UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica

Studio e sviluppo di algoritmi di elaborazione del segnale musicale su piattaforma HW SHARC Audio Module Evaluation Board

Study and development of musical signal processing algorithms on HW platform SHARC Audio Module Evaluation Board

Candidato:
Amerigo Aloisi

Relatore:
Prof. Stefano Squartini

Correlatore:
Prof. Leonardo Gabrielli

Anno Accademico 2021-2022

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica

Studio e sviluppo di algoritmi di elaborazione del segnale musicale su piattaforma HW SHARC Audio Module Evaluation Board

Study and development of musical signal processing algorithms on HW platform SHARC Audio Module Evaluation Board

Candidato:
Amerigo Aloisi

Relatore:
Prof. Stefano Squartini

Correlatore:
Prof. Leonardo Gabrielli

Anno Accademico 2021-2022

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica
Via Brecce Bianche – 60131 Ancona (AN), Italy

Ai miei genitori Tiberio e Annamaria

Ringraziamenti

Ringrazio i professori Squartini e Gabrielli che mi hanno dato la possibilità di svolgere l'attività di tirocinio su un tema che da sempre mi appassiona: la musica e il suono in tutte le sue forme. Voglio inoltre ringraziare tutti i miei colleghi e compagni di corso che mi hanno accompagnato in questo percorso ricco di soddisfazioni.

Roseto degli Abruzzi, 11 Giugno 2022

Amerigo Aloisi

Sommario

Questo elaborato ha lo scopo di documentare l'attività di tirocinio svoltasi come atto finale del corso di laurea in Ingegneria Elettronica presso l'Università Politecnica delle Marche. Il tirocinio prevedeva lo studio del funzionamento e il collaudo di una piattaforma hardware dedicata all'elaborazione del segnale audio e musicale, programmabile via software attraverso algoritmi di DSP. Dopo aver introdotto il contesto e l'ambiente di lavoro, sono stati analizzati e testati diversi elementi ed effetti forniti dal costruttore all'interno di due librerie di codice. Infine per concludere il lavoro è stato sviluppato un algoritmo originale più complesso avvalendosi del supporto di un linguaggio di alto livello appositamente sviluppato per le applicazioni audio e compatibile con la piattaforma.

Indice

1 Introduzione **1**

1.1 Digital Signal Processing 1

1.2 Sistemi embedded 1

1.3 Presentazione del lavoro: obiettivi e finalità 2

1.4 Introduzione alla scheda 2

1.4.1 Hardware 2

1.4.2 Software 4

2 Configurazione e funzionamento dell’IDE **5**

2.1 Creazione di un nuovo progetto 5

2.2 Modalità Single/Dual Core 5

2.3 Struttura progetto e funzioni chiave 6

2.4 Buffer Audio 8

2.5 Condivisione dati tra i vari processori 8

2.6 Debugging dell’applicazione 9

2.7 Booting dell’applicazione 9

3 DSP Audio Elements **11**

3.1 Linear DSP Audio Elements 11

3.1.1 Biquad Filter 11

3.1.2 Integer Delay 14

3.1.3 Oscillators 15

3.2 Nonlinear DSP Audio Elements 15

3.2.1 Compressor 17

3.2.2 Clipper 18

3.2.3 Amplitude Modulation 20

3.3 Simple Synth 21

4 Audio Effects **25**

4.1 Wah-wah e Auto-wah 25

4.2 Tremolo 27

4.3 Ring Modulator 27

4.4 Stereo Flanger 28

4.5 Audio Effects Selector 30

5 Sviluppo di algoritmi mediante linguaggio di alto livello Faust **31**
5.1 Linguaggio Faust 31
5.1.1 Presentazione 31
5.1.2 Caratteristiche principali 31
5.1.3 Graphic User Interface 33
5.1.4 Compatibilità tra Faust e SAM 34
5.2 Implementazione di un sintetizzatore FM 35
5.2.1 Richiami teorici sulla sintesi FM 36
5.2.2 Realizzazione mediante Faust 37

6 Conclusioni **41**

Elenco delle figure

1.1 Piattaforma SHARC Audio Module 3
1.2 Modulo di espansione Audio Project Fin 3
1.3 Emulatore JTAG ICE1000 3

2.1 Configurazione Single Core 6
2.2 Latenza modalità Single Core 6
2.3 Configurazione Dual Core 7
2.4 Latenza modalità Dual Core 7

3.1 Schema del filtro biquadratico digitale 12
3.2 File header dell'elemento BIQUAD FILTER : qui sono visibili l'istanza della struttura e la dichiarazione delle funzioni 13
3.3 Schema del delay IIR 14
3.4 Funzione di setup del delay 15
3.5 Differenza tra le uscite di un sistema lineare e di uno non lineare . . 16
3.6 Relazione ingresso uscita tipica di un compressore 17
3.7 Schema generale di un compressore audio 17
3.8 Rappresentazione grafica dell'effetto di attack e release 19
3.9 Funzione smoothstep nelle sue due versioni smooth e smoother . . . 20
3.10 Schema di elaborazione non-lineare mediante sovracampionamento . 20
3.11 Algoritmo per la modulazione di ampiezza 21
3.12 Le quattro componenti dell'ADSR in una forma d'onda 22
3.13 Funzione synth_read 24

4.1 Schema effetto Wah-wah 25
4.2 Funzione autowah_read 26
4.3 Bande laterali inferiore e superiore centrate attorno alla frequenza di portante f_c in seguito all'operazione di ring modulation 28
4.4 Schema effetto ring modulator 28
4.5 Delay frazionario mediante interpolazione 29
4.6 schema effetto flanger 29
4.7 Setup degli oscillatori in quadratura di fase 29
4.8 Funzione process per l'effetto autowah 30

5.1 Esempio di codice FAUST: un generatore di rumore bianco posto in serie a un filtro passa-basso 32

5.2 Esempio 2: generatore di rumore bianco splittato sul parallelo di due filtri (passa-basso e passa-alto) 33
5.3 Stesso schema del primo esempio, con l'aggiunta di una GUI per il controllo della frequenza di taglio del filtro e dell'accensione del generatore. 34
5.4 Mapping MIDI dei controlli dell'Audio Project Fin 35
5.5 Spettro del segnale modulato all'aumentare di β, con f_c=220 Hz e f_m=440 Hz 37
5.6 Sintetizzatore FM realizzato mediante FAUST 38
5.7 Dettaglio della forma d'onda di uscita nella fase di rilascio della nota, realizzato tramite MATLAB 39
5.8 Spettro di ampiezza della nota riprodotta, realizzato tramite MATLAB 39
5.9 Spettrogramma della nota riprodotta, realizzato tramite MATLAB . 40

6.1 Scheda correttamente connessa al computer, stato LED emulatore: verde, pronto per il debugging. 41
6.2 Codice caricato correttamente, stato LED emulatore: viola, in fase di debugging. 42

Capitolo 1

Introduzione

1.1 Digital Signal Processing

L'espressione digital signal processing (spesso abbreviato con "DSP") racchiude al suo interno l'insieme di tutte quelle tecniche che comportano l'elaborazione, la manipolazione e la trasmissione di contenuti informativi sotto forma di segnali digitali, ovvero sequenze ordinate di bit. Le applicazioni del DSP sono immense: multimedia, medicina, telecomunicazioni, industria... esso è alla base di tutte quelle tecnologie che ad oggi caratterizzano la cosiddetta *società dell'informazione*. Il DSP ha subito negli ultimi anni un forte sviluppo, arrivando a prevaricare in moltissimi campi a causa della sinergia tra sviluppo teorico (algoritmi sempre più performanti) e l'emergere di piattaforme e architetture hardware adatte all'implementazione di tali algoritmi. Nel prossimo futuro si prevede che il DSP assumerà un ruolo sempre più importante grazie agli straordinari traguardi raggiunti, impensabili con la tecnologia analogica (basti pensare ai video ad altissima risoluzione o alle trasmissioni spaziali).[1]

1.2 Sistemi embedded

L'utilità degli algoritmi di DSP non sarebbe potuta essere espressa senza piattaforme hardware adeguate per l'effettiva realizzazione fisica di quanto teorizzato. L'elaboratore, nato come macchina di calcolo universale, si è con il tempo evoluto e differenziato in una grande varietà di strumenti e dispositivi, in base all'ambito applicativo. Proprio per questo a fianco al classico personal computer detto *general purpose* (a scopo generico) sono nate piattaforme dedicate e specializzate per una determinata applicazione dette sistemi *embedded*. Queste piattaforme dispongono di risorse limitate in termini di capacità di calcolo rispetto ai chip general purpose (parliamo di alcune centinaia di MHz contro diversi GHz), ma il vantaggio che offrono sta in un significativo risparmio a livello di complessità del chip e di conseguenza anche economico, rendendole molto competitive sul mercato poiché permettono di realizzare dispositivi a prezzi contenuti, cosa che sarebbe stata molto difficile usando un ben più costoso chip ad uso generico. Inoltre questi sistemi sono progettati per garantire un'elevata affidabilità di risposta, ovvero avere con certezza un risultato entro i tempi prestabiliti, aspetto critico per le applicazioni dette in tempo reale (si

pensi al sistema frenante di un automobile). Tutte queste caratteristiche rendono i sistemi embedded una soluzione ottima per applicazioni di tipo DSP, che richiedono l'esecuzione di una grande mole di operazioni relativamente semplici (somme e prodotti) ma rispettando la latenza prestabilita. [2].

1.3 Presentazione del lavoro: obiettivi e finalità

All'interno di questo elaborato verrà presentata, analizzata e testata una piattaforma embedded di ultima generazione dedicata all'elaborazione del segnale audio digitale, settore in cui il DSP è diventato predominante soppiantando quasi tutti i dispositivi analogici. In particolare l'obiettivo è l'implementazione di algoritmi che realizzano effetti musicali e sintesi sonora, questi algoritmi possono emulare mediante tecniche numeriche il comportamento di dispositivi nati come analogici ma anche implementare tecniche tipicamente digitali (come ad esempio la sintesi FM [3]). Tutti questi strumenti, come pedali per chitarra elettrica e sintetizzatori, hanno reso possibile (e continuano a farlo tutt'oggi) la nascita di nuovi e svariati generi di musica, i quali hanno caratterizzato le culture e le società di tutto il mondo a partire dalla metà del secolo scorso.

1.4 Introduzione alla scheda

Di seguito verrò introdotta la piattaforma SHARC Audio Module analizzando le sue funzionalità sia dal punto di vista hardware che software.

1.4.1 Hardware

La *SHARC Audio Module* [4] è una piattaforma hardware espandibile creata per il progetto e lo sviluppo di applicazioni audio in tempo reale, basate sull'elaborazione numerica del segnale (DSP). Il nucleo della scheda è costituito dal processore ADSP-SC589 della famiglia SHARC, prodotto dalla Analog Devices. Questo processore multicore è composto dall'unità centrale di controllo (CPU) ARM Cortex-A5, da 450 MHz, e da due processori DSP, anch'essi da 450 MHz, predisposti per i calcoli numerici a virgola mobile. In aggiunta a questi è presente un acceleratore ottimizzato per il calcolo delle FFT/IFFT in background e una gran quantità di I/O che, supportando un'ampia gamma di ingressi analogici e digitali, rendono la scheda molto versatile.

La scheda è concepita per essere espansa con altri moduli che introducono nuove funzionalità, in particolare per lo studio effettuato in questo documento è stato utilizzato il modulo di espansione *Audio Project Fin*, sviluppato dalla Analog Devices e montato direttamente sulla scheda principale. Questo modulo è stato realizzato per rendere la scheda utilizzabile in tempo reale da musicisti e performer. Esso aggiunge agli I/O già presenti sulla main board un ingresso e un'uscita jack 1/4 e tre MIDI (In,Out,Thru). Inoltre dispone di quattro *pushbuttons* (pulsanti a pressione) e

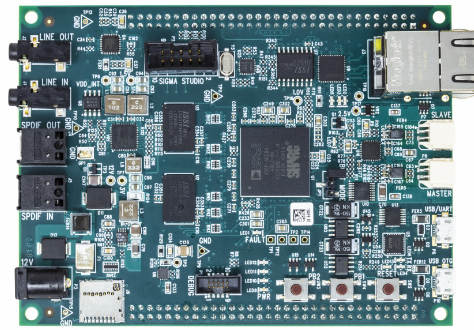

Figura 1.1: Piattaforma SHARC Audio Module

tre potenziometri che possono essere usati per controllare in tempo reale qualsiasi parametro dell'algoritmo di sound processing.

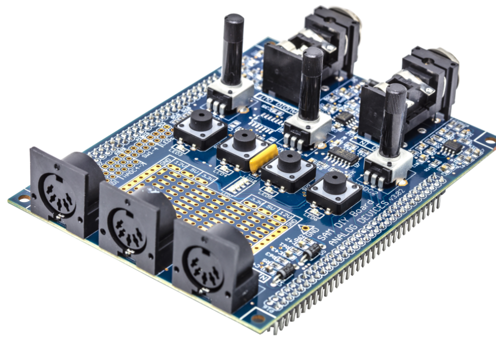

Figura 1.2: Modulo di espansione Audio Project Fin

Infine la scheda arriva dotata dell'emulatore JTAG ICE1000, utile per il debugging e indispensabile per utilizzare la scheda, poiché permette di caricare il codice sviluppato al computer sulla scheda (non è possibile collegarla direttamente).

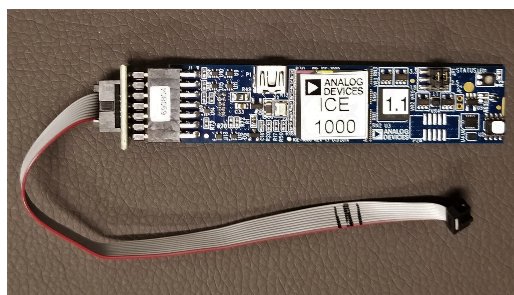

Figura 1.3: Emulatore JTAG ICE1000

1.4.2 Software

Per quanto riguarda l'aspetto software, per programmare la scheda è stato utilizzato l'ambiente di sviluppo integrato (IDE) *Cross Core Embedded Studio* (CCES da qui in poi). Questo programma, di proprietà della Analog Device, permette di lavorare con processori della famiglia SHARC supportando i linguaggi di programmazione C e C++. In particolare per la scheda SAM ci si è avvalsi del *Bare Metal Framework*, un framework dedicato progettato appositamente per interfacciarsi con l'architettura hardware della piattaforma. Il framework è basato su una struttura modulare che rende più facile il passaggio tra diverse configurazioni per i processori e moduli di espansione. Inoltre esso include due vaste librerie di codice scritto e commentato dagli sviluppatori della scheda: *Audio Elements*, che contiene l'implementazione dei mattoni principali per l'elaborazione audio (filtri, oscillatori ...) e *Audio Effects*, in cui si trovano molti effetti classici già pronti all'utilizzo, realizzati a partire dagli elementi di cui sopra. Nel prossimo capitolo saranno spiegate nel dettaglio le funzionalità della piattaforma e come si interfacciano all'interno di CCES e del Bare Metal Framework, allo scopo di fornire una guida per l'utilizzo della scheda.

Capitolo 2

Configurazione e funzionamento dell'IDE

Il contenuto di questo capitolo è una rielaborazione di quanto riportato in [4].

2.1 Creazione di un nuovo progetto

Per poter cominciare a lavorare con la scheda in primo luogo bisogna creare un nuovo progetto all'interno di CCES selezionando la voce `File → New → SHARC Audio Module Bare Metal Project`. Al momento della creazione del progetto è possibile configurare diverse opzioni quali:

- Presenza o meno del modulo di espansione Audio Project Fin.
- Presenza o meno di altri moduli connessi con il bus audio A2B.
- Abilitare o meno il supporto del linguaggio di alto livello *Faust* [5]. Per adesso si consiglia di non spuntare questa opzione per prendere familiarità con il software e le sue librerie. Il linguaggio in questione verrà ripreso in dettaglio nell'ultimo capitolo.
- *Audio Block Size*: lunghezza del blocco di dati che viene processato ad ogni ciclo di elaborazione (solitamente 16 o 32 campioni).
- *Sample Rate*: frequenza di campionamento di lavoro (di default 48000 Hz).
- *Use both Cores*: spuntando questa opzione si decide di lavorare sfruttando entrambi i core DSP, incrementando la potenza di calcolo a disposizione ma anche la latenza (si veda la sezione successiva).

2.2 Modalità Single/Dual Core

Una volta creato il progetto appariranno tre cartelle all'interno del project explorer, ognuna delle quali corrisponde a uno dei tre Core della scheda. La cartella Core 0 contiene il codice per il processore ARM, e si occupa di inizializzare tutti i componenti esterni come ADC,DAC o eventuali espansioni. I Core 1 e 2 sono invece relativi alle due unità di calcolo. Il Core 1, oltre che effettuare i calcoli, si occuppa del DMA

(Direct Memory Access) per spostare i dati audio dalla memoria alle porte di uscita, e per gestire il flusso tra il Core 1 e il Core 2 (se abilitato).

Lavorando con un solo Core il flusso dei dati sarà:
`ADC → SHARC Core 1 → DACs` [Figura 2.1]

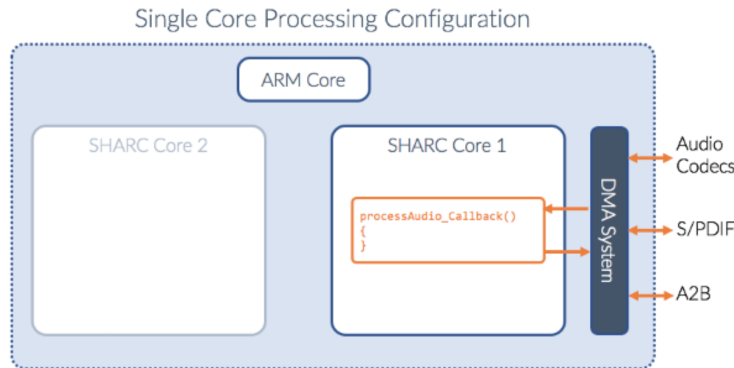

Figura 2.1: Configurazione Single Core

Questa configurazione introduce una latenza pari a 2 volte l'Audio Block Size. [Figura 2.2]

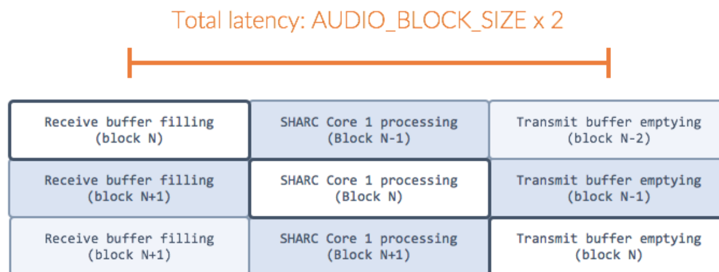

Figura 2.2: Latenza modalità Single Core

Quando si decide di utilizzare anche il secondo processore DSP il flusso sarà invece il seguente:
`ADC → SHARC Core 1 [processing] → SHARC Core 2 [processing] → SHARC Core 1 [final data transfer] → DACs` [Figura 2.3]

Questa configurazione introduce una latenza pari a 4 volte l'Audio Block Size. [Figura 2.4]

Quindi il Core 2 si limita ad effettuare i calcoli, e tutti i dati che riceve dal Core 1 tornano, dopo l'elaborazione, al mittente che si occupa di trasferire definitivamente il segnale alle porte di uscita.

2.3 Struttura progetto e funzioni chiave

In entrambi i Core DSP è presente un file chiamato `callback_audio_processing.cpp`. Questo è il file dove il programmatore può inserire le sue funzioni di elaborazione

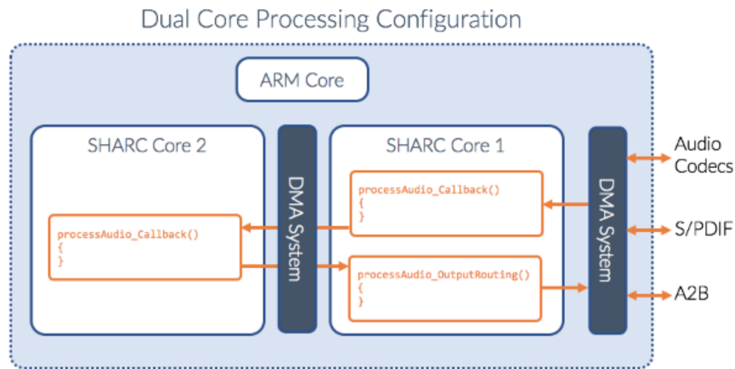

Figura 2.3: Configurazione Dual Core

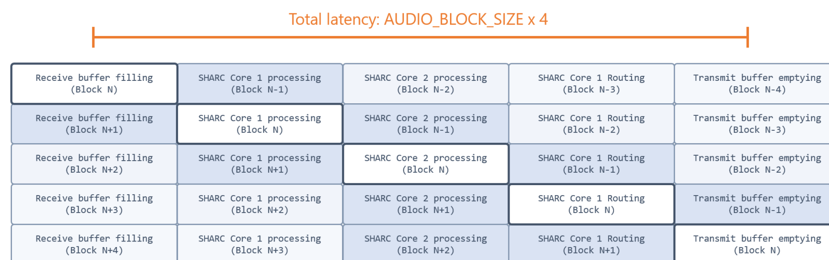

Figura 2.4: Latenza modalità Dual Core

audio. La struttura del file è la stessa per entrambi i Core e sono presenti alcune funzioni chiave:

- `processaudio_setup()` : questa funzione contiene codice di inizializzazione, come ad esempio la generazione di coefficienti per un filtro.

- `processaudio_callback()`: questa funzione contiene l'algoritmo di elaborazione vero e proprio, è presente un ciclo *for* impostato sulla lunghezza dell'Audio Block Size definita al momento della creazione del progetto. Questa funzione viene richiamata ogni volta che un nuovo blocco di dati è pronto per essere processato.

- `processaudio_background_loop()`: se l'algoritmo prevede un'elaborazione in background (es. FFT) essa viene collocata qui.

- `processaudio_mips_overflow()`: gestisce l'eccezione che viene lanciata quando la funzione `processaudio_callback` non riesce a completare le operazioni richieste prima che arrivi il prossimo blocco da processare.

Altri file degni di nota sono:

- `callback_MIDI_message.cpp`: questo file contiene funzioni di elaborazione MIDI.

- `callback_Pushbuttons.cpp`: questo file serve a configurare la risposta del programma alla pressione dei pulsanti (presente solo sul Core 0).

Queste *callback functions* lavorano in maniera ciclica su ogni blocco di dati in ingresso, e sono state create in modo tale da astrarre tutta la meccanica di movimento dei dati audio a basso livello così da essere *hardware agnostic*, ossia indipendenti dal tipo di hardware su cui si programma.

2.4 Buffer Audio

Nel loop di elaborazione principale `processaudio_callback()` è presente un set di buffer non commentati chiamati:

- `audiochannel_0_left_in[]`
- `audiochannel_0_right_in[]`
- `audiochannel_0_left_out[]`
- `audiochannel_0_right_out[]`

Questi buffer puntano alle uscite audio più utilizzate. Di default gli array di ingresso puntano ai dati audio stereo che provengono dall'ADC, mentre i corrispondenti array di uscita puntano al buffer contenente i dati che arrivano al DAC se si lavora in modalità Single Core, altrimenti puntano ai rispettivi buffer di ingresso del secondo processore quando si lavora in modalità Dual Core. Essi possono essere configurati anche per puntare ad altri I/O (ad esempio la porta SPDIF, formato digitale molto usato nei lettori CD) usando buffer appositi piuttosto che quelli più generali riportati sopra, come nell'esempio seguente:

```
 // Invia l'audio ricevuto dall'ingresso S/PDIF al DAC (ADAU1761)
audiochannel_adau1761_0_left_out[i] = audiochannel_spdif_0_left_in[i];
audiochannel_adau1761_0_right_out[i] = audiochannel_spdif_0_right_in[i];
```

2.5 Condivisione dati tra i vari processori

Tutti e tre i processori hanno accesso a un blocco di memoria condivisa da cui possono leggere e scrivere. Questa porzione di memoria è accessibile dal programmatore all'interno del file `common/multicore_shared_memory.h`, presente in ciascun Core. Il file *.h* contiene la dichiarazione della struttura `MULTICORE_DATA` che contiene tutte le variabili comuni a tutti i processori. In aggiunta a questo è presente un omonimo file *.c* che si occupa poi di allocare questa struttura nell'area di memoria condivisa. La struttura può essere personalizzata aggiungendo ulteriori variabili. Ogni Core può leggere queste variabili con una sintassi del tipo: `float x = multicore_data->var_1`. Tra le variabili predefinite sono presenti quelle che sono associate ai controlli presenti sul modulo Audio Project Fin:

- Pushbuttons: per i pushbuttons è presente un flag per ogni pulsante e per ogni Core. In questo modo non possono verificarsi ambiguità dovute al fatto che un Core rilevi un evento di pressione di un pulsante prima dell'altro. Esempio:

  `uint32_t audioproj_fin_sw_1_core1_pressed;`

  `uint32_t audioproj_fin_sw_1_core2_pressed;`

- Potenziometri: i valori di questi potenziometri sono aggiornati dal processore ARM ogni millisecondo e sono sempre valori in virgola mobile nell'intervallo $[0, 1]$. Esempio:

  `float audioproj_fin_pot_hadc0;`

2.6 Debugging dell'applicazione

Quando si è pronti a testare il funzionamento dell'algoritmo si passa alla fase di debugging. In primo luogo bisogna compilare il codice sorgente andando su `Project → Build All`. Se l'operazione va a buon fine viene generato un file eseguibile *.dxe* per ogni Core. In seguito si può passare all'interfaccia di debug cliccando `Debug` in alto a destra nella finestra di CCES. La prima volta che si effettua questa operazione bisogna creare una *Debug Configuration* andando su `Run → Debug As → Application with CrossCore Debugger` (si può fare anche cliccando sull'icona dell'insetto). Il software chiederà di specificare il modello del processore (ADSP-SC589) e dell'emulatore (ICE 1000). Inoltre è possibile configurare altre opzioni come la selezione dei breakpoint automatici. Dopo aver creato una configurazione e aver connesso in maniera corretta la scheda all'emulatore e l'emulatore al computer tramite la porta USB si può procedere cliccando nuovamente sull'icona del Debug. Stabilita la connessione col processore, il programma partirà dal `main()` e si fermerà lì (se non è stato rimosso il breakpoint). Il menu a tendina `Run` contiene le funzioni per far proseguire, terminare, mettere in pausa ... l'applicazione, mentre dalla vista `Variables` è possibile monitorare il valore di qualsiasi variabile. Queste sono solo alcune delle utilità fornite dal software, molte altre sono accessibili da `Window → Show View`.

2.7 Booting dell'applicazione

Quando si lavora in modalità debug il codice sviluppato viene caricato sulla porzione di memoria volatile della scheda che viene resettata a ogni spegnimento. Per salvare il proprio progetto l'applicazione deve essere caricata sulla memoria flash del processore, a questo proposito bisogna creare una *loader image* (LDR). In primo luogo bisogna aver creato il file eseguibile *.dxe* dalla sezione precedente. La procedura per creare l'immagine è disponibile al seguente indirizzo insieme a dei video esplicativi `https://wiki.analog.com/resources/tools-software/`

`crosscore/cces/getting-started/boot-app-sc5xx`. Una volta che è stato generato il file LDR l'ultimo passaggio è quello di caricare il progetto sulla memoria flash, questa operazione è gestita dall'applicazione *Command Line Device Programmer (cldp.exe)*, accessibile tramite Command Line. In alternativa si può affidare l'incarico a CCES che chiamerà automaticamente il *cldp* in seguito a ogni operazione di Build. Per fare questo selezionare con il tasto destro su un progetto qualsiasi e andare su `Properties`. Da lì selezionare `C/C++ Build Settings` e poi `Build Steps`. I comandi vanno inseriti nella casella `Command` sotto `Post Build Steps`. La spiegazione e la sintassi dei comandi da inserire è disponibile allo step 2 della seguente guida: `https://wiki.analog.com/resources/tools-software/crosscore/cces/getting-started/app#step_2write_application_to_flash_memory`.

Capitolo 3

DSP Audio Elements

Una volta che sono chiari gli strumenti e le funzioni forniti da CCES e dal framework dedicato si può finalmente passare alla creazione di algoritmi per applicazioni musicali. In questo capitolo verranno analizzati i componenti elementari che stanno alla base dell'elaborazione audio, fornendo prima una breve trattazione teorica e in seguito un esempio di implementazione commentando il codice contenuto all'interno della libreria *Audio Elements*. La libreria è situata all'interno della cartella `audio_processing` accessibile dai file di progetto dei Core 1 e 2. Per ogni elemento sono presenti due file: un *header (.h)*, contenente i vari *include* delle librerie di sistema, le istanze di enumerazioni e strutture e le dichiarazioni delle varie funzioni, e un *source (.c)* che contiene l'effettiva implementazione delle suddette funzioni. Tutti i calcoli eseguiti in background sono incapsulati in maniera molto efficiente, in questo modo lo sviluppatore ha a disposizione una serie di funzioni che consentono di effettuare rapide modifiche senza doversi preoccupare di tutte le operazioni nascoste dietro, d'altro canto non vi sono *black boxes* in queste librerie e nel caso si volessero comprendere a fondo i layer sottostanti è sempre possibile accedere ai file di interesse. Tutto il codice riportato in questo capitolo e nel seguente è di proprietà della Analog Devices, mentre per le basi teoriche in questo capitolo e nel successivo si è fatto prevalentemente riferimento a [6].

3.1 Linear DSP Audio Elements

In questa sezione saranno introdotti i principali elementi di elaborazione lineare contenuti nella libreria *Audio Elements*.

3.1.1 Biquad Filter

Il filtro biquadratico è un generico filtro IIR del secondo ordine (*second order section o sos*), descritto da un'equazione alle differenze del tipo:

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_1y[n-1] - a_2y[n-2] \tag{3.1}$$

Che corrisponde nel dominio Z alla seguente funzione di trasferimento:

$$H[z] = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \tag{3.2}$$

Andando a variare i 5 coefficienti è possibile ottenere qualsiasi tipo di filtro, questa

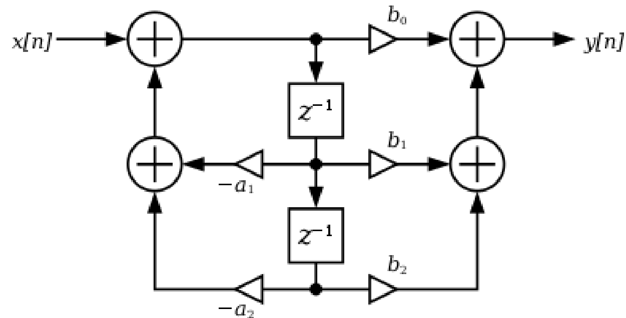

Figura 3.1: Schema del filtro biquadratico digitale

operazione è resa molto semplice infatti basta specificare al momento dell'istanza della struttura il tipo e i parametri del filtro di interesse. Per applicazioni in cui i parametri del filtro vengono controllati in tempo reale gli IIR si dimostrano più efficienti dei FIR a livello computazionale, per questo nell'implementazione degli effetti sarà sempre usato un filtro biquadratico. Inoltre per ottenere filtri di ordine superiore, o combinazioni di diversi tipi di filtri, sarà sufficiente porre in cascata più sezioni del secondo ordine. Di seguito saranno commentate le funzioni di base, visibili in figura 3.2. Queste funzioni sono strutturate in maniera analoga per ogni elemento o effetto quindi nel proseguo della trattazione non verranno più commentate.

- `filter_setup`: questa funzione inizializza l'istanza del filtro, da notare come per il programmatore è sufficiente dichiarare il tipo di filtro che si vuole (BIQUAD_FILTER_TYPE) e tutti gli altri parametri (frequenza di taglio, fattore di risonanza, guadagno) senza preoccuparsi di tutti le altre variabili necessarie per i calcoli intermedi dei coefficienti che è possibile notare dall'istanza della struttura in figura 3.2. L'unica accortezza del programmatore deve essere quella di definire un array per contenere i valori dei coefficienti, dato che la funzione vuole in ingresso un puntatore a questo vettore (`(pm float *) c->filter_coeffs`). Se l'operazione di setup và a buon fine restituisce un `RESULT_BIQUAD` del tipo `BIQUAD_OK`.

- `filter_modify_freq`: questa funzione molto intuitiva si occupa di variare la frequenza di taglio (o centrale) per applicazioni che richiedono filtri tempo-varianti. Essa inoltre gestisce la variazione in modo graduale poiché un cambio brusco di parametri può causare instabilità in un filtro IIR.

- `filter_modify_q`: analogamente alla funzione sopra, questa modifica il fattore Q del filtro (picco di risonanza).

```c
// Instance struct with parameters and state information
typedef struct {

    bool initialized;

    BIQUAD_FILTER_TYPE filter_type;
    BIQUAD_FILTER_TRANSITION_SPEED transition_speed;

    float audio_sample_rate;

    float freq;
    float freq_last;
    float freq_dest;
    float freq_inc;
    uint32_t freq_steps;

    float q;
    float q_last;
    float q_dest;
    float q_inc;
    uint32_t q_steps;

    float gain_db;

    float scaling_factor;
    float scaling_factor_dest;
    float scaling_factor_inc;

    float pm * sos_coeffs;
    float sos_state[3];
    float sos_coeffs_dest[4];
    float sos_coeffs_inc[4];
    uint32_t sos_coeffs_steps;

} BIQUAD_FILTER;

#ifdef __cplusplus
extern "C" {
#endif

RESULT_BIQUAD filter_setup(BIQUAD_FILTER * c, BIQUAD_FILTER_TYPE type,
        BIQUAD_FILTER_TRANSITION_SPEED transition_speed, float pm * sos_coeffs,
        float freq, float q, float gain_db, float audio_sample_rate);

RESULT_BIQUAD filter_modify_q(BIQUAD_FILTER * c, float new_q);

RESULT_BIQUAD filter_modify_freq(BIQUAD_FILTER * c, float new_freq);

void filter_read(BIQUAD_FILTER * c, float * audio_in, float * audio_out,
        uint32_t audio_block_size);
```

Figura 3.2: File header dell'elemento BIQUAD FILTER : qui sono visibili l'istanza della struttura e la dichiarazione delle funzioni

- `filter_read`: questa funzione effettua il filtraggio del segnale, prende come parametri il buffer d'ingresso, il buffer di uscita, l'istanza del filtro e l'audio block size, e non ha tipo di ritorno (void).

3.1.2 Integer Delay

Questa unità implementa un delay di base, un effetto che nella sua versione più generale è descritto da un'equazione del tipo:

$$y[n] = FT \cdot x[n] + FF \cdot x[n-M] + FB \cdot y[n-M] \tag{3.3}$$

Ovvero l'uscita è la somma del segnale di ingresso $x[n]$, una sua versione ritardata $x[n-M]$ e una versione ritardata dell'uscita $y[n-M]$. I parametri di controllo sono:

- `Delay length`: spesso indicata con τ quando espressa in secondi, da cui si ricava la lunghezza M in campioni: $M = \tau/fs$.
- `Feedthrough` (FT): ampiezza del segnale originale.
- `Feedforward` (FF): ampiezza dell'ingresso ritardato.
- `Feedback` (FB): ampiezza dell'uscita ritardata.

Nella versione fornita in questa libreria il parametro di *feedforward* è impostato a 0, mentre parametri su cui si può lavorare sono il *feedback* e il *feedthrough*. In questo caso l'effetto è percepito solo quando il parametro di feedback assume valore diverso da 0 (ma comunque in modulo <1 per ovvi motivi di stabilità). L'unità di delay può allora essere considerata un filtro IIR, all'ingresso della *delay line* arriva un segnale di uscita già affetto da delay, si generano così dei *delay loops* utili per modellare effetti di riflessioni infinite (ad esempio l'echo). Infine è possibile decidere al momento

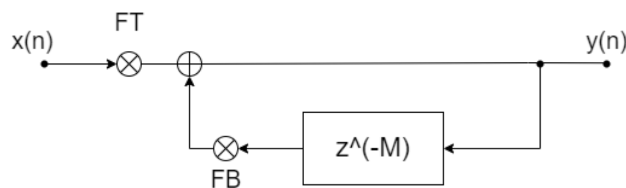

Figura 3.3: Schema del delay IIR

dell'istanza di porre in ingresso all'anello di feedback un filtro passa basso assegnando al parametro `a_coeff` un valore diverso da 0, questo è talvolta usato per ottenere un sound più naturale. Anche in questo caso bisogna passare come parametro il puntatore al buffer di memoria RAM allocato per contenere la delay line, definito con un comando del tipo:

```
float section("seg_sdram") integer_delay_line[INT_DELAY_LEN] .
```

```
RESULT_DELAY delay_setup(DELAY_LPF * c, float * delay_buffer,
        uint32_t delay_buffer_size, uint32_t delay_initial_length,
        float feedback, float feedthrough, float a_coeff);
```

Figura 3.4: Funzione di setup del delay

3.1.3 Oscillators

Questo file contiene la definizione di vari oscillatori elementari, ovvero generatori di forme d'onda che possono essere usate per pilotare i parametri di un effetto o per la sintesi sonora. I quattro tipi di forma d'onda elementari sono:

- `oscillator_sine`: produce una forma d'onda sinusoidale in funzione della variabile temporale t, moltiplicata per 2π in modo tale che l'onda sia periodica per multipli interi di t.
- `oscillator_square`: produce un'onda quadra.
- `oscillator_triangle`: produce un'onda triangolare.
- `oscillator_ramp`: produce una funzione rampa.
- `oscillator_pulse`: produce un treno di impulsi di cui è possibile impostare il parametro `width`, ovvero il tempo in un periodo in cui il segnale assume valore alto, il rapporto tra width e periodo viene detto *duty cicle* (un'onda quadra è un caso particolare di treno di impulsi con duty cicle pari a 0.5).

3.2 Nonlinear DSP Audio Elements

In questa sezione verranno descritti gli elementi per l'elaborazione audio che ricadono nella categoria del *nonlinear processing*. Per i sistemi non lineari non vale il principio di sovrapposizione degli effetti e non è possibile definire una risposta impulsiva. Dato un sistema numerico la cui relazione ingresso-uscita vale $y[n] = f[x[n]]$ allora in generale:

$$f[x_1[n]] + f[x_2[n]] = y_1[n] + y_2[n] \neq f[x_1[n] + x_2[n]] = y_3[n] \tag{3.4}$$

Se consideriamo le sequenze x[n] come segnali a tempo discreto si può affermare che i sistemi non lineari hanno la caratteristica di introdurre nel segnale di uscita componenti spettrali che non erano presenti nel segnale di partenza. In particolare le nuove componenti vengono dette armoniche se multiple della frequenza fondamentale del segnale di ingresso, inarmoniche altrimenti. Consideriamo come segnale di ingresso una singola sinusoide a frequenza f_1:

$$x[n] = Asin(2\pi f_1) \tag{3.5}$$

Un'unità di elaborazione non lineare fornirà in uscita a tale ingresso un segnale del tipo:

$$y[n] = A_0 + A_1 sin(2\pi f_1) + A_2 sin(2 \cdot 2\pi f_1) + ... + A_N sin(N \cdot 2\pi f1) \qquad (3.6)$$

con $A_0 ... A_N \neq 0$. Un sistema lineare descritto da una risposta impulsiva $h[n]$ fornisce invece in uscita a un ingresso sinusoidale un segnale del tipo:

$$y[n] = A_{out} sin(2\pi f_1 + \phi_{out}) \qquad (3.7)$$

Ovvero amplifica il segnale di una quantità pari alla *magnitude response* e sfasa il segnale di una quantità pari alla *phase response*, come noto dalla teoria dei sistemi LTI, ma senza introdurre nuove componenti spettrali. Un parametro fondamentale

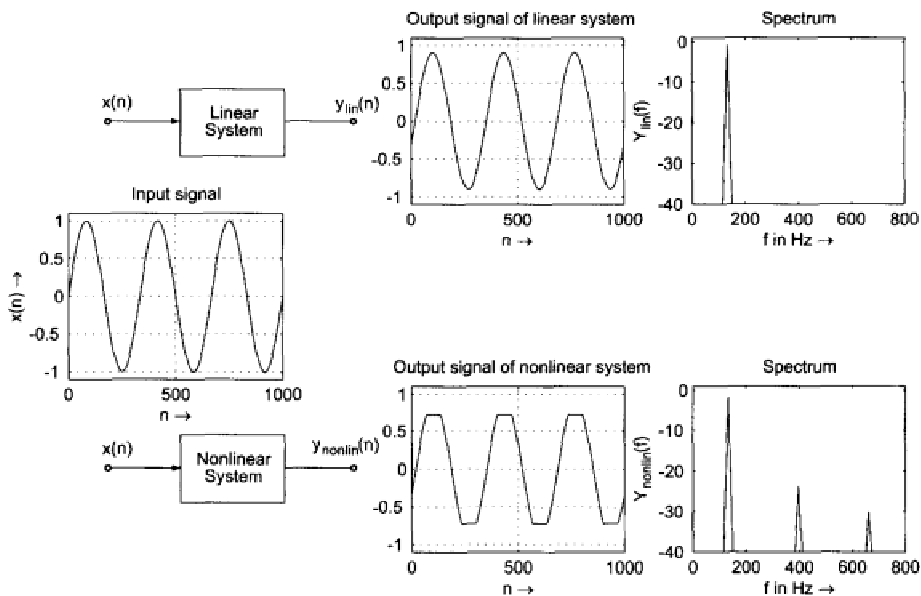

Figura 3.5: Differenza tra le uscite di un sistema lineare e di uno non lineare

ai fini della misura di distorsione armonica introdotta dal sistema non lineare è il *THD: Total Harmonic Distortion*, definito come:

$$THD = \sqrt{\frac{A_2^2 + A_3^2 + ... + A_N^2}{A_1^2 + A_2^2 + ... + A_N^2}} \qquad (3.8)$$

Esso corrisponde al rapporto quadratico tra la somma delle ampiezze di tutte le armoniche oltre la fondamentale e la somma delle ampiezze di tutte le armoniche inclusa la fondamentale. A seconda dell'applicazione questo parametro potrebbe essere desiderato il più basso possibile, come nel caso dei controllori automatici di livello o *dynamic range controllers*, o potrebbe essere volutamente tenuto a un certo livello per introdurre un certo tipo di distorsione tale da caratterizzare il timbro del

suono: è il caso ad esempio degli amplificatori e dei distorsori per chitarra elettrica. Nel seguito saranno descritte due unità non lineari contenute nella libreria Audio Elements per ciascuna delle due possibili finalità oltre che un terzo elemento che non ricade in nessuna delle due categorie ma ha anch'esso la caratteristica di introdurre nuove componenti spettrali.

3.2.1 Compressor

Il compressore è un tipico elemento che rientra tra i controllori automatici del livello del segnale, ovvero amplificatori il cui guadagno è pilotato dal livello del segnale di ingresso. In particolare lo scopo del compressore è quello di ridurre il range dimanico del segnale: esso attenua livelli alti di segnale che superano una certa soglia (*threshold*) mentre lascia inalterate le parti in cui il livello del segnale è sotto soglia. Il risultato è una minor differenza di volume tra le parti più e meno rumorose del segnale che si predispone così a una successiva amplificazione più omogenea. Ai

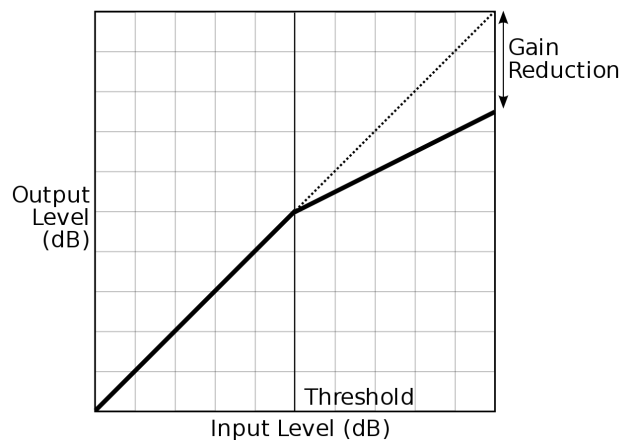

Figura 3.6: Relazione ingresso uscita tipica di un compressore

fini di questo controllo automatico è necessario uno schema di rilevamento del livello del segnale, quale può essere un rivelatore di inviluppo, e un algoritmo che determini il guadagno a partire dall'ampiezza misurata. La relazione ingresso-uscita è espressa di solito da una curva statica non-lineare $y[n] = f[x[n]]$ (Figura 3.6). Lo schema è

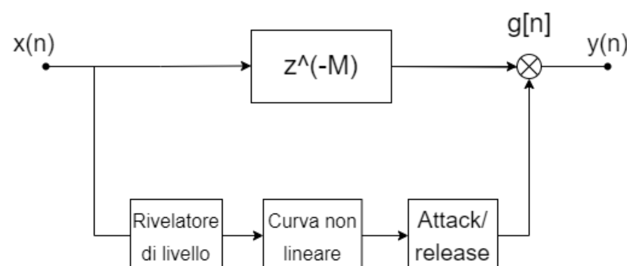

Figura 3.7: Schema generale di un compressore audio

quello riportato in figura 3.7, la parte inferiore, composta dal rivelatore di inviluppo e l'elaborazione conseguente per derivare il guadagno $g[n]$, prende il nome di *sidechain*. Solitamente il percorso diretto che porta il segnale di ingresso prevede un delay che ritarda il segnale del tempo necessario alla sidechain per effettuare le operazioni. Per la misurazione del livello del segnale viene calcolato l'RMS (valore quadratico medio) su un intervallo predefinito (100 ms nell'implementazione fornita). In seguito per determinare il guadagno viene applicata la funzione non-lineare, il risultato viene poi adattato in base ai parametri di `attack` e `release` (Figura 3.8). Nella fase successiva al calcolo del fattore di guadagno $g[n]$ si lavora con unità logaritmiche, quindi la relazione $y[n] = g[n] \cdot x[n - D]$ in dB diventa $Y = X + G$. Il motivo di questa scelta, oltre alla semplificazione dei calcoli, è che l'orecchio umano percepisce naturalmente la variazione di intensità del suono secondo una scala logaritmica. Veniamo ora ai parametri con cui è possibile configurare il compressore [7]:

- `threshold_db`: definisce il livello di segnale in dB per cui il compressore si attiva e attenua il volume, al di sotto di tale livello lascia passare inalterato.

- `ratio`: questo parametro specifica il rapporto ingresso-uscita per segnali sopra la soglia, espresso in dB. Un ratio 1:1 rappresenta un guadagno unitario e quindi nessuna compressione, un ratio di 2:1 indica che un segnale che eccede la soglia di 2 dB viene abbassato a 1 dB sopra soglia e così via. Il caso estremo è un ratio di $\infty : 1$, ovvero qualsiasi porzione di segnale che eccede la soglia non viene lasciata passare in uscita, in quel caso l'effetto viene chiamato *limiter*.

- `attack_ms`: il parametro di attack, espresso in millisecondi, definisce il tempo impiegato dall'effetto per attivarsi una volta che il segnale ha superato la soglia (tipicamente intorno ai 5 ms).

- `release_ms`: dualmente all'attack, definisce il tempo impiegato all'effetto attivo per disinnescarsi e tornare a far passare inalterato il segnale (solitamente più lungo dell'attack, attorno ai 50 ms).

- `gain`: definisce il guadagno che viene applicato al segnale uscente dal compressore, per compensare l'attenuazione introdotta da quest'ultimo.

3.2.2 Clipper

Tra gli storici dispositivi analogici per l'elaborazione musicale spiccano sicuramente gli amplificatori a valvola e i distorsori, che hanno conferito alle chitarre elettriche il sound tipico della musica rock e di tutti i generi derivati che fanno un uso più o meno massiccio della distorsione. Questo sound è dovuto a varie non-linearità intrinseche in tutti i circuiti analogici: basta pensare alla curva caratteristica corrente-tensione *i-v* non lineare delle valvole e dei transistor usati negli stadi di amplificazione. Per ottenere simili effetti mediante elaborazione digitale è richiesto uno studio teorico

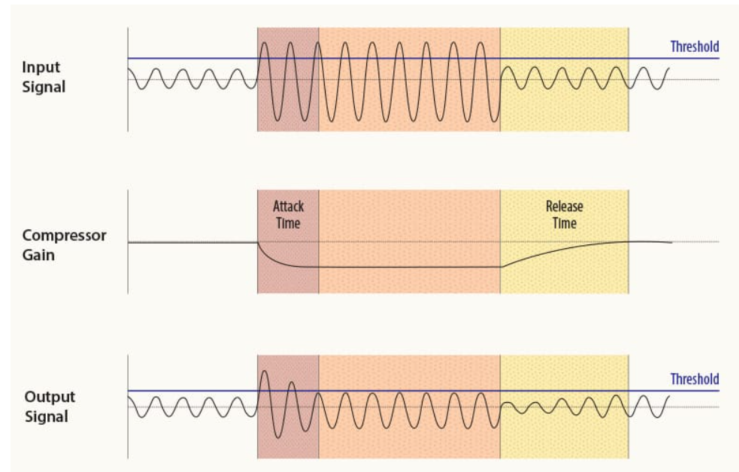

Figura 3.8: Rappresentazione grafica dell'effetto di attack e release

di questi sistemi non-lineari affinché possano essere emulati in maniera efficiente. A questo proposito sono stati proposti numerosi metodi matematici, come ad esempio l'applicazione di curve non-lineari statiche che mappano ogni valore di ampiezza del segnale di ingresso su un valore di ampiezza del segnale di uscita. Un esempio di curva potrebbe essere anche $y[n] = x[n]^2$, ma per l'implementazione dell'elemento `clipper`, che ha lo scopo proprio di ricostruire le non-linearità di un circuito analogico, gli sviluppatori hanno deciso di usare una funzione detta *smoothstep*. Questa funzione è definita come segue:

$$\text{smoothstep(x)}= \begin{cases} 0 & x < 0 \\ 3x^2 - 2x^3 & 0 \leq x \leq \text{Th} \\ 1 & x > \text{Th} \end{cases}$$

Dove Th rappresenta la soglia (*threshold*) oltre cui il segnale viene "clippato" (Figura 3.9).

Come già detto in precedenza l'applicazione di funzioni non-lineari genera nuove componenti spettrali che possono trovarsi a frequenze molto alte, tanto da eccedere il limite di Nyquist e creare aliasing nel segnale di output. Se si vuole evitare questo all'interno del file `clipper.c` sono state definite due fuzioni di `upsample` e `downsample`, così il segnale viene sovracampionato prima di applicare l'operazione non lineare per poi essere sottocampionato e riportato alla frequenza di campionamento di partenza. In particolare dopo il primo sovracampionamento si applica un filtraggio *anti-imaging* per eliminare repliche spettrali ad alta frequenza, dopo viene applicata la funzione non lineare e infine, prima del sottocampionamento, un filtraggio *anti-alasing*. Lo schema è mostrato in figura 3.10. I parametri del clipper sono i sequenti:

- `threshold`: stabilisce la soglia nella definizione della funzione non-lineare.
- `poly_clip`: il tipo di funzione polinomiale che si vuole usare (`smoothstep` o

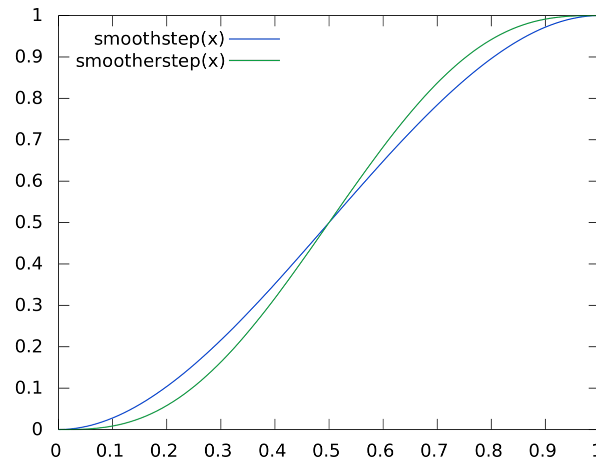

Figura 3.9: Funzione smoothstep nelle sue due versioni smooth e smoother

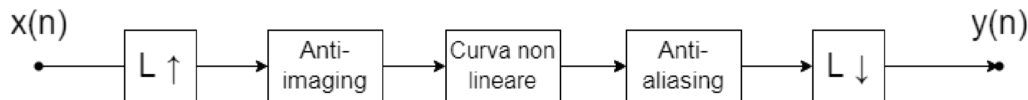

Figura 3.10: Schema di elaborazione non-lineare mediante sovracampionamento

`smootherstep`).

- `upsample`: questa variabile è un booleano, se impostato come vero il sistema applica l'upsampling come descritto precedentemente, altrimenti si limita ad applicare l'elaborazione con il rischio di aliasing (può essere voluto per ottenere distorsioni ancora più spinte).

3.2.3 Amplitude Modulation

Questo elemento implementa un modulatore di ampiezza. Una modulazione di ampiezza è un operazione atta ad andare a variare l'ampiezza di un segnale di ingresso $m[n]$, detto segnale modulante, in funzione dell'andamento di un secondo segnale $c[n]$, detto segnale di portante (*carrier*). Il segnale risultante viene chiamato segnale modulato e, nella versione più classica di modulazione di ampiezza, ha espressione del tipo:

$$x[n] = (1 + \alpha m[n]) \cdot c[n] \tag{3.9}$$

Considerando modulante e portante come sinusoidi a frequenze f_m e f_c, il segnale modulato sarà caratterizzato da tre componenti spettrali: la portante a frequenza $f_1 = f_c$ e due componenti a frequenze $f_2 = f_c - f_m$ e $f_3 = f_c + f_m$, ovvero la somma e la differenza delle frequenze di partenza, a questo risultato si può giungere applicando semplici formule trigonometriche. Questa operazione nasce nell'ambito delle telecomunicazioni per rendere più efficiente in termini energetici e di protezione dai disturbi la trasmissione di informazione. Esistono anche altri tipi di modulazione

in base al parametro che si va a modificare (modulazioni di frequenza e di fase). La modulazione può essere usata a scopi creativi quando applicata su segnali audio usando tipicamente come portante oscillatori a bassa frequenza (*LFO: Low Frequency Oscillator*) con $f \leq 20Hz$, l'effetto risultante avrà caratteristiche diverse in funzione della forma d'onda e della frequenza di portante. Al momento dell'istanza della struttura, i parametri che si possono impostare sono:

- `Depth`: entità della modulazione, corrisponde ad α nella formula sopra. Il suo valore è compreso tra 0 (effetto non attivato) e 1 (effetto massimo).
- `Rate`: frequenza della forma d'onda periodica usata come segnale di portante.
- `AMPLITUDE_MOD_TYPE`: tipo di forma d'onda, selezionabile dall'apposita enumerazione definita nel file header. Tra questi è presente anche il tipo `EXT_LFO`, da impostare per utilizzare una forma d'onda non presente tra quelle predefinite. In tal caso bisogna anche allocare un buffer `ext_mod` della lunghezza dell'audio block size, da passare come parametro alla funzione `amplitude_modulation_read` (se si usa una forma d'onda standard si passa un puntatore NULL).

Il seguente algoritmo produce il segnale modulato a partire dai parametri di cui sopra: il segnale di ingresso viene moltiplicato per il fattore di modulazione `trem_factor`

```
case AMP_MOD_SIN:
    for (int i = 0; i < audio_block_size; i++) {
        trem_factor = 1.0
                - (depth * (0.5 * oscillator_sine(t += inc) + 0.5));
        audio_out[i] = audio_in[i] * trem_factor;
    }
    break;
```

Figura 3.11: Algoritmo per la modulazione di ampiezza

che può assumere, in base a come definito, valori nel range $[0; 1]$. Quando il parametro depth è impostato a 0, l'effetto si traduce in una semplice attenuazione, essendo il fattore di modulazione costante e pari a 0.5, al contrario se depth vale 1 la variabilità del fattore è portata al massimo. La variabile `inc` rappresenta l'incremento temporale dell'oscillatore ed è definita in fase di setup come `mod_rate/audio_sample_rate`.

3.3 Simple Synth

Questo elemento, anche se presente nella libreria *Audio Elements*, non è considerabile una unità elementare alla stregua di quelle descritte in precedenza bensì un elemento più complesso che può essere composto sia di parti lineari che non. Un sintetizzatore è un apparato in grado di generare autonomamente segnali audio, sotto il controllo di un musicista o performer. Lo scopo può essere quello di ricreare il suono

di uno strumento reale o di creare suoni completamente nuovi, caratterizzanti della musica elettronica. Il synth con le sue sonorità innovative ha rivoluzionato la musica contemporanea, passando dall'essere uno strumento riservato allo sperimentalismo e all'avanguardia a diventare elemento chiave nei più celebri successi commerciali, basti pensare a un brano come *I Feel Love* di Donna Summer e Giorgio Moroder. Questi strumenti operano con circuiti analogici di svariati tipi e applicando svariate tecniche di sintesi, è da qui che nasce il particolare timbro che contraddistingue ogni synth. Con l'avvento del DSP sono state studiate soluzioni per emulare le tecniche di sintesi analogica mediante elaborazione numerica o anche tecniche del tutto nuove basate sull'approccio digitale. Ad oggi sono diffusissimi in ogni studio di produzione i cosiddetti *VST (Virtual Studio Technology)* ovvero delle versioni software di sintetizzatori utilizzabili all'interno del contesto più ampio dei programmi di elaborazione musicale al computer detti *DAW (Digital Audio Workstation)*. A questo scopo l'elemento `Simple Synth` nella libreria Audio Elements costituisce l'implementazione di un semplice sintetizzatore che sfrutta una forma d'onda di partenza e una tecnica di manipolazione dell'inviluppo detta *ADSR*. L'acronimo deriva dalle quattro parole *Attack, Decay, Sustain, Release*. Queste sono le fasi temporali successive in cui può essere suddivisa l'articolazione di un suono. [8] Di seguito una breve spiegazione di

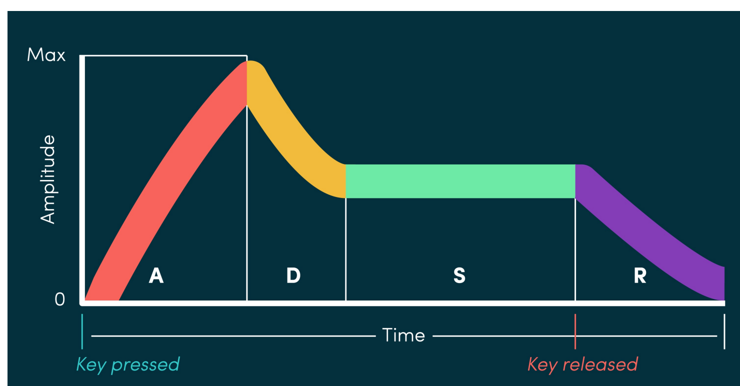

Figura 3.12: Le quattro componenti dell'ADSR in una forma d'onda

ognuno dei quattro parametri, alcuni dei quali ricorreranno anche in altri elementi o effetti.

- `attack`: tempo impiegato dal suono per arrivare al livello più alto (picco) partendo dal livello nullo in seguito riproduzione di una nota (ad esempio pressione di un tasto).
- `decay`: tempo impiegato dal suono per decadere dal picco iniziale al livello di sustain.
- `sustain`: tempo in cui il segnale mantiene il suo livello principale, finché il tasto resta premuto. Espresso talvolta in dB ma in questo caso in secondi.

- `release`: tempo necessario al segnale per andare dal livello di sustain fino a tornare allo 0 in seguito al rilascio del pulsante.

Per controllare l'inviluppo della forma d'onda è presente una variabile di tipo intero `position` che all'interno del loop di elaborazione viene incrementata insieme al campione che viene processato. Questa variabile serve da riferimento alla funzione `get_envelope` che in base a un confronto tra valori interi che corrispondono a campioni stabilisce in che fase dell'ADSR si trova l'onda e di conseguenza restituisce il giusto valore di inviluppo. Questi costituiscono quindi i parametri da passare alla funzione `synth_setup`, espressi in campioni. Gli altri parametri sono `synth_operator`, ossia il tipo di forma d'onda da utilizzare per la sintesi, e la frequenza di campionamento con cui si lavora. Ogni istanza dell'elemento Simple Synth può generare una specifica tonalità, quindi per sovrapporre più note bisogna istanziare un elemento per ogni tono. Inoltre la struttura è provvista di un flag `playing` da impostare a `true` quando si vuole che lo strumento suoni. Oltre alle solite funzioni di `setup` e `read` questo elemento contiene altre funzioni fondamentali:

- `synth_play_note`: questa funzione è quella che imposta a true il flag playing facendo si che lo strumento suoni la nota. Prende come parametri il volume della nota (normalizzato tra 0 e 1) e la nota in questione, indicata attraverso un numero intero secondo lo standard *MIDI (Musical Instrument Digital Interface)*. Questo standard assume come riferimento di partenza la nota A4 (La4) che corrisponde a una frequenza di 440 Hz. A partire da questa si ricavano le altre note in base alla relazione matematica che esiste tra loro:

  $$f = 440 \cdot 2^{(n-69)/12} \tag{3.10}$$

  Dove n è uno dei valori riservati alle note nel protocollo MIDI che vanno dal 21 (A0) al 108 (C8). Passata la nota la funzione si occupa poi di determinare la frequenza sfruttando un'altra funzione `note_to_increment`, che non fa altro che calcolare la formula riportata sopra. A partire da questa frequenza si trova poi l'incremento temporale della forma d'onda che viene assegnato alla variabile `t_inc` della struttura.

- `synth_play_note_freq`: questa funzione ha la stessa finalità di quella sopra, ma invece che prendere come parametro il valore secondo lo standard MIDI viene passata direttamente la frequenza in Hz della nota da suonare.

- `synth_update_note_freq`: anche questa funzione prende come parametro la frequenza della nota e aggiorna `t_inc` in modo che lo strumento cambi nota mentre continua a suonare.

- `synth_stop_note`: questa funzione fa si che lo strumento smetti di suonare: se il flag playing non è impostato true non fa niente, altrimenti se la variabile

position si trova già in fase di release lascia dissolvere la nota, in caso contrario salta alla fase di release.

A questo punto dovrebbe essere chiaro il funzionamento del loop di elaborazione principale `synth_read` riportato in figura 3.13.

```
case SYNTH_SINE:
    for (i = 0; i < audio_block_size; i++) {
        audio_out[i] = vol * get_envelope(c) * oscillator_sine(t);
        t += t_inc; // frequenza della nota
        if (t >= 1.0) // wrap di t
            t -= 1.0;
        c->position++; // scorre insieme al campione
    }
    break;
```

Figura 3.13: Funzione synth_read

Capitolo 4

Audio Effects

In questo capitolo saranno discussi e analizzati alcuni effetti musicali di uso comune tra quelli presenti nella libreria *Audio Effects*, creati a partire dagli elementi contenuti in *Audio Elements*.

4.1 Wah-wah e Auto-wah

Il primo effetto che si andrà a analizzare è il filtro *Wah-wah*, esso consiste in un filtro passa-banda con una frequenza di risonanza variabile nel tempo e una banda stretta. L'effetto è stato reso celebre dai grandi chitarristi del passato che lo controllavano tramite un pedale che, muovendosi avanti e indietro, pilota la frequenza di risonanza del filtro. Il segnale filtrato viene poi miscelato con il segnale d'ingresso come in figura. Se la variazione della frequenza centrale del filtro viene controllata

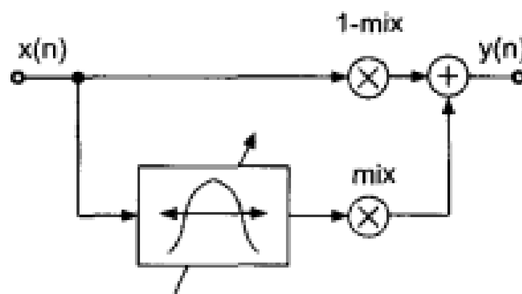

Figura 4.1: Schema effetto Wah-wah

da un parametro del segnale di ingresso si parla di *Auto-wah*, in particolare nel caso in questione verrà descritta l'implementazione di un Auto-wah dinamico pilotato dall'ampiezza del segnale di ingresso. Quando l'ampiezza cresce (viene suonata una nuova nota) la frequenza centrale viene spostata verso le alte frequenze, mentre con il decadere del suono essa torna alle basse frequenze. Questo effetto produce una sagomatura dello spettro simile a quella della voce umana, da questo deriva in maniera onomatopeica il suo nome. I parametri dell'effetto sono:

- `Depth`: regola quanto ampia sarà la transizione di banda del filtro in relazione all'ampiezza della nota suonata.
- `Q`: regola l'ampiezza della campana di risonanza del filtro
- `Decay`: impatta sul tempo impiegato dall'effetto per disinnescarsi. Se il valore è prossimo a 1 il decadimento sarà più lento e di conseguenza il filtro continuerà a muoversi più a lungo, al contrario un valore prossimo a 0 si traduce in un cessare dell'effetto più repentino.

Per l'implementazione dell'effetto verranno usati gli elementi `Biquad Filter` per il filtro passa-banda e, per misurare l'ampiezza, un rivelatore di picco definito nel file `Audio Utilities` contenuto in *Audio Elements*. Per il setup vengono istanziati tre filtri passa-banda che verranno posti in cascata per ottenere un filtro del sesto ordine e si inizializzano i tre parametri con il valore letto dai potenziometri attraverso l'uso delle variabili condivise. Questa è la funzione che esegue l'elaborazione (Figura 4.2): si può notare come il parametro decay è già incorporato nella funzione

```
void autowah_read(AUTOWAH * c, float * audio_in, float * audio_out,
        uint32_t audio_block_size) {

    // If this instance hasn't been properly initialized, pass audio through
    if (c == NULL || !c->initialized) {
        for (int i = 0; i < audio_block_size; i++) {
            audio_out[i] = audio_in[i];
        }
        return;
    }

    // Update amplitude
    for (int i = 0; i < audio_block_size; i++) {
        measure_amp_peak(audio_in[i], &c->measured_amplitude, c->decay);
    }

    float env_freq = c->measured_amplitude * c->depth;
    if (env_freq > AUTOWAH_MAX_BF_FREQ)
        env_freq = AUTOWAH_MAX_BF_FREQ;

    // Update filter center frequency based on amplitude
    filter_modify_freq(&c->bpf1, 300.0 + env_freq);
    filter_modify_freq(&c->bpf2, 300.0 + env_freq);
    filter_modify_freq(&c->bpf3, 300.0 + env_freq);

    // Apply band pass filters in series to create a 6th order filter
    filter_read(&c->bpf1, audio_in, audio_out, audio_block_size);
    filter_read(&c->bpf2, audio_out, audio_out, audio_block_size);
    filter_read(&c->bpf3, audio_out, audio_out, audio_block_size);
}
```

Figura 4.2: Funzione autowah_read

`measure_amp_peak`, che per ogni campione processato misura l'ampiezza e, se inferiore al picco precedentemente rivelato, la scala del valore di decay impostato (se a 1 la lascia invariata). La variabile `env_freq`, che rappresenta il "salto" di frequenza del filtro, è, come ci si aspettava, proporzionale all'ampiezza del campione, con

costante di proporzionalità definita proprio dal parametro `depth` (opportunatamente moltiplicato all'interno della funziona di modifica per un fattore 1000, tenendo conto che i valori di ampiezza misurati sono in virgola mobile). L'ultimo passaggio è quello dell'aggiornamento della frequenza centrale del filtro tramite la funzione `filter_modify_freq` e infine il il filtraggio del segnale con i tre filtri posti in cascata.

4.2 Tremolo

Il tremolo è un classico effetto in cui un segnale audio è sottoposto a una modulazione di ampiezza convenzionale con portante sinusoidale a frequenza molto bassa (LFO). A queste frequenze di portante le componenti spettrali che si generano sono molto vicine a quelle del segnale di partenza per cui l'effetto è udibile come una variazione dell'intensità della nota suonata. Diversamente se si usa una portante a frequenze più alte il segnale risultante viene percepito in maniera profondamente alterata rispetto all'originale, a causa delle componenti spettrali udibili distintamente. Per implementare questo effetto sarà sufficiente fare uso del modulatore di ampiezza, i parametri sono gli stessi del modulatore quindi `depth` e `rate` della sinusoide. La funzione `tremolo_read` qui non riportata è piuttosto semplice e si limita ad applicare a sua volta la funzione `amplitude_modulation_read`.

4.3 Ring Modulator

La *ring modulation* è una variante di modulazione di ampiezza che dato un segnale audio $x[n]$ e una portante sinusoidale $c[n]$ produce un segnale del tipo:

$$y[n] = x[n] \cdot c[n] \tag{4.1}$$

Lo spettro del segnale risultante è composto da due copie dello spettro del segnale originale: la banda laterale inferiore (LSB) e la banda laterale superiore (USB), dette *sidebands* (Figura 4.3). La LSB è speculare in frequenza allo spettro originale e entrambe le due repliche sono centrate attorno la frequenza di portante. Questo vale in caso di portante sinusoidale poiché per ogni armonica della portante si generano due sidebands e quindi due repliche. Se si usa una portante ad alto contenuto armonico (es. onda quadra) si genereranno molte più sidebands con probabilità di andare incontro ad una distorsione per via dell'aliasing. L'effetto acustico è facilmente comprensibile per segnali di ingresso semplici, ma con segnali più articolati si possono ottenere suoni molto particolari e d'avanguardia. I parametri come nella modulazione di ampiezza standard sono `depth` e `rate`.

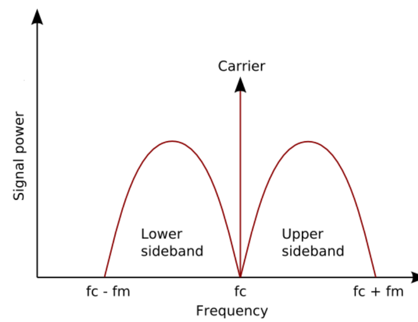

Figura 4.3: Bande laterali inferiore e superiore centrate attorno alla frequenza di portante f_c in seguito all'operazione di ring modulation

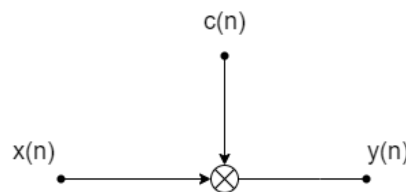

Figura 4.4: Schema effetto ring modulator

4.4 Stereo Flanger

Il flanger è un effetto che consiste sostanzialmente nell'applicazione di un delay tempo variante: la durata del delay (e quindi la lunghezza della delay line in campioni) non è fissa ma viene variata continuamente da un LFO. Per implementare questo effetto è stato usato l'elemento di base `variable_delay` che non è stato commentato tra gli Audio Elements perché è semplicemente una combinazione tra le operazioni di modulazione e di delay, già trattate in precedenza, che prende il nome di *delay line modulation*. Si noti che questa operazione può produrre valori di delay che sono multipli non-interi del periodo di campionamento, detti delay frazionari, con espressione del tipo:

$$y[n] = x[n - (M + frac)] \; con \; M \in \mathbb{N}, \; frac \in \mathbb{R} \; t.c. \; 0 < frac < 1 \tag{4.2}$$

ovvero un ritardo del segnale di ingresso di M campioni più una frazione di campione. Per calcolare questa uscita viene effettuata un'interpolazione (realizzabile tramite vari algoritmi, in questo caso gli sviluppatori hanno optato per la più semplice interpolazione lineare) che calcola il campione di uscita posto tra il campione M e il campione M+1 (Figura 4.5). L'effetto viene chiamato *stereo* flanger perché i due buffer mono di sinistra e di destra non sono ritardati allo stesso modo, infatti l'oscillatore di destra parte con uno sfasamento di 180 gradi rispetto quello di sinistra, per ottenere un effetto più immersivo. I parametri del flanger sono gli stessi dell'elemento variable delay, quindi `rate` e `depth` della modulazione e `feedback` del delay. I delay variabili vengono istanziati con il tipo di oscillatore `VARIABLE_DELAY_EXT_LFO` per poi andare

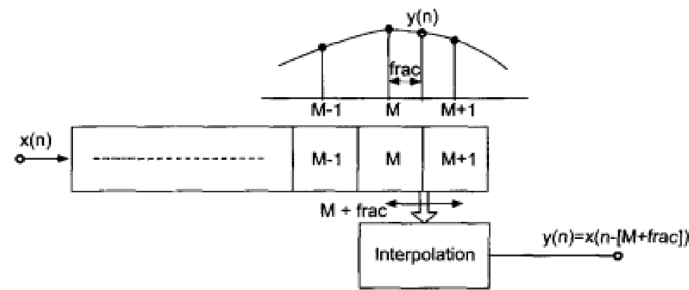

Figura 4.5: Delay frazionario mediante interpolazione

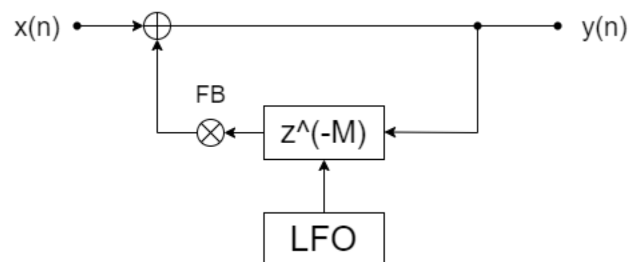

Figura 4.6: schema effetto flanger

a definire i due oscillatori all'interno della funzione di elaborazione, in questo modo si possono far partire gli oscillatori sfasati. Nel loop di elaborazione, dopo aver allocato i buffer di memoria per gli LFO, di lunghezza pari all'audio block size, vengono generati i due segnali, in questo caso sinusoidali. L'oscillatore di destra parte con un valore iniziale pari a 0.5 che corrisponde a uno sfasamento di 180 gradi se si ricorda che all'interno della funzione `oscillator_sine` i valori di t sono moltiplicati per 2π. Infine viene applicata due volte la funzione `variable_delay_read` usando i due

```
variable_delay_setup(&c->var_del_left, depth, feedback, rate_hz,
        audio_sample_rate, VARIABLE_DELAY_EXT_LFO);

variable_delay_setup(&c->var_del_right, depth, feedback, rate_hz,
        audio_sample_rate, VARIABLE_DELAY_EXT_LFO);

// Set up oscillators to be 180 degrees out of phase
c->lfo_t_left = 0.0;
c->lfo_t_right = 0.5;
c->inc = rate_hz / audio_sample_rate;
```

Figura 4.7: Setup degli oscillatori in quadratura di fase

LFO rispettivamente sui buffer di sinistra e di destra.

4.5 Audio Effects Selector

Questa funzione non implementa un effetto audio vero e proprio bensì è una funzione molto utile per organizzare i vari effetti che si vogliono caricare sulla scheda consentendo anche di passare da uno all'altro durante l'uso. Tutto questo evitando di dover scrivere una gran quantità di codice all'interno del file di `callback_audio_processing`, che risulterebbe illeggibile. All'interno del selettore, contenuto nella cartella `audio_processing`, sono previsti nove effetti che verranno caricati sul Core 1 e tra i quali è possibile scorrere avanti e indietro attraverso i push-button SW3 e SW4, sul Core 2 viene invece caricato un riverbero, un effetto basato sul delay, che potrà quindi essere sovrapposto all'effetto in uso sul Core 1 abilitandolo con il pulsante SW1. Per ogni effetto sono definite una funzione di `setup` e una di `process`. La prima effettua appunto il setup della struttura con le varie impostazioni iniziali ed eventuali altre operazioni preliminari come l'allocazione di memoria per una delay line. La seconda contiene la funzione che esegue l'elaborazione per quello specifico effetto più le funzioni che permettono la modifica dei parametri in tempo reale prendendo come valore di ingresso quello letto dal potenziometro. Tutte le funzioni di setup vengono

```
static void effect_autowah_process(void) {

    // Apply effect
    autowah_read(&autowah, audio_effects_left_in, audio_effects_left_out,
    AUDIO_BLOCK_SIZE);

    copy_buffer(audio_effects_left_out, audio_effects_right_out,
    AUDIO_BLOCK_SIZE);

    // Use pot (HADC0) to set the depth (i.e. frequency range of sweep)
    autowah_modify_depth(&autowah, multicore_data->audioproj_fin_pot_hadc0);

    // Use pot (HADC0) to set the decay time
    autowah_modify_decay(&autowah, multicore_data->audioproj_fin_pot_hadc1);

    // Use pot (HADC2) to set the width of the filter
    autowah_modify_q(&autowah, multicore_data->audioproj_fin_pot_hadc2);

}
```

Figura 4.8: Funzione process per l'effetto autowah

poi raggruppate all'interno di una funzione globale `audio_effects_setup_core1()` che andrà inserita all'interno della funzione `processaudio_setup()` contenuta in `callback_audio_processing`. Analogamente per le funzioni di elaborazione è definita una funzione `audio_effects_process_audio_core1()` che contiene uno *switch-case* controllato dalla variabile di tipo intero `effects_preset`, definita in `multicore_shared_memory` e sulla quale i pulsanti sono mappati per generare un incremento o decremento all'interno di `callback_Pushbuttons`. Tutto quanto discusso fin'ora costituisce una guida all'utilizzo della scheda e alle sue librerie, integrata con gli opportuni richiami teorici. Nel prossimo capitolo sarà presentata una modalità di utilizzo alternativa a quella standard, usata poi per implementare un algoritmo originale.

Capitolo 5

Sviluppo di algoritmi mediante linguaggio di alto livello Faust

In questo ultimo capitolo verrà introdotto il linguaggio di programmazione *Faust* e saranno presentate le sue principali caratteristiche. In seguito sarà illustrato in che modo il codice scritto può essere trasposto sulla *Sharc Audio Module* permettendo l'implementazione di algoritmi complessi in una maniera molto più semplice e intuitiva rispetto allo sviluppo standard in C++ discusso fin'ora.

5.1 Linguaggio Faust

5.1.1 Presentazione

Il linguaggio di programmazione *Faust (Functional Audio Stream)* [5] è un linguaggio di alto livello che nasce allo scopo specifico dello sviluppo di algoritmi di elaborazione e sintesi sonora. Il principale punto di forza di Faust è la sua portabilità, infatti il compilatore permette di esportare il codice Faust (indicato dall'estensione *.dsp*) su una grande varietà di sistemi operativi e piattaforme dedicate grazie alla possibilità di tradurlo nei linguaggi più usati (C,C++,Java,Javascript) per applicazioni audio in molti campi (dai plugin per DAWs ad app Android). Faust costituisce una valida alternativa ai linguaggi standard, con il vantaggio rispetto a quest'ultimi di una sintassi molto più semplice che astrae gran parte delle operazioni di basso livello permettendo quindi allo sviluppatore di concentrarsi sull'elaborazione del segnale vera e propria. Sarà poi il compilatore di Faust a optare per la soluzione migliore per tradurre il codice in altri linguaggi in maniera ottimizzata.

5.1.2 Caratteristiche principali

Per cominciare a programmare in Faust è sufficiente collegarsi all'IDE online al link `https://faustide.grame.fr/`. Questo ambiente di programmazione consente di testare il codice in tempo reale con tanto di ascolto e registrazione dell'output, possibilità di generare una GUI (Graphic User Interface) e di visionare grafici e schemi a blocchi. Di seguito viene riportato un esempio basilare.

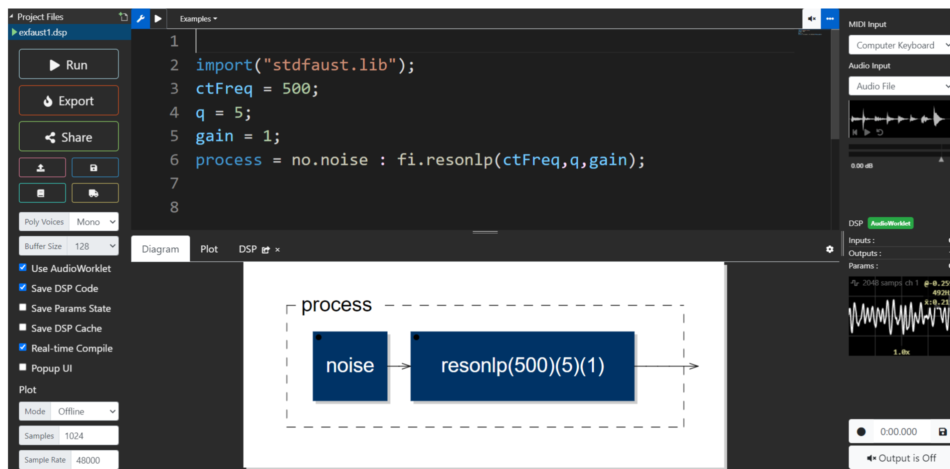

Figura 5.1: Esempio di codice FAUST: un generatore di rumore bianco posto in serie a un filtro passa-basso

La prima espressione chiave è `import("stdfaust.lib")`, si tratta dell'unico import di cui bisogna preoccuparsi quando si programma con Faust. La *Standard Faust Library* raggruppa al suo interno tutte le librerie di Faust, consultabili dalla documentazione online, che si suddividono tra loro in *environments*, ovvero macrocategorie di elementi di elaborazione audio preimplementati e messi a disposizione del programmatore. Nel caso in questione vengono richiamati l'elemento `noise` (generatore di rumore bianco) dalla `noise.lib`, indicata dal prefisso `no.`, e l'elemento `resonlp` (filtro passa basso risonante) dalla libreria `filters.lib` indicata dal prefisso `fi.`. Ogni elemento audio è una funzione e come tale può prevedere dei parametri, in questo caso l'elemento `fi.resonlp` prende come ingressi dentro parentesi tonde i parametri frequenza di taglio, fattore di risonanza e guadagno, definiti in precedenza. La parola chiave `process` rappresenta l'equivalente del main in altri linguaggi. Essa definisce il flusso di elaborazione audio principale (in maniera sequenziale da sinistra verso destra). In questo caso l'elaborazione consiste in un generatore di rumore collegato in serie ad un filtro passa basso, il collegamento in serie è espresso dall'operatore "`:`". Allo stesso modo è possibile collegare in parallelo più elementi tramite l'operatore "`,`" o effettuare lo *splitting* di un segnale su due canali con "`<:`" ed il *merging* di due canali in un unico ingresso con "`:>`". Un esempio di uso di questi operatori è riportato di seguito (Figura 5.2). Il rumore bianco viene splittato sul paralello di due filtri, passa-basso e passa-alto, per poter poi consentire un elaborazione sulle singole bande (ad esempio compressore multibanda). Quando si usano questi operatori bisogna assicurarsi di rispettare il numero di I/O previsti dall'elemento in questione, indicati sulla documentazione (ad esempio l'elemento `no.noise` ha 0 input e 1 output), altrimenti il compilatore segnalerà un errore. Ovviamente Faust supporta anche i classici operatori aritmetici tipici di ogni linguaggio di programmazione. Questi sono solo i più semplici dei molti operatori e delle funzioni che si possono usare, per una conoscenza più approfondita si consiglia di far riferimento al manuale completo

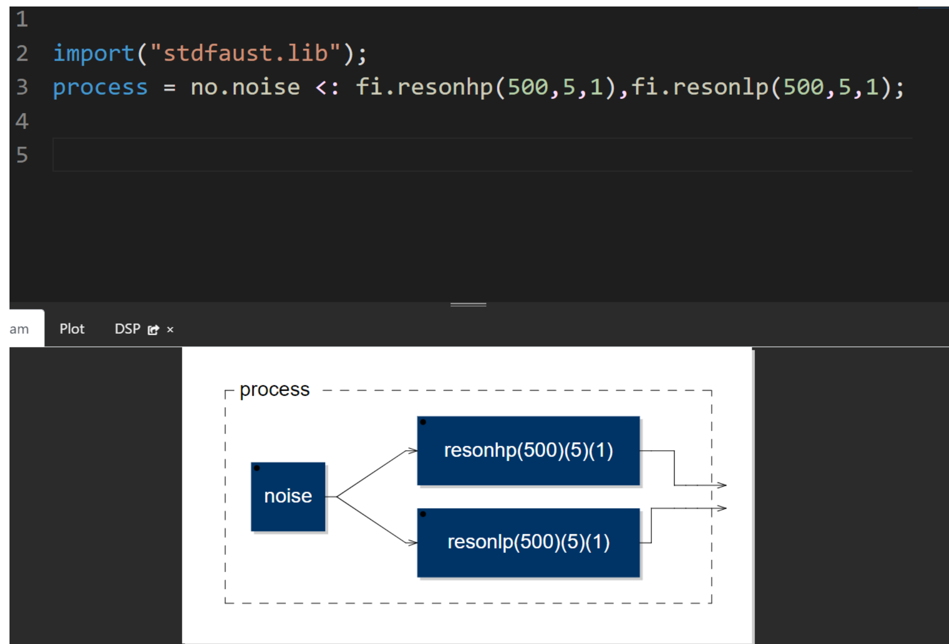

Figura 5.2: Esempio 2: generatore di rumore bianco splittato sul parallelo di due filtri (passa-basso e passa-alto)

disponibile sul sito ufficiale di Faust: `https://faustdoc.grame.fr/`. Gli esempi riportati dimostrano come con poche righe di codice è possibile implementare un algoritmo di elaborazione che su un altro linguaggio avrebbe richiesto molto più lavoro dovendo gestire direttamente aspetti come operazioni tra array, tipi di variabile ...

5.1.3 Graphic User Interface

Quando si vuole testare un algoritmo destinato all'utilizzo da parte di un performer è necessario introdurre degli elementi di controllo sui vari parametri per simulare una situazione di utilizzo di un effetto o di uno strumento digitale in tempo reale (come avveniva con l'Audio Project Fin). A questo scopo gli sviluppatori di Faust hanno messo a disposizione una serie di elementi GUI configurabili per controllare i parametri dell'algoritmo in maniera molto intuitiva. I più importanti tra questi sono:

- `hslider`,`vslider` : slider orizzontale o verticale che genera valori in maniera continua, compresi in un range che va da un minimo ad un massimo con un certo step intermedio.
- `button` : interruttore con due possibili stati (1/0) per abilitare o disabilitare un elemento.

Di seguito viene mostrato un esempio di utilizzo della GUI all'interno del primo algoritmo presentato. Come si può vedere in figura per utilizzare uno slider è sufficiente dichiarare la variabile da controllare con la sintassi:

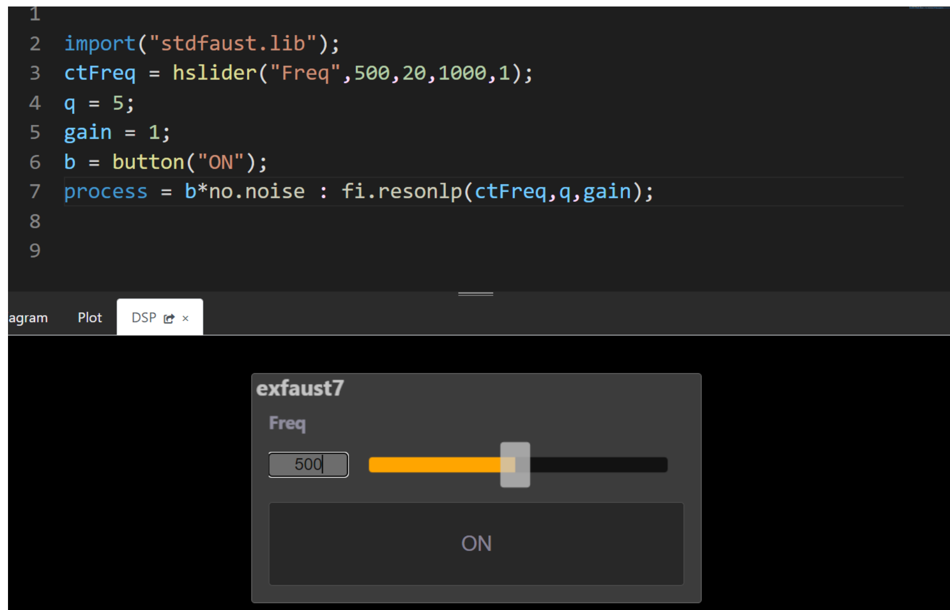

Figura 5.3: Stesso schema del primo esempio, con l'aggiunta di una GUI per il controllo della frequenza di taglio del filtro e dell'accensione del generatore.

```
var = hslider("Name",defaulValue,min,max,step)
```

per poi andare ad usare la variabile all'interno del processo, in questo caso come argomento di una funzione, come già fatto in precedenza. Per quanto riguarda l'elemento `button` esso viene dichiarato semplicemente con il suo nome, in seguito va a moltiplicare direttamente il parametro da controllare in maniera molto logica (quando vale 1 lascia passare il segnale inalterato, quando vale 0 lo annulla). In questo esempio infatti solo quando viene premuto il pulsante ON sarà possibile ascoltare il rumore filtrato alla frequenza indicata dallo slider.

5.1.4 Compatibilità tra Faust e SAM

Come è stato detto in precedenza il compilatore di Faust consente un ampia portabilità su diverse piattaforme, generando un codice specifico per l'architettura desiderata. Una delle piattaforme compatibili è ovviamente proprio la SHARC Audio Module. Per esportare codice destinato alla board bisogna cliccare sull'icona del camion che si trova a sinistra sotto al tasto `Run`, si aprirà un menù a tendina dove, tra le varie opzioni, va selezionata la voce `sam`. A questo punto il compilatore si avvale di un tool chiamato *faust2sam* per generare tre file *.cpp* disponibili al download. Questi tre file vanno inseriti all'interno della cartella `faust` situata nel percorso del progetto che è stato creato all'interno di CCES, avendo selezionato al momento della creazione la voce: "*Enable optional support for the Faust DSP environment*". In particolare se si sceglie di lavorare con Faust su un solo Core bisogna selezionare: "*I'll be running Faust on just Core 1*", altrimenti al momento del Build il compilatore darà errore perché si aspetta di trovare dei file `.cpp` generati con Faust anche all'interno della

cartella relativa al Core 2. Per quanto riguarda la trasposizione dei controlli definiti con la GUI sui pushbutton e i potenziometri disponibili sull'Audio Project Fin il compilatore si avvale del protocollo MIDI, supportato sia da Faust che dalla SAM. Come già detto in precedenza il protocollo associa a ogni intero un segnale di controllo, mentre un intervallo di numeri è riservato alle note musicali i restanti possono essere usati per mappare altri parametri dell'algoritmo a scelta del programmatore. In particolare per l'Audio Project Fin è stato stabilito convenzionalmente il mapping qui riportato:

Pot Mapping

Pot	CC
HADC0	CC-2
HADC1	CC-3
HADC2	CC-4

Push Button Switch Mapping

PB	CC
SW1	CC-102
SW2	CC-103
SW3	CC-104
SW4	CC-105

Figura 5.4: Mapping MIDI dei controlli dell'Audio Project Fin

Uno slider all'interno della GUI Faust si presta bene per sua natura a essere trasposto su di un potenziometro, per far si che un elemento di controllo Faust si metta in ascolto di un determinato segnale MIDI è sufficiente inserire nella definizione la seguente dicitura:

```
var = hslider("Freq [midi:ctrl 2]",defaultValue,min,max,step)
```

In questo modo il parametro `Freq` controllato dallo slider all'interno di Faust sarà automaticamente configurato per essere controllato dal potenziometro HADC0 al momento della compilazione. A questo punto sono state descritte tutte le funzionalità necessarie per procedere all'implementazione di un algoritmo complesso con controllo in tempo reale.

5.2 Implementazione di un sintetizzatore FM

La scelta effettuata è stata quella di utilizzare Faust per sviluppare un semplice sintetizzatore FM, la cui implementazione a partire dalle librerie fornite all'interno di CCES sarebbe altresì risultata abbastanza ostica.

5.2.1 Richiami teorici sulla sintesi FM

La sintesi FM [3] è una tecnica di design del suono basata sul metodo della modulazione di frequenza. Questa soluzione, difficilmente realizzabile con precisione con strumenti analogici, ha favorito il successo commerciale dei sintetizzatori digitali. In un'epoca in cui la memoria e la potenza di calcolo a disposizione erano piuttosto limitate, la sintesi FM ha consentito la creazione di suoni con un ricco contenuto armonico a discapito di una potenza di calcolo necessaria veramente modesta, considerando che bastano anche due oscillatori sinusoidali. Nell'ambito delle telecomunicazioni questa tecnica ha sopperito la modulazione di ampiezza per via della resistenza ai disturbi nettamente superiore. Il segnale risultante da una modulazione di frequenza, assumendo come segnale modulante una sinusoide del tipo $m(t) = sin(2\pi f_m t)$ e come portante sempre una sinusoide del tipo $c(t) = Asin(2\pi f_c t)$, assume la seguente forma:

$$x_{FM}(t) = Asin(2\pi f_c t + \beta sin(2\pi f_m t)) \tag{5.1}$$

I parametri della modulazione sono:

- f_m : frequenza di modulante;
- f_c : frequenza di portante;
- β : indice di modulazione, esprime l'entità della modulazione. Se $\beta = 0$ il segnale modulato coincide con la portante.

Questa operazione, similmente alla modulazione di ampiezza, ha la caratteristica di introdurre nuove armoniche nel segnale risultante, dovute alle cosiddette *sidebands*. Quando il segnale modulante è un LFO (oscillatore con f $\leq$ 20 Hz) il risultato della modulazione si traduce in un effetto detto *vibrato* applicato al segnale di portante, che resta comunque riconoscibile all'ascolto. L'idea alla base della sintesi FM è stata quella di utilizzare un segnale di modulante anch'esso con frequenza in banda audio, in questo modo le armoniche che si vengono a creare in seguito alla modulazione sono abbastanza distanti tra loro da far sì che il segnale risultante venga avvertito come profondamente diverso da quello di partenza. [9] Le caratteristiche del tipo di suono che si ottiene dipendono dal *ratio* della modulazione, ovvero dal rapporto f_c/f_m. Se questo rapporto è intero le nuove armoniche sono situate a frequenze multiple intere della fondamentale, risultando in un suono armonico. Al contrario se il rapporto non è intero il suono risultante sarà fortemente inarmonico all'ascolto. Per questo motivo nell'implementazione proposta piuttosto che andare a variare le due frequenze indipendentemente si andrà a considerare il `rate` di una in funzione dell'altra. La sintesi FM permette di ottenere sonorità nuove, ma se questo suono si ripete invariato nel tempo senza alterazioni risulterà noioso all'ascolto dopo poco tempo. Per questo motivo ad ognuno degli oscillatori coinvolti nella modulazione si associa un generatore di inviluppo per il controllo del volume di uscita, come ad esempio un ADSR, di cui si è già parlato in precedenza. L'insieme di oscillatore e

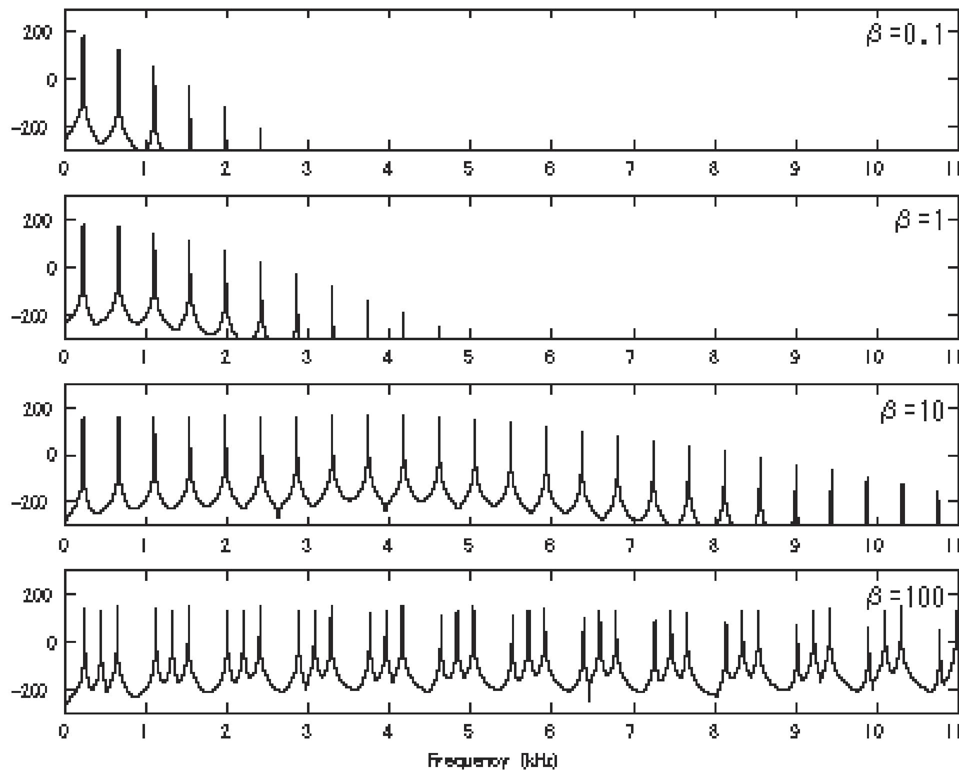

Figura 5.5: Spettro del segnale modulato all'aumentare di β, con f_c=220 Hz e f_m=440 Hz

generatore di inviluppo nella terminologia della sintesi FM prende il nome di *operator*. Un sintetizzatore FM professionale è solitamente provvisto di un certo numero di operatori e prevede diverse possibili disposizioni per questi ultimi che determinano ad esempio quale operatore modula quale o quanti operatori modulano lo stesso operatore. Queste disposizioni sono chiamate algoritmi e costituiscono un ulteriore punto di forza della sintesi FM per la varietà di timbriche ottenibili che aumenta esponenzialmente con il numero di operatori a disposizione e, quindi, di algoritmi.

5.2.2 Realizzazione mediante Faust

Il sintetizzatore FM realizzato tramite Faust prevede due operatori e un solo algoritmo disponibile. Questa scelta è stata effettuata compatibilmente al fatto che questo synth è destinato ad essere utilizzato sulla SAM che, sul modulo Audio Project Fin, dispone di soli tre controlli continui per i parametri degli operatori. Di conseguenza non avrebbe senso introdurre altri operatori per i quali non si hanno a disposizione abbastanza potenziometri su cui mappare i controlli. Il codice può però sempre essere preso come punto di partenza per sviluppare un sintetizzatore più complesso che preveda più operatori. All'interno del sintetizzatore i due operatori coinvolti sono costituiti da un oscillatore sinusoidale (`os.osc`) e e da un generatore di inviluppo (`en.ar`) di tipo AR (Attack Release). Il secondo oscillatore modula

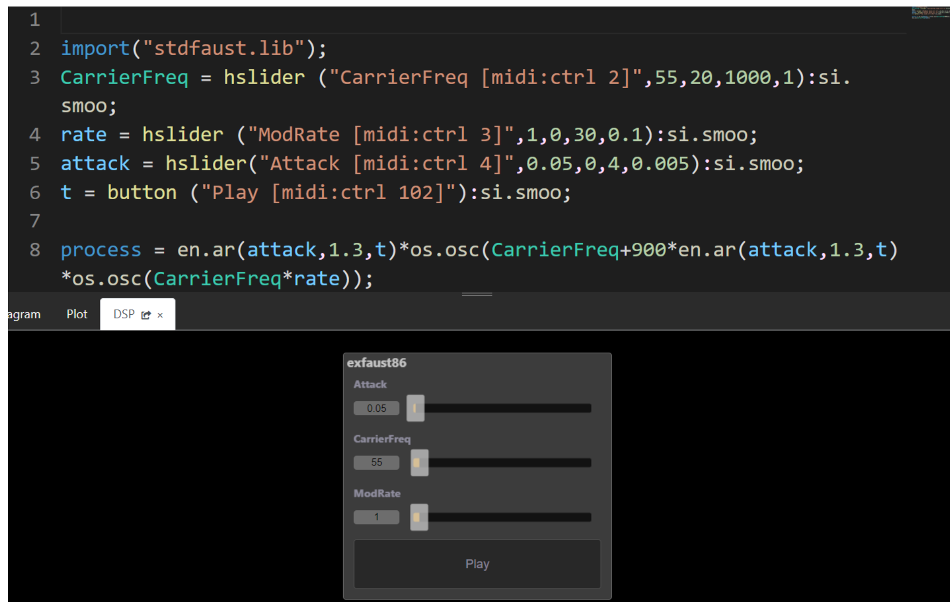

Figura 5.6: Sintetizzatore FM realizzato mediante FAUST

la frequenza del primo, con indice di modulazione fisso e pari a 900, questo valore determina una presenza pronunciata dell'effetto della modulazione. I parametri di controllo, mappati sui tre potenziometri della board tramite i controlli MIDI, sono:

- `CarrierFreq` : frequenza di portante che determina la tonalità dell'oscillatore principale (quello udibile) impostata di default alla frequenza di 55 Hz, che corrisponde alla nota A1 (La1).
- `rate` : rapporto tra frequenza di portante e frequenza di modulante, impostato di default a 1.
- `attack` : tempo di attacco, il medesimo per i due operatori. Si è scelto di poter variare l'attacco mentre il tempo di rilascio è stato fissato a 1.3 s.

Il pulsante `Play`, mappato sul pushbutton SW1, permette di suonare una nota sul synth. In questo caso il pulsante viene passato come parametro alla funzione `en.ar` che prevede come ingressi il tempo di attacco, quello di rilascio e il pulsante che dà il comando di suonare la nota. Infine la funzione `si.smoo` applicata agli slider è uno *smoother*, ovvero un elemento che "ammorbidisce" le transizioni brusche dello slider al fine di evitare la produzione di un suono di click. I valori dei parametri degli operatori visibili in figura sono stati individuati con un procedimento a tentativi, con lo scopo di ottenere un sound riconducibile a quello di un tipico *synth bass*. Con le impostazioni di default viene infatti riprodotta una nota a bassa frequenza con un attacco molto rapido e un rilascio più lungo. Queste considerazioni sono confermate dall'analisi dello spettro e dello spettrogramma del suono riprodotto di seguito riportate.

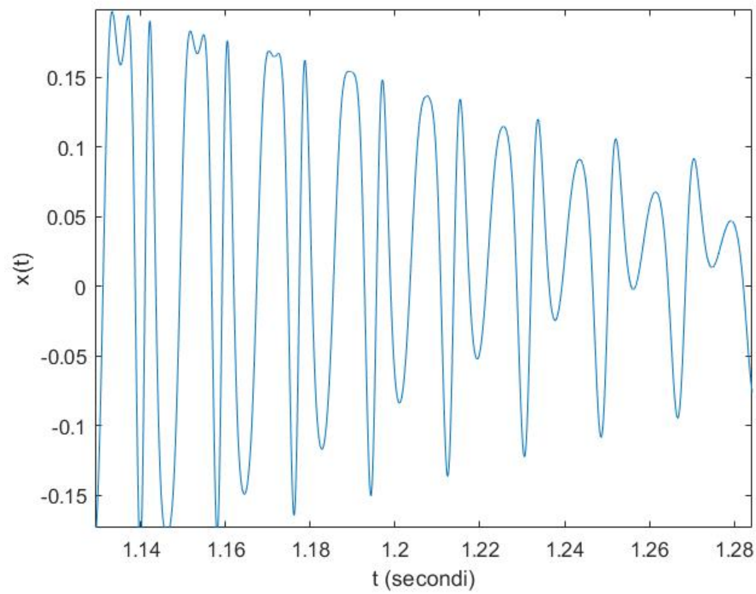

Figura 5.7: Dettaglio della forma d'onda di uscita nella fase di rilascio della nota, realizzato tramite MATLAB

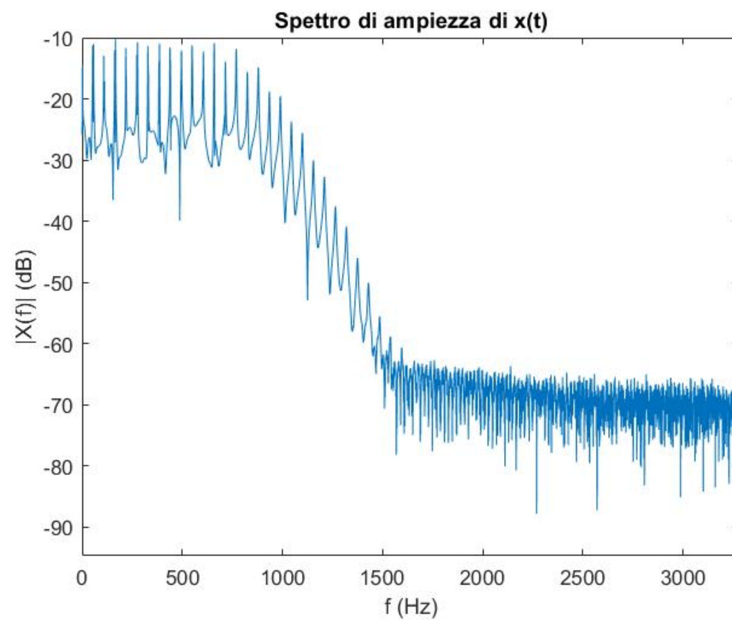

Figura 5.8: Spettro di ampiezza della nota riprodotta, realizzato tramite MATLAB

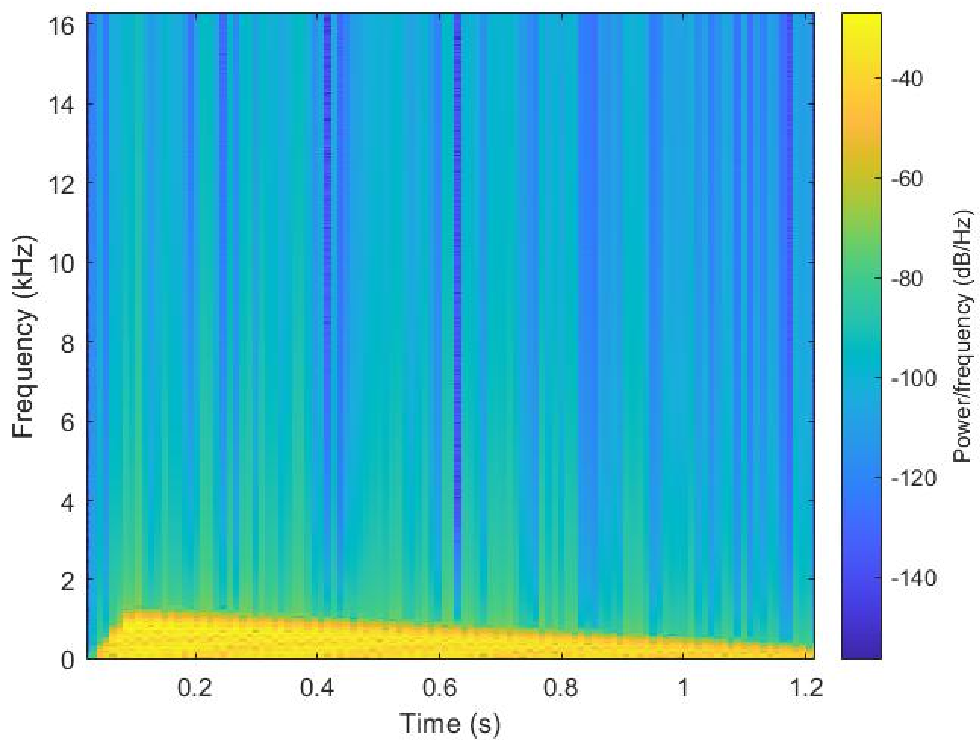

Figura 5.9: Spettrogramma della nota riprodotta, realizzato tramite MATLAB

Capitolo 6

Conclusioni

In conclusione al lavoro svolto si può affermare che la *Sharc Audio Module* si dimostra una piattaforma molto potente e valida per le applicazioni di elaborazione del segnale musicale in tempo reale. Non sono stati riscontrati problemi nell'effettuare il debugging e gli effetti forniti dagli sviluppatori sono stati testati e validati con l'ausilio di strumentazione elettronica esterna come sorgente audio. Per quanto

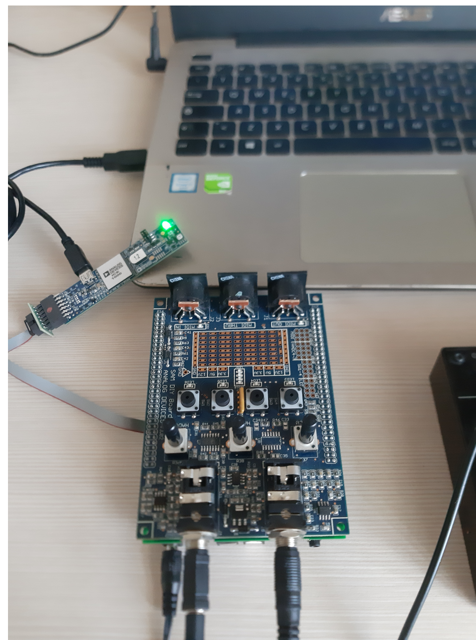

Figura 6.1: Scheda correttamente connessa al computer, stato LED emulatore: verde, pronto per il debugging.

riguarda le librerie di codice *Audio Elements* e *Audio Effects*, esse forniscono un ottimo punto di partenza per chiunque voglia approcciarsi alla programmazione DSP su piattaforma dedicata, permettendo di comprendere quali sono le basi teoriche dell'elaborazione e fornendo una soluzione software adatta a farsi un'idea di come strutturare un progetto del genere. Ciò nonostante gli algoritmi proposti al loro interno non possono essere considerati implementazioni allo stato dell'arte, infatti il codice non è ottimizzato e non potrebbe essere utilizzato in un contesto di mercato in cui bisogna sfruttare parsimoniosamente ogni singola risorsa in termini di memoria

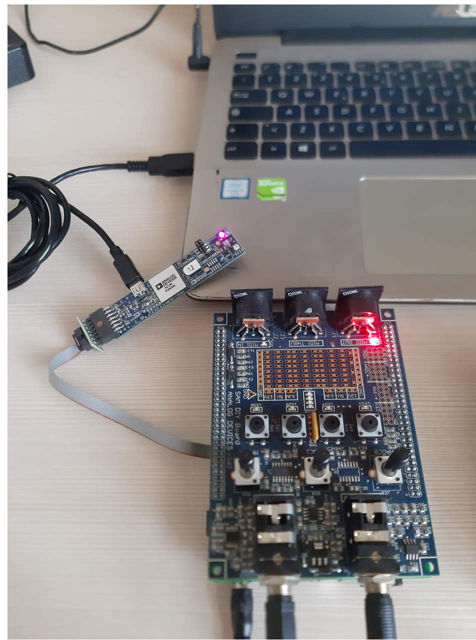

Figura 6.2: Codice caricato correttamente, stato LED emulatore: viola, in fase di debugging.

e CPU per rendere il prodotto finale più economico e quindi più competitivo. Questo fattore comunque non costituisce un problema all'uso della scheda che dispone di un'abbondante potenza di calcolo, le suddette librerie rimangono quindi un importante strumento utile a fini didattici e dimostrativi. Un successivo lavoro potrebbe essere quello di generare codice ottimizzato da far girare sulla scheda e confrontare le prestazioni rispetto ai codici studiati in questo elaborato. Infine l'integrazione del linguaggio Faust, non prevista originariamente nel lavoro assegnato, si è rivelata una piacevole scoperta poiché questo strumento software rappresenta una soluzione molto immediata e concreta che consente di realizzare algoritmi complessi anche a chi non è familiare con la programmazione a basso livello su sistemi embedded. Ricollegandosi anche al discorso dell'ottimizzazione, il compilatore *faust2sam* permette di generare codice abbastanza performante senza il bisogno di essere esperti di complessità algoritmica in C/C++. L'unica nota di demerito per quanto riguarda Faust consiste nel fatto che il codice *.cpp* generato con il compilatore online non risulta compatibile con Cross Core Embedded Studio in quanto utilizza l'ultima versione di C++ che non è ancora supportata dal software in questione. Comunicando con il team di assistenza della Analog Devices si è convenuti che la soluzione a questo problema è quella di esportare il codice utilizzando una vecchia versione di Faust, questo purtroppo non è consentito dall'editor online ma è possibile farlo usando *faust2sam* da command line, supportato però solo da sistemi Mac e Linux. In futuro è comunque probabile che questo problema verrà risolto. Il codice Faust presentato è riportato in [10], insieme a un video dimostrativo sull'utilizzo della scheda.

Bibliografia

[1] Oppenheim A. Schafer R. *Discrete-Time Signal Processing, Third edition.* Pearson, 2014.

[2] Carraturo A. Trentini A. *Sistemi embedded: teoria e pratica.* Ledizioni, 2017.

[3] John M Chowning. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the audio engineering society*, 21(7):526–534, 1973.

[4] Analog Devices. SHARC Audio Module. `https://wiki.analog.com/resources/tools-software/sharc-audio-module`.

[5] Grame-CNCM. Faust Programming Language. `https://faust.grame.fr/`.

[6] Zolzer U. *Dafx: Digital Audio Effects.* Wiley, 2002.

[7] Hicks M. Audio compression basics. `https://www.uaudio.com/blog/audio-compression-basics/`.

[8] Onda N. Gli inviluppi ADSR: come costruire il suono perfetto. `https://blog.landr.com/it/inviluppi-adsr/`.

[9] Cosimi E. *Analog e virtual analog. Come funziona un sintetizzatore.* Curci, 2018.

[10] Aloisi A. Faust repository. `https://github.com/Amerigo25/Faust`.