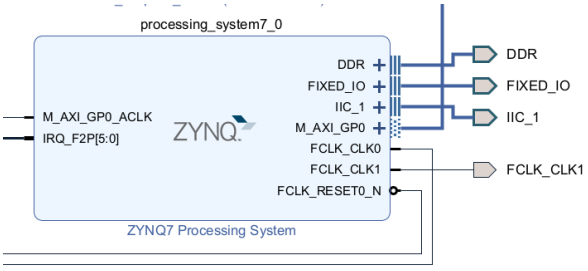

Figure 4.1: processing system ZYNQ

- **Blocchi FIR personalizzati**: sono state inserite due copie di ciascuno dei tre filtri FIR (passa-basso, passa-banda, passa-alto), per un totale di 6 blocchi. Ogni coppia di filtri è assegnata rispettivamente ai canali audio **sinistro (left)** e **destro (right)**.

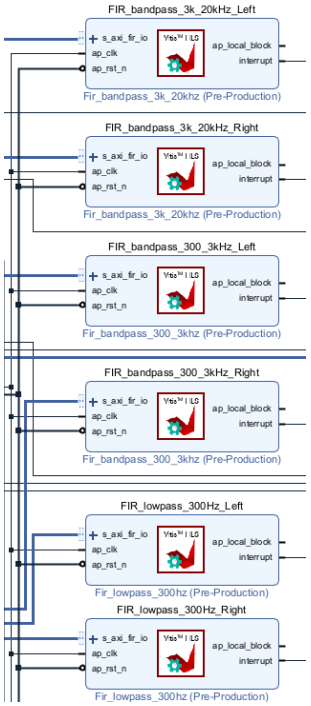

Figure 4.2: blocchi FIR per il filtraggio audio

- **AXI Interconnect**: consente la comunicazione tra il Processing System, l'AXI GPIO e i blocchi FIR tramite l'interfaccia AXI4-Lite. Questo modulo gestisce la trasmissione dei segnali di controllo e dati tra master (CPU) e slave (filtri + AXI GPIO).

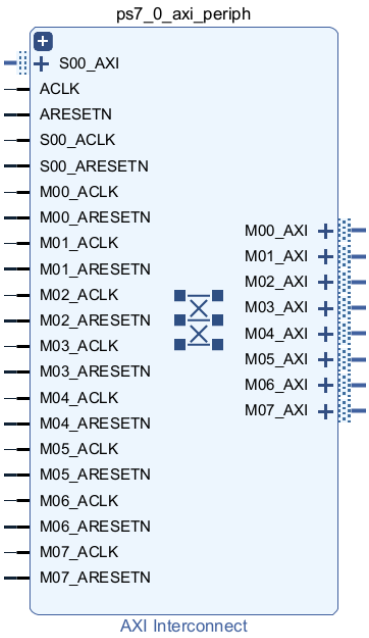

Figure 4.3: blocco AXI interconnect

- **Processor System Reset**: gestisce il segnale di reset per tutti i componenti del design. È collegato al sistema di clock del Processing System e assicura che il reset sia distribuito

correttamente ai blocchi collegati.

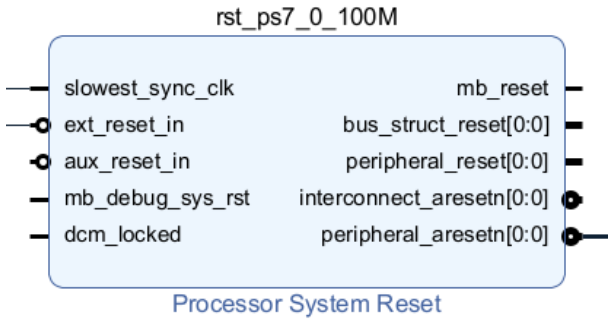

Figure 4.4: blocco per il reset sincrono dell'intero sistema

- **AXI GPIO**: questo modulo consente l'interfaccia con segnali digitali esterni (come switch o pulsanti). In questo progetto è utilizzato per leggere lo stato dei **tre switch** che controllano l'attivazione dei filtri e per abilitare il segnale MUTE, attivo basso e quinidi da impostare alto per abilitare l'audio.

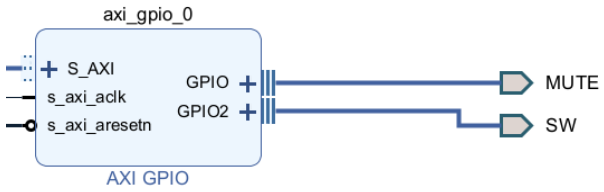

Figure 4.5: blocco AXI GPIO per la lettura degli switch della scheda e abilitazione del segnale MUTE

- **Concat (Concatenatore)**: utilizzato per unire più segnali di tipo interrupt in un unico vettore, da collegare al sistema di interrupt del Processing System. È utile per gestire eventi provenienti da più blocchi, come i filtri FIR.

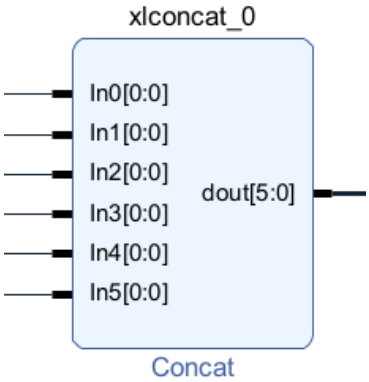

Figure 4.6: blocco per la concatenazione degli interrupt

- **Zybo Audio Controller**: è un IP core dedicato alla gestione dell'input/output audio tramite il codec audio presente sulla scheda Zybo. Questo modulo si occupa di ricevere i

campioni audio dal microfono o dalla linea in ingresso e di inviarli in uscita verso cuffie o speaker. Comunica con il processore e con i blocchi FIR per elaborare il segnale audio in tempo reale.

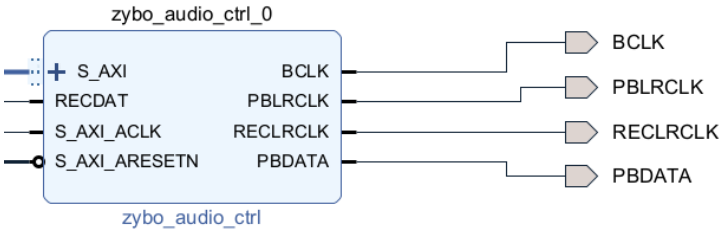

Figure 4.7: blocco per l'interfacciamento al codec SSM2603

l'architettura completa è presentata in figura:

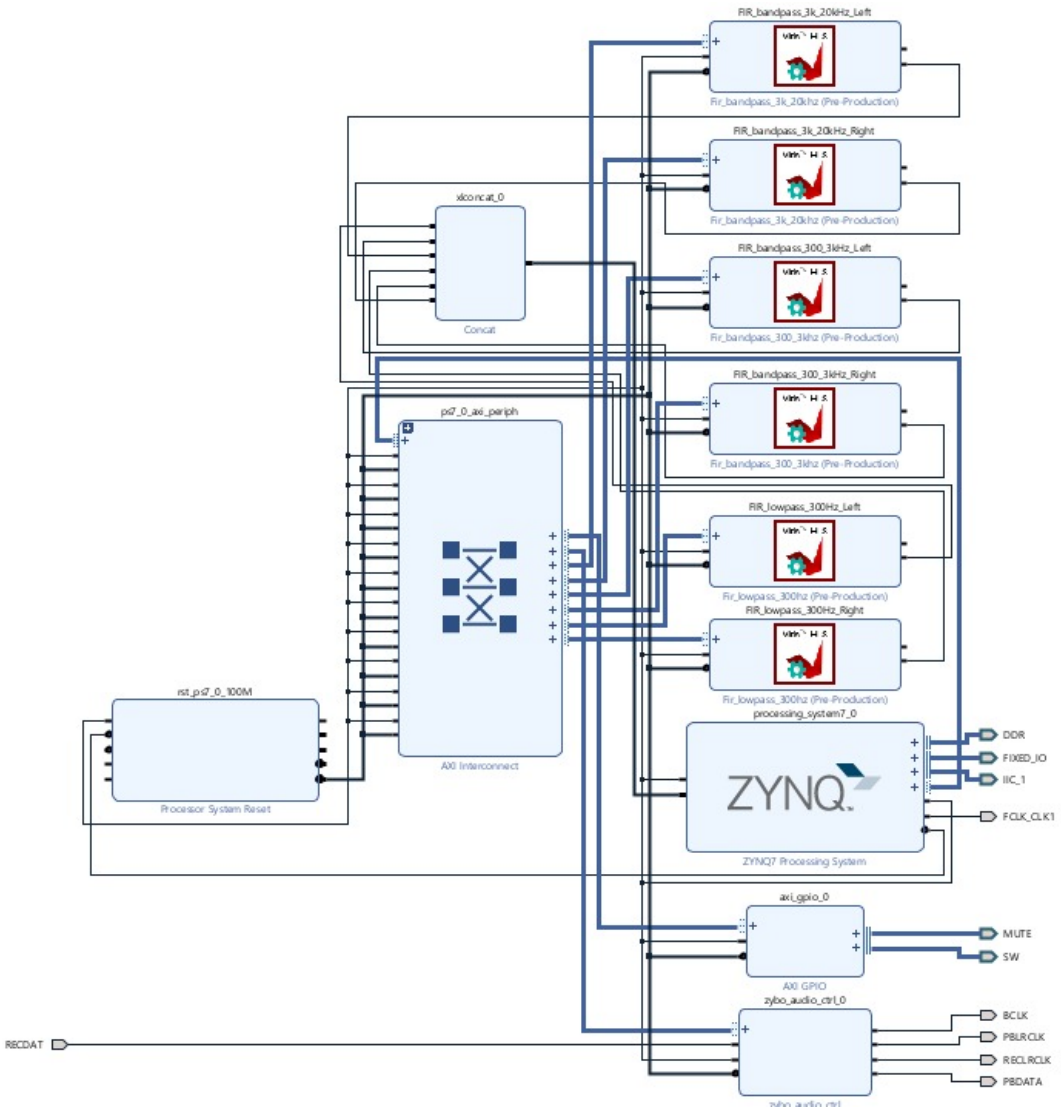

Figure 4.8: schema a blocchi della piattaforma hardware

4.2 configurazione PS

Il processing system ZYNQ è stato configurato in questo modo:

- periferiche attive:
  - UART1: periferica UART usata per comunicare tra il SoC e un computer;
  - IIC1: periferica I^2C dedicata alla comunicazione tra PS e il codec audio tramite I^2S
- PL Fabric clocks:
  - FCLK_0 = 100 MHz; il clock principale del sistema;
  - FCLK_1 = 12.288 MHz: clock collegato all'ingresso del codec audio per permettere il campionamento a 48 kHz come descritto dal datasheet del codec SSM2603;
- interfacce AXI:
  - M_AXI_GP0: interfaccia AXI che fa da master per tutti gli slave AXI presenti;
- interrupt:
  - IRQ_F2P[15:0]: linea di interrupt a 16 bit che riceve in ingresso gli interrupt dei vari filtri.

4.3 aggiunta di vincoli alla piattaforma

Il prossimo passo consiste nel definire le periferiche di input/output per il sistema fisico, in modo da poter interagire con la piattaforma; bisogna quindi creare un file XDC che andrà a definire tutti i vari I/O. Il file usato è il seguente:

```
##Clock signal
set_property -dict { PACKAGE_PIN R19   IOSTANDARD LVCMOS33 } [
    get_ports { BCLK }];
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [
    get_ports { FCLK_CLK1 }];

##Switches
set_property -dict { PACKAGE_PIN G15   IOSTANDARD LVCMOS33 } [
    get_ports { SW_tri_i[0] }]; #IO_L19N_T3_VREF_35 Sch=SW0
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [
    get_ports { SW_tri_i[1] }];  #IO_L24P_T3_34 Sch=SW1
set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [
    get_ports { SW_tri_i[2] }]; #IO_L4N_T0_34 Sch=SW2
#set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [
    get_ports { sw[3] }]; #IO_L9P_T1_DQS_34 Sch=SW3


##Buttons
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [
    get_ports { PBDATA }]; #IO_L20N_T3_34 Sch=BTN0
#set_property -dict { PACKAGE_PIN P16   IOSTANDARD LVCMOS33 } [
    get_ports { btn[1] }]; #IO_L24N_T3_34 Sch=BTN1
#set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [
    get_ports { btn[2] }]; #IO_L18P_T2_34 Sch=BTN2
#set_property -dict { PACKAGE_PIN Y16   IOSTANDARD LVCMOS33 } [
    get_ports { btn[3] }]; #IO_L7P_T1_34 Sch=BTN3


##LEDs
#set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [
    get_ports { led[0] }]; #IO_L23P_T3_35 Sch=LED0
#set_property -dict { PACKAGE_PIN M15   IOSTANDARD LVCMOS33 } [
    get_ports { led[1] }]; #IO_L23N_T3_35 Sch=LED1
#set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 } [
    get_ports { led[2] }]; #IO_0_35=Sch=LED2
```

```
#set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [
    get_ports { led[3] }]; #IO_L3N_T0_DQS_AD1N_35 Sch=LED3

##I2S Audio Codec
set_property -dict { PACKAGE_PIN T19   IOSTANDARD LVCMOS33 } [
    get_ports PBLRCLK]; #IO_25_34 Sch=AC_MCLK
set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [
    get_ports MUTE_tri_o]; #IO_L23N_T3_34 Sch=AC_MUTEN
set_property -dict { PACKAGE_PIN R16    IOSTANDARD LVCMOS33 } [
    get_ports RECDAT];

##Audio Codec/external EEPROM IIC bus
set_property -dict { PACKAGE_PIN N18   IOSTANDARD LVCMOS33 } [
    get_ports IIC_1_scl_io]; #IO_L13P_T2_MRCC_34 Sch=AC_SCL
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [
    get_ports IIC_1_sda_io]; #IO_L23P_T3_34 Sch=AC_SDA
set_property -dict { PACKAGE_PIN Y18   IOSTANDARD LVCMOS33 } [
    get_ports { RECLRCLK }]; #IO_L17P_T2_34 Sch=JB3_P
```

le linee commentate sono state lasciate nel file per possibili revisioni future della piattaforma.

4.4 Esportazione della piattaforma

Una volta completato il design a blocchi, è stato generato il file `.xsa` (Xilinx Shell Archive), contenente la descrizione dell'intero hardware. Questo file è stato poi esportato e utilizzato successivamente all'interno di **Vitis** per sviluppare il software di controllo in linguaggio C.

Chapter 5

Implementazione su Vitis

Per quanto riguarda la parte software della piattaforma, è stata realizzata un'applicazione bare-metal nell'ambiente di sviluppo **Vitis**, configurata per girare sull'architettura della board **Zybo Z7**. Lo scopo principale è ricevere un segnale audio in ingresso, filtrarlo tramite i tre blocchi FIR (banda bassa, media e alta) a seconda del valore degli switch, e restituire in uscita il segnale ricostruito. Di seguito vengono descritti i principali aspetti implementativi che possiamo trovare all'interno del file `main.c`.

5.1 Inizializzazione dei filtri

I tre blocchi FIR sono istanziati in maniera indipendente, uno per ciascun canale (sinistro e destro) e per ciascuna banda di frequenza. L'inizializzazione avviene tramite la funzione `hls_fir_init()` (il nome è diverso per ogni filtro), che utilizza le funzioni `LookupConfig` e `CfgInitialize` fornite dai driver di Vitis per ciascun blocco IP.

Listing 5.1: Inizializzazione di un filtro FIR

```
XFir_lowpass_300hz_Config *cfg_low;
cfg_low = XFir_lowpass_300hz_LookupConfig(
    XPAR_FIR_LOWPASS_300HZ_LEFT_DEVICE_ID);
XFir_lowpass_300hz_CfgInitialize(&HlsFir_low_left, cfg_low);
```

5.2 Gestione delle interruzioni

Ogni blocco HLS genera un'interruzione al termine del calcolo del filtro. Le ISR (*Interrupt Service Routine*) sono registrate con il `GIC` (Generic Interrupt Controller) e ciascuna imposta un flag globale al termine dell'elaborazione.

Listing 5.2: Esempio di ISR

```
void hls_fir_low_left_isr(void *InstancePtr) {
```

```
    XFir_lowpass_300hz_InterruptClear((XFir_lowpass_300hz *)
        InstancePtr, 1);
    ResultLow_left = 1;
}
```

5.3 Sezione principale del firmware

il codice nella funzione **main** consiste nell'inizializzare le varie periferiche presenti nel sistema:

Listing 5.3: inizializzazione delle periferiche

```
//Configure the IIC data structure
IicConfig(XPAR_XIICPS_0_DEVICE_ID);

// Initialize PS7 timer
TimerInitialize();

//Configure the Line in and Line out ports.
LineinLineoutConfig();
```

le tre funzioni attivano rispettivamente la periferica I^2S, un timer per la configurazione del codec e il codec audio impostando gli ingressi e uscite opportuni.

Listing 5.4: inizializzazione e configurazione IIC

```
unsigned char IicConfig(unsigned int DeviceIdPS){
    XIicPs_Config *Config;
    int Status;

    //Initialize the IIC driver so that it's ready to use
    //Look up the configuration in the config table, then
        initialize it.
    Config = XIicPs_LookupConfig(DeviceIdPS);
    if(NULL == Config) {
            return XST_FAILURE;
    }

    Status = XIicPs_CfgInitialize(&Iic, Config, Config->
        BaseAddress);
    if(Status != XST_SUCCESS) {
            return XST_FAILURE;
    }

    //Set the IIC serial clock rate.
```

```
        XIicPs_SetSClk(&Iic, IIC_SCLK_RATE);

        return XST_SUCCESS;
}
```

Listing 5.5: inizializzazione e configurazione timer

```
int TimerInitialize(void)
{
        int Status;
        XScuTimer *TimerInstancePtr = &TimerInstance;
        XScuTimer_Config *ConfigTmrPtr;

        /* Initialize the Scu Private Timer driver. */
        ConfigTmrPtr = XScuTimer_LookupConfig(
           XPAR_PS7_SCUTIMER_0_DEVICE_ID);

        /* This is where the virtual address would be used, this
           uses physical address. */
        Status = XScuTimer_CfgInitialize(TimerInstancePtr,
           ConfigTmrPtr,
                        ConfigTmrPtr->BaseAddr);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        /* Disable Auto reload mode and set prescaler to 1 */
        XScuTimer_SetPrescaler(TimerInstancePtr, 0);

        return Status;
}
```

Listing 5.6: inizializzazione e configurazione codec

```
void LineinLineoutConfig() {

        // software reset
        AudioWriteToReg(R15_SOFTWARE_RESET, 0x000);
        TimerDelay(75000);
        // power mgmt: 0_00110010=>0,Power up, power up, OSC dn, out
            off, DAC up, ADC up, MIC off, LineIn up
        //        AudioWriteToReg(R6_POWER_MANAGEMENT, 0x030);
        AudioWriteToReg(R6_POWER_MANAGEMENT, 0x032);
```

```
        // left ADC Input: 0_01010111=>0,mute disable, Line volume 0
            dB
        AudioWriteToReg(R0_LEFT_ADC_INPUT,0x017);
        // right ADC Input: 0_00010111=>0,mute disable, Line volume
           0 dB
        AudioWriteToReg(R1_RIGHT_ADC_INPUT,0x017);
        AudioWriteToReg(R2_LEFT_DAC_VOLUME,0x079);
        AudioWriteToReg(R3_RIGHT_DAC_VOLUME,0x079);
        // analog audio path: 0_00010010=>0,-6 dB side attenuation,
           sidetone off, DAC selected, bypass disabled, line input,
           mic mute disabled, 0 dB mic
        AudioWriteToReg(R4_ANALOG_AUDIO_PATH, 0x012);
        // digital audio path: 0_00000000=>0_000, clear offset, no
           mute, no de-emphasize, adc high-pass filter enabled
        AudioWriteToReg(R5_DIGITAL_AUDIO_PATH, 0x000);
        // digital audio interface: 0_00001110=>0, BCLK not inverted
           , slave mode, no l-r swap, normal LRC and PBRC, 24-bit,
           I2S mode
        AudioWriteToReg(R7_DIGITAL_AUDIO_INTERFACE, 0x00A);
        TimerDelay(75000);
        // Digital core:0_00000001=>0_0000000, activate core
        AudioWriteToReg(R9_ACTIVE, 0x001);
        // power mgmt: 0_00100010 0_Power up, power up, OSC dn, out
           ON, DAC up, ADC up, MIC off, LineIn up
        AudioWriteToReg(R6_POWER_MANAGEMENT, 0x022); // power mgmt:
           001100010 turn on OUT
}
```

Listing 5.7: inizializzazione e configurazione dei filtri

```
// Initialize all six FIR filter IPs
    status = hls_fir_init(
        &HlsFir_low_left, &HlsFir_mid_left, &HlsFir_high_left,
        &HlsFir_low_right, &HlsFir_mid_right, &HlsFir_high_right
    );
    if (status != XST_SUCCESS) {
        xil_printf("ERROR: HLS FIR initialization failed.\r\n");
        return XST_FAILURE;
    }
```

Listing 5.8: abilitazione degli interrupt per i filtri, il codice mostrato è ripetuto per ogni singolo filtro sinistro e destro

```
XFir_lowpass_300hz_InterruptEnable(&HlsFir_low_left, 1);
XFir_lowpass_300hz_InterruptGlobalEnable(&HlsFir_low_left);
```

Listing 5.9: chiamata alla funzione ”filter_or_bypass_input”, che consiste nell'elaborazione dell'ingresso audio e restituisce poi l'uscita, è un loop infinito, quindi non raggiungerà mai ”return 0”.

```
filter_or_bypass_input();
return 0;
```

5.4 Elaborazione audio in tempo reale

La funzione `filter_or_bypass_input()` gestisce l'elaborazione del segnale audio in tempo reale. Il segnale viene acquisito tramite i registri I2S, convertito in formato a 16 bit, e inoltrato ai filtri FIR in base allo stato dei tre switch (`SW0`, `SW1`, `SW2`). Si inizia dichiarando tutte le variabili interne utili alla funzione:

Listing 5.10: inizializzazione delle variabili interne della funzione

```
unsigned long u32DataL, u32DataR, u32Temp;
short inL, inR;
short y_lowL = 0, y_midL = 0, y_highL = 0;
short y_lowR = 0, y_midR = 0, y_highR = 0;
short outL = 0, outR = 0;
```

si crea poi un loop infinito con:

```
while{1} {
```

In questo loop avverrà l'elaborazione vera e propria.
Si comincia aspettando che arrivi un campione audio in ingresso, leggendo un registro tramite AXI:

```
do {
    u32Temp = Xil_In32(I2S_STATUS_REG);
} while (u32Temp == 0);
```

Una volta che il registro di stato è diverso da 0 vuol dire che è stato ricevuto un campione in ingresso, si procede quindi resettando il bit di ready nel registro del codec audio, poi si assegna il campione alle due variabili *u32DataL* e *u32DataR* per separare i canali sinistro e destro; queste variabili vengono poi spostate a destra di 8 bit per convertirle in interi a 16 bit in modo da essere mandate in ingresso ai nostri filtri:

```
Xil_Out32(I2S_STATUS_REG, 0x00000001); // Clear ready bit
u32DataL = Xil_In32(I2S_DATA_RX_L_REG);
u32DataR = Xil_In32(I2S_DATA_RX_R_REG);

inL = (short)(u32DataL >> 8);  // 24-bit to 16-bit
inR = (short)(u32DataR >> 8);
```

Si continua leggendo il valore degli switch dal registro dell'AXI GPIO, per decidere poi quale banda o bande far passare:

- `SW0`: attiva la banda bassa;
- `SW1`: attiva la banda media;
- `SW2`: attiva la banda alta.

```
u32Temp = Xil_In32(XPAR_AXI_GPIO_0_BASEADDR + 8);
int enable_low  = (u32Temp >> 0) & 0x1;
int enable_mid  = (u32Temp >> 1) & 0x1;
int enable_high = (u32Temp >> 2) & 0x1;
```

Se almeno uno degli switch è alto, si procede al filtraggio, prima di tutto resettando le flag per gli interrupt dei filtri;

```
if (enable_low || enable_mid || enable_high)
{
    // Reset result flags
    ResultLow_left = ResultMid_left = ResultHigh_left = 0;
    ResultLow_right = ResultMid_right = ResultHigh_right = 0;
```

Si controlla poi per ogni filtro se lo switch corrispondente è attivo, se sì allora si imposta l'ingresso con la variabile corrispondente e si attiva il filtro(il codice è ripetuto e modificato propriamente per ogni singolo filtro):

```
if (enable_low) {
    XFir_lowpass_300hz_Set_x(&HlsFir_low_left, inL);
    XFir_lowpass_300hz_Start(&HlsFir_low_left);
}
```

Fatto ciò si aspetta che ogni filtro abbia finito e mandato il suo dato in uscita, se il filtro è attivo allora si prende il suo valore in uscita, altrimenti la variabile corrispondente viene impostata a zero:

```
// Wait for results (LEFT)
if (enable_low) while (!ResultLow_left);
if (enable_mid) while (!ResultMid_left);
if (enable_high) while (!ResultHigh_left);
```

```
// Wait for results (RIGHT)
if (enable_low) while (!ResultLow_right);
if (enable_mid) while (!ResultMid_right);
if (enable_high) while (!ResultHigh_right);

// Read results
y_lowL  = enable_low  ? XFir_lowpass_300hz_Get_y(&
    HlsFir_low_left)  : 0;
y_midL  = enable_mid  ? XFir_bandpass_300_3khz_Get_y(&
    HlsFir_mid_left)  : 0;
y_highL = enable_high ? XFir_bandpass_3k_20khz_Get_y(&
    HlsFir_high_left) : 0;

y_lowR  = enable_low  ? XFir_lowpass_300hz_Get_y(&
    HlsFir_low_right)  : 0;
y_midR  = enable_mid  ? XFir_bandpass_300_3khz_Get_y(&
    HlsFir_mid_right)  : 0;
y_highR = enable_high ? XFir_bandpass_3k_20khz_Get_y(&
    HlsFir_high_right) : 0;
```

Dopo di che si procede sommando tutte le componenti del segnale d'ingresso filtrate, eseguendo poi un'operazione di clamping per prevenire casi di overflow prima di convertire di nuovo le variabili a 24 bit con uno shift sinistro di 8:

```
// Sum and clamp
outL = y_lowL + y_midL + y_highL;
outR = y_lowR + y_midR + y_highR;

if (outL > 32767) outL = 32767;
if (outL < -32768) outL = -32768;
if (outR > 32767) outR = 32767;
if (outR < -32768) outR = -32768;

u32DataL = ((unsigned long)outL) << 8;
u32DataR = ((unsigned long)outR) << 8;
}
```

infine, il segnale filtrato e ricostruito(se il filtraggio è stato abilitato, altrimenti si tratta semplicemente del campione originale ricevuto in ingresso) viene mandato in uscita al codec audio alla linea **LINE_OUT** della scheda:

```
        Xil_Out32(I2S_DATA_TX_L_REG, u32DataL);
        Xil_Out32(I2S_DATA_TX_R_REG, u32DataR);
        }
}
```

5.5 Fase di test

Dopo aver implementato tutte le funzionalità necessarie all'interno del progetto, è stata eseguita una fase di test pratico collegando l'uscita audio del sistema a un paio di cuffie. Durante questa fase, sono state verificate le prestazioni dei filtri digitali progettati, sperimentando diverse combinazioni operative: attivazione di ciascun filtro singolarmente, attivazione simultanea di tutti i filtri, attivazione selettiva delle sole bande bassa e media, e infine delle sole bande media e alta.
Tutti i test funzionali hanno dato esito positivo, confermando il corretto comportamento del sistema nei diversi scenari. Tuttavia, è emersa una lieve criticità nel caso in cui siano attivi contemporaneamente il filtro passa-basso (banda bassa) e il filtro passa-banda centrato sulla gamma media. In questa configurazione, si è osservata la comparsa di picchi acustici localizzati in un intervallo di frequenze stimato tra i 300Hz e i 1000Hz. Tali picchi causano improvvisi aumenti di volume, seppur di breve durata, percepiti come fastidiosi all'ascolto.
L'origine di questo comportamento anomalo sembra risiedere nella parziale sovrapposizione tra la banda di transizione del filtro passa-basso (che si estende fino a circa 1000Hz) e la frequenza di inizio della banda passante del filtro medio, posta anch'essa attorno ai 300Hz. Tale sovrapposizione può generare un effetto di somma costruttiva indesiderata tra le componenti di segnale comprese in quella porzione di spettro, con conseguente amplificazione non intenzionale.
Una possibile soluzione a questo problema consisterebbe nella riprogettazione del filtro passa-basso, restringendone la banda passante per ridurre l'interferenza con la banda media. Tuttavia, ciò comporterebbe l'impiego di un numero maggiore di coefficienti per ottenere una transizione più netta tra la banda passante e quella attenuata. Questa modifica implicherebbe un aumento della complessità computazionale e della latenza del sistema, aspetti che devono essere attentamente valutati in base ai vincoli prestazionali e alle risorse disponibili sull'hardware.
In ogni caso, il comportamento osservato non compromette il corretto funzionamento generale dell'equalizzatore, ma rappresenta un utile spunto per futuri miglioramenti del progetto in ottica di ottimizzazione qualitativa.

Chapter 6

Conclusioni

Il progetto dimostra concretamente come sia possibile sfruttare in modo efficace la piattaforma **Zynq** per la realizzazione di un *equalizzatore audio completamente funzionante*, integrando componenti sia hardware che software. In particolare, la distribuzione del carico computazionale tra la **parte logica programmabile (PL)** e la **parte di controllo software (PS)** permette di ottenere prestazioni elevate in termini di efficienza e velocità di elaborazione, pur mantenendo un buon livello di flessibilità e riconfigurabilità.
L'approccio *modulare* adottato nella progettazione rende agevole l'eventuale modifica del sistema, ad esempio per aggiungere nuove bande di filtraggio o per personalizzare la risposta in frequenza in base a specifiche esigenze applicative. Inoltre, la struttura del progetto è stata pensata per essere **scalabile**, ovvero facilmente adattabile a contesti audio più complessi o con requisiti più stringenti, come il processamento multicanale o l'utilizzo in tempo reale su segnali audio ad alta risoluzione.
In sintesi, questo lavoro mette in evidenza i vantaggi derivanti dall'integrazione tra progettazione hardware (tramite *Vitis HLS* e *Vivado*) e software (tramite *Vitis*), mostrando come la piattaforma Zynq rappresenti una soluzione versatile ed efficiente per lo sviluppo di applicazioni embedded nel campo dell'elaborazione del segnale audio.