



UNIVERSITÀ
POLITECNICA
DELLE MARCHE

FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

Sviluppo tramite FAUST di un algoritmo multieffetto per chitarra ed implementazione su DSP Sharc

**FAUST development of multi-effects algorithm for guitar and
implementation on DSP Sharc**

Candidato:
Dennis Rapaccini

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

Sviluppo tramite FAUST di un algoritmo multieffetto per chitarra ed implementazione su DSP Sharc

**FAUST development of multi-effects algorithm for guitar and
implementation on DSP Sharc**

Candidato:
Dennis Rapaccini

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 Brezze Bianche – 60131 Ancona (AN), Italy

Ai resilienti, ispirazione di vita.

Ringraziamenti

In primis desidero ringraziare il prof. Stefano Squartini e il prof. Leonardo Gabrielli per la disponibilità dimostrata in questo periodo e per avermi dato l'opportunità di scrivere questa tesi in un ambito che, sin da piccolo, mi affascina.

Grazie agli amici, ai colleghi e alla famiglia, per avermi incoraggiato fin dall'inizio di questo percorso.

Infine vorrei ringraziare il mio lato istintivo e irrazionale che, seppur marginalmente, mi distrae dall'ordinaria linearità e mi permette di percorrere sempre la giusta via.

Ancona, Dicembre 2022

Dennis Rapaccini

Sommario

Il linguaggio di programmazione Faust, data la sua versatilità, ben si presta ad essere utilizzato per applicazioni di effettistica musicale. Il presente elaborato, difatti, ha il fine di sviluppare un algoritmo multieffetto per chitarra tramite Faust e di effettuarne il porting su SHARC Audio Module. In tal senso, una volta introdotte le piattaforme utilizzate, vengono trattati gli aspetti di design teorico e di implementazione in codice degli effetti sviluppati. Data la specificità dell'hardware utilizzato, un particolare focus viene seguentemente dedicato all'interfaccia di controllo del multieffetto. Infine, si dettaglia la procedura di porting su board evidenziandone gli aspetti di compatibilità, i problemi riscontrati e le possibili soluzioni.

Indice

Introduzione	1
Pedale multieffetto	2
1 Hardware e Software	3
1.1 SHARC Audio Module	3
1.1.1 Audio Project Fin	4
1.1.2 ICE-1000	4
1.1.3 Cross Core Embedded Studio	4
1.2 Faust	5
1.2.1 Paradigma	5
1.2.2 Sintassi	6
1.2.3 Ambiente di sviluppo	8
1.2.4 Tools faust2xx	9
2 Design e implementazione degli effetti	11
2.1 Design degli effetti	11
2.1.1 Distorsione	11
2.1.2 Chorus	14
2.1.3 Riverbero	17
2.2 Implementazione in Faust	19
2.2.1 Distorsione	20
2.2.2 Chorus	21
2.2.3 Riverbero	22
2.2.4 Volume	23
2.3 Interfaccia di controllo	24
2.3.1 Gestione hardware dei parametri del multieffetto	25
2.3.2 Gestione software dei parametri del multieffetto	26
2.3.3 Gestione grafica dei parametri del multieffetto	27
2.4 Codice completo	28
3 Porting su SHARC Audio Module	31
3.1 Compatibilità	31
3.1.1 Installazione di Faust su macchina Linux	31

Indice

3.1.2	Compilazione/conversione dell'algoritmo	33
3.2	Building e booting	34
3.2.1	Operazioni preliminari	34
3.2.2	Building	35
3.2.3	Debugging	36
3.2.4	Bootig	36
	Conclusioni	37

Elenco delle figure

0.1	Pedale multieffetto Neural DSP Quad Cortex	1
0.2	Schema a blocchi di un generico multieffetto	2
1.1	SHARC Audio Module	3
1.2	SHARC Audio Module Audio Project Fin	4
1.3	ICE-1000	4
1.4	Operatore ‘:’	6
1.5	Operatore ‘,’	6
1.6	Operatore ‘<:’	7
1.7	Operatore ‘:>’	7
1.8	Operatore ‘~’	7
1.9	Operatore ‘@’	7
1.10	Funzione <i>select2</i>	7
1.11	Esempi di sliders	8
1.12	Interfaccia della Faust IDE	9
2.1	Andamento della funzione distorsione	12
2.2	Andamento della funzione distorsione per diversi valori di α	13
2.3	Risposta nel tempo del distorsore ad un ingresso sinusoidale	13
2.4	Schema a blocchi di un generico chorus	15
2.5	Schema a blocchi di una delay line interpolata linearmente	17
2.6	Raffigurazione di un ambiente in presenza di riverbero	17
2.7	Schema di un riverberatore piano	18
2.8	Schema a blocchi del "Freeverb"	19
2.9	Implementazione Faust della distorsione mediante schemi a blocchi	20
2.10	Implementazione Faust del chorus mediante schema a blocchi	22
2.11	Implementazione Faust del riverbero mediante schemi a blocchi	23
2.12	Implementazione Faust del controllo volume	23
2.13	Suddivisione degli elementi costituenti la Audio Project Fin	24
2.14	Esempio di hslider utilizzato come pulsante	25
2.15	Esempio di sample and hold	27
2.16	Faust GUI del multieffetto	28

Elenco delle figure

2.17 Codice Faust del multieffetto 30

Elenco delle tabelle

2.1	Mapping tra dispositivi fisici e controller MIDI	24
-----	--	----

Introduzione

Sin dai primi anni del XVI secolo, la chitarra ha subito una continua evoluzione di forma e sonorità. Con la crescente diffusione nell'era contemporanea della chitarra acustica e con la necessità di ottenere volumi sempre maggiori, nel 1931 Adolph Rickenbacker realizza il primo pick-up elettromagnetico, dando così vita al primo prototipo di chitarra elettrica [1] [2].

A partire dagli anni '50, con l'avvento del rock 'n roll, si è sempre più alla ricerca di nuove sonorità : viene ideato il primo pedale per chitarra.

Il primo effetto a pedale a transistor, il Maestro Fuzz Tone, nasce nel 1962 e da lì numerosi effetti come chorus, wah-wah e phaser vengono commercializzati.

I primi effetti digitali non vedranno luce fino agli anni '80, con l'ampia diffusione dei primi microcontrollori [3]. L'avvento del digitale ha consentito di implementare effetti a pedale in modo più flessibile, economico e con dimensioni ridotte. La maggior flessibilità ha portato a sviluppare, con grande successo commerciale, pedali all-in-one che includessero numerosi effetti e amps emulation, ossia modellizzazioni digitali di amplificatori storici a valvole termoioniche. Esempi famosi sono Line 6 Helix e Neural DSP Quad Cortex.



Figura 0.1: Pedale multieffetto Neural DSP Quad Cortex

Dato il sempre crescente interesse per l'elaborazione digitale dei segnali, in particolar modo quelli audio, il mercato propone diverse soluzioni software e hardware a scopo prototipale e didattico per far avvicinare studenti

ed appassionati all'argomento. Esempi noti sono la TSA1701 Audio DSP Board e la SHARC Audio Module per l'hardware e Faust come linguaggio di programmazione. Proprio su questi ultimi due sarà incentrato il presente elaborato.

Pedale multieffetto

In linea di principio, un pedale (o *stompbox* in inglese) si pone come step intermedio tra il segnale in uscita dalla chitarra elettrica e l'ingresso dell'amplificatore. Il pedale modifica, in numerosi possibili modi, il segnale entrante nell'amplificatore. Nel caso digitale, l'elaborazione del segnale viene effettuata da un processore (*Digital Signal Processor*) a seguito di una conversione analogico-digitale. Dopo l'elaborazione, una conversione digitale-analogico consente di inviare il segnale all'ingresso dell'amplificatore.

Nel caso di un pedale multieffetto, più effetti sono posizionati in serie o in parallelo, formando una cosiddetta *catena*.

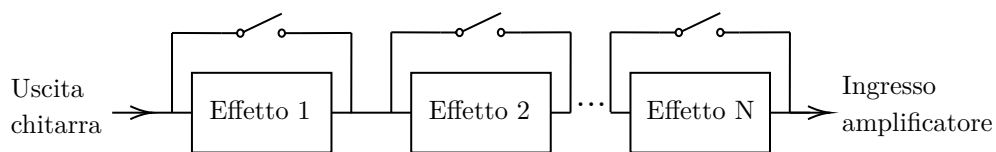


Figura 0.2: Schema a blocchi di un generico multieffetto

Capitolo 1

Hardware e Software

Questo Capitolo ha l'intenzione di presentare, ai fini della comprensione dell'elaborato, le piattaforme hardware e software utilizzate.

1.1 SHARC Audio Module

La *SHARC Audio Module* [4] della Analog Devices è una piattaforma hardware/software espandibile che consente il prototyping, lo sviluppo e l'implementazione di applicazioni audio come il processing di effetti, sistemi audio multi-canale e sintetizzatori MIDI.

L'elemento centrale di essa è il processore ADSP-SC589. Questo contiene un ARM Cortex-A5 e due cores SHARC+ floating point da 450 MHz.

In aggiunta, sulla board sono presenti due memorie RAM DDR3 da 2 GB, una memoria flash SPI da 512Mb, un modulo Ethernet ed un vasto numero di pin I/O.

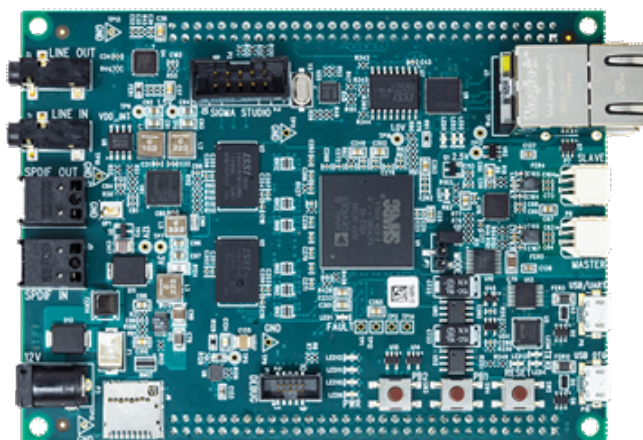


Figura 1.1: SHARC Audio Module

1.1.1 Audio Project Fin

La Analog Devices mette a disposizione una espansione, la *SHARC Audio Module Audio Project Fin* [5]. Questa può essere connessa semplicemente dai pin I/O e fornisce un ottimo supporto per il testing in real-time degli algoritmi implementati sulla board principale.

L'espansione fornisce due jacks stereo da 1/4" (6.35 mm) (IN e OUT), tre porte MIDI (IN, OUT e THROUGH), 4 pulsanti, 3 potenziometri e 4 indicatori LED.

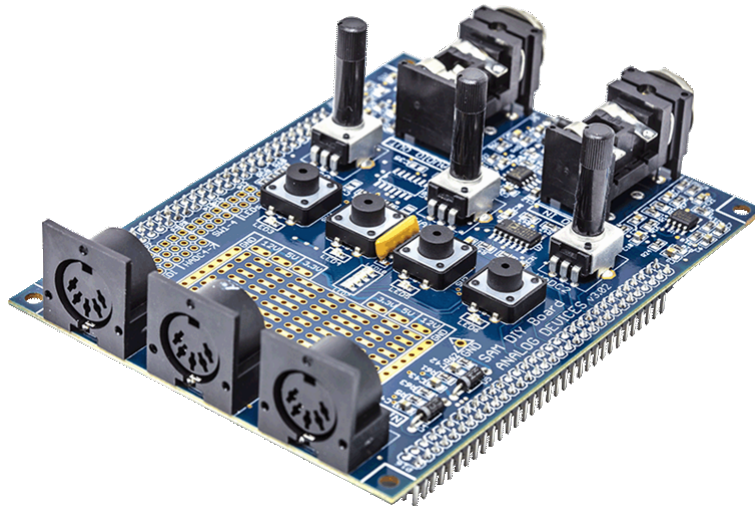


Figura 1.2: SHARC Audio Module Audio Project Fin

1.1.2 ICE-1000

Per poter programmare e debuggare la board è necessario disporre dell'emulatore *ICE-1000* [6]. Essendo un'interfaccia JTAG (Joint Test Action Group), permette di poter comunicare correttamente con l'ambiente di sviluppo .



Figura 1.3: ICE-1000

1.1.3 Cross Core Embedded Studio

L'ambiente di sviluppo *CrossCore Embedded Studio* (CCES), di proprietà della Analog Devices, è un IDE basato su Eclipse che permette di sviluppare, debuggare e implementare algoritmi scritti in C/C++ all'interno di processori

SHARC. Per potersi interfacciare con il processore precedentemente illustrato, occorre avvalersi di un supporto software aggiuntivo chiamato *Bare Metal framework*. Questo framework fornisce in fase progettuale modelli, librerie e inizializzazioni standard, permettendo di poter configurare in modo semplice e intuitivo i vari parametri necessari per il corretto funzionamento dell'algoritmo da implementare sui cores.

1.2 Faust

Faust (Functional Audio Stream) [7] è un linguaggio di programmazione funzionale sviluppato dal dipartimento di ricerca del GRAME-CNCM (Générateur de Ressources et d'Activités Musicales Exploratoires - Centre National de Création Musicale) di Lione, in Francia. Tale linguaggio viene ideato per il sound synthesis e l'audio processing con un particolare focus alla progettazione di sintetizzatori, strumenti musicali ed effetti audio.

Il componente fondamentale di Faust è il compilatore, che permette di "tradurre" ogni specifica scritta in Faust in un'ampia gamma di linguaggi di dominio non specifico come C, C++, LLVM bitcode, WebAssembly, Rust e altri .

La forza di questo linguaggio è sicuramente il vasto numero di librerie standard presenti che permettono di poter implementare, in modo semplice, funzioni DSP avanzate di audio processing e di sintesi.

1.2.1 Paradigma

La programmazione in Faust combina due approcci: la *programmazione funzionale* e l'*algebra degli schemi a blocchi*.

Programmazione funzionale

Si basa sulla valutazione di funzioni matematiche. Tutti gli elementi possono essere intesi come funzioni e il codice può essere eseguito tramite richieste di funzioni sequenziali. Questo paradigma ha il focus sul cosa risolvere (piuttosto che sul come risolverlo) e per questo si contrappone allo stile di programmazione imperativo, come ad esempio la programmazione orientata agli oggetti.

Una caratteristica della programmazione funzionale sono gli oggetti immutabili. Una volta definito il valore di un oggetto immutabile, questo non può più essere modificato: vengono infatti create e modificate copie dell'oggetto.

Algebra degli schemi a blocchi

Il formalismo basato sui diagrammi a blocchi viene largamente utilizzato nei cosiddetti VPL (Visual Programming Language), in particolar modo nei linguaggi per applicazioni musicali. Il programma viene scritto connettendo tra loro blocchi grafici rappresentanti le funzionalità del sistema. Ogni blocco viene rappresentato internamente da un grafo e interpretato come un dataflow. Approcci di questo tipo comportano intrinsecamente diversi svantaggi. Per esempio a causa della complessità semantica dei modelli a dataflow, la maggior parte dei linguaggi di programmazione siffatti non ha una semantica formale esplicita, risultando così complicato capire il reale comportamento del diagramma a blocchi. Spesso i blocchi costitutivi sono rappresentati mediante grafi che sono in genere complessi da manipolare.

Per far fronte a questi problemi Faust utilizza, piuttosto che una rappresentazione a grafi, una *rappresentazione ad albero* basata su *operatori di composizione* (*block-diagram algebra*) [8].

1.2.2 Sintassi

Verranno ora introdotti alcuni operatori e alcune funzioni di pratico interesse presenti in Faust.

Sequential composition ($A : B$) : questo operatore viene utilizzato per collegare le uscite di un generico blocco A agli ingressi di un generico blocco B .

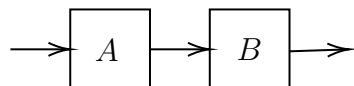


Figura 1.4: Operatore ‘:’

Parallel composition (A, B) : questo operatore posiziona il blocco A sopra il blocco B , senza connessioni.

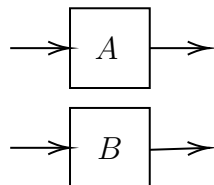


Figura 1.5: Operatore ‘,’

Split composition ($A <: B$) : questo operatore distribuisce le uscite del blocco A agli ingressi del blocco B .

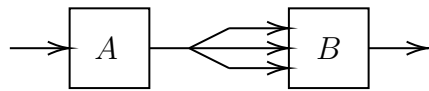


Figura 1.6: Operatore '<:'

Merge composition ($A :> B$) : questo operatore è il duale dello split composition. Esso infatti unisce le uscite del blocco A al numero di ingressi del blocco B .

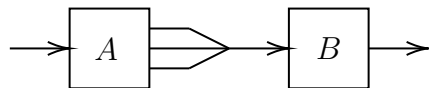


Figura 1.7: Operatore ':>'

Recursive composition ($A \sim B$) : questo operatore connette gli ingressi del blocco B alla corrispondente uscita del blocco A e le uscite di B ai corrispondenti ingressi di A , in modo da rappresentare una ricorsione.

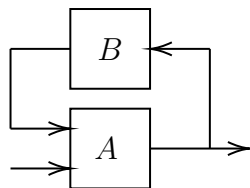


Figura 1.8: Operatore '~'

N-Delay (@) : questo operatore consente di ritardare di N campioni il segnale in ingresso ad esso.

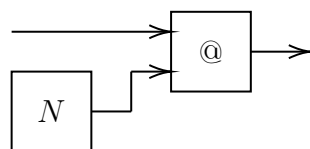
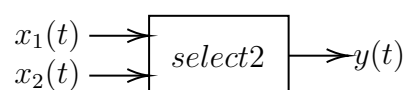


Figura 1.9: Operatore '@'

2-way selector ($select2(s)$) : questa funzione implementa un selettore a due vie. La funzione prende in ingresso due segnali $x_1(t)$ e $x_2(t)$. Al verificarsi della condizione $s(t)$ in argomento seleziona $x_1(t)$ se $s(t) = 0$ e $x_2(t)$ se $s(t) = 1$.

Figura 1.10: Funzione $select2$

Button : la funzione `button("label")` fa parte del gruppo delle GUI e permette di visualizzare un pulsante nell'ambiente di sviluppo che può essere premuto, cambiandone lo stato. Viene utilizzato come una costante moltiplicativa che può assumere valore 0 o 1. Come argomento la stringa "label" permette di definire una etichetta da visualizzare nell'interfaccia grafica.

Slider : le funzioni `hslider("label",init,min,max,step)` e `vslider` permettono di visualizzare uno slider nell'interfaccia grafica dell'IDE controllabile durante l'esecuzione del programma. La prima implementa uno slider orizzontale e la seconda verticale.

Il parametro `init` inizializza lo slider ad un valore fissato ad ogni nuova compilazione, `min` imposta il valore minimo, `max` il valore massimo e `step` consente di impostare il passo (precisione).

Per una maggiore completezza e flessibilità, è possibile includere nell'argomento "label" dei *metadata*. Ad esempio includendo `[style:knob]` lo slider verrà visualizzato come un potenziometro ed includendo `[midi: ctrl num]` è possibile assegnare un controllo MIDI allo slider. Quest'ultima opzione verrà utilizzata in seguito per mappare gli switches e i potenziometri presenti nella SHARC Audio Module Audio Project Fin (1.1.1) ai vari slider utilizzati.



(a) Slider orizzontale



(b) Slider in formato knob

Figura 1.11: Esempi di sliders

process : è l'istruzione fondamentale di Faust ed è equivalente al `main` presente in C/C++. Ogni programma Faust, per poter essere valido, deve contenere questa istruzione.

1.2.3 Ambiente di sviluppo

Una particolarità che contraddistingue Faust da altri linguaggi è l'assenza di un ambiente di sviluppo in locale, difatti viene fornita una IDE completamente

online all'indirizzo <https://fausteditor.grame.fr>.

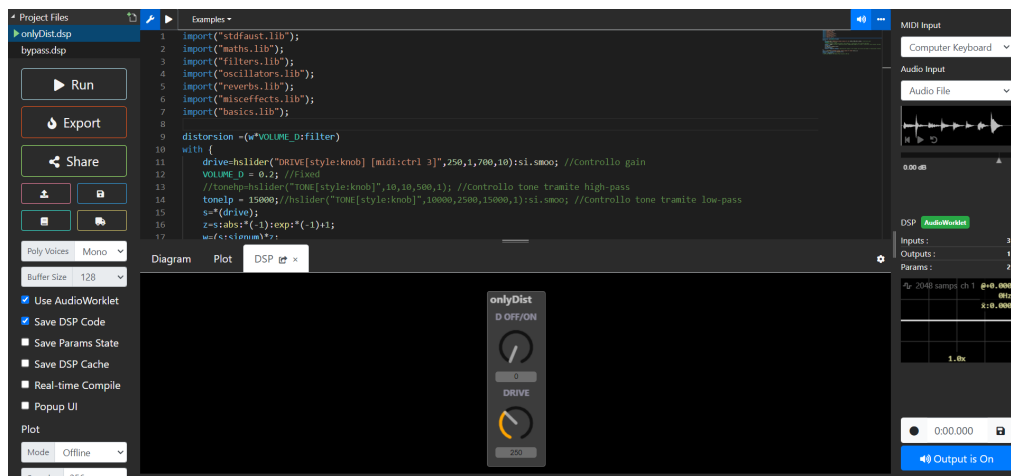


Figura 1.12: Interfaccia della Faust IDE

L'ambiente permette di compilare e di processare in real-time codici Faust sopra implementati. Grazie alla GUI (*Graphical User Interface*) è possibile controllare i vari parametri in maniera user-friendly durante l'esecuzione del programma, come si può notare dalla Figura 1.12. Essendo Faust un linguaggio basato sui diagrammi a blocchi, come accennato in 1.2.1, vi è la possibilità di visualizzare i blocchi automaticamente creati nella sezione "Diagram". Questa funzione può risultare molto utile in fase di debugging, in quanto permette di visualizzare in maniera chiara e intuitiva il flusso logico del programma. L'IDE supporta anche il protocollo MIDI, con l'interessante possibilità di poter utilizzare la tastiera del computer come MIDI controller. Come segnale audio da processare, l'ambiente mette a disposizione un breve sample con l'opzione di poter importarne uno personalizzato, alternativamente è possibile utilizzare una sorgente esterna (ad esempio un microfono). Ultima utile feature si trova sotto la voce "Plot", in cui è possibile visualizzare in real-time l'andamento del segnale di uscita nel tempo e il suo spettrogramma.

1.2.4 Tools faust2xx

Faust mette a disposizione parecchi tools (*faust2xx*) che permettono di generare scripts compatibili con molte piattaforme ed architetture, esempi sono *faust2android* che genera un file *.apk* da installare su dispositivi Android, *faust2ios* per dispositivi Apple e *faust2sam* che permette di generare 3 files C++ importabili nell'ambiente di sviluppo CrossCore Embedded Studio e quindi implementabili su piattaforme SHARC Audio Module.

Capitolo 2

Design e implementazione degli effetti

In questo Capitolo vengono presentati in dettaglio gli effetti utilizzati per lo sviluppo del multieffetto, con la loro relativa implementazione in Faust.

Viene successivamente presentata la relativa interfaccia di controllo, con particolare attenzione al mapping tra i dispositivi di controllo fisici presenti nella Audio Project Fin (pulsanti e potenziometri) (1.1.1) e la loro controparte virtuale in Faust.

Il multieffetto è dotato di tre effetti: *distorsione*, *chorus* e *riverbero*. Per ogni effetto possono essere controllati due parametri: *drive* e *tone* per la distorsione, *speed* e *depth* per il chorus, *room* e *wet* per il riverbero.

2.1 Design degli effetti

Vengono ora discusse, in modo teorico, le scelte effettuate per la progettazione degli effetti.

2.1.1 Distorsione

Il *distorsore*, così come l'*overdrive* e il *fuzz*, si basa sul concetto di *non-linearità*. La sostanziale differenza tra i tre è la quantità di distorsione impressa al segnale.

Se infatti l'*overdrive* è un effetto quasi-lineare per piccole ampiezze e progressivamente sempre più non-lineare all'aumentare di questa, non si può dire lo stesso per il distorsore e il fuzz. Nel distorsore infatti si opera principalmente in regione non-lineare mentre il fuzz si può considerare un effetto totalmente non-lineare, il che comporta un segnale di uscita decisamente più "duro" all'ascolto. Per lo sviluppo del multieffetto è stato scelto il distorsore, una soluzione intermedia tra il leggero overdrive e il fuzz.

Considerando infatti un sistema lineare, in esso vale il principio di sovrapposizione degli effetti: la risposta di un sistema lineare ad una combinazione lineare di un certo numero di segnali è uguale alla combinazione lineare delle singole risposte relative ad ogni segnale d'ingresso.

Infatti, nel caso di due ingressi $x_1[n]$ e $x_2[n]$:

$$f(\alpha_1 x_1[n] + \alpha_2 x_2[n]) = \alpha_1 f(x_1[n]) + \alpha_2 f(x_2[n]) \quad (2.1)$$

A causa della loro non-linearità quindi, gli effetti sopra citati non soddisfano l'Equazione (2.1).

Un effetto costituisce a tutti gli effetti un sistema, è possibile quindi definirne una relazione ingresso-uscita caratteristica. Nel caso del distorsore è stata scelta la seguente funzione:

$$f(x) = \operatorname{sgn}(\alpha x)(1 - e^{-\alpha|x|}) \quad (2.2)$$

con il parametro α che indica il "drive", ossia l'ammontare della distorsione prodotta.

Si può rappresentare graficamente tale funzione:

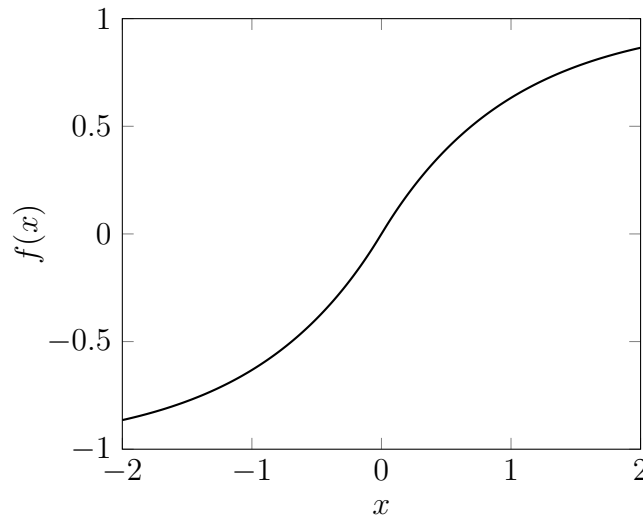


Figura 2.1: Andamento della funzione distorsione

Per diversi valori del parametro α si ottiene:

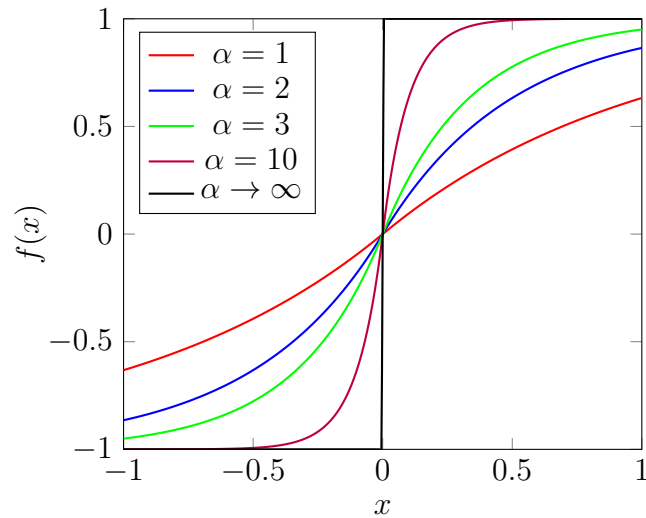


Figura 2.2: Andamento della funzione distorsione per diversi valori di α

Si può notare come al tendere del drive all'infinito, prevale sempre di più la funzione segno, comportando così una totale distorsione: il segnale di uscita potrà assumere solo due valori.

L'effetto che la funzione distorcente produce viene chiamato *clipping* e consiste in un "taglio" della forma d'onda superato un certo valore di ampiezza del segnale di ingresso [9] [10].

Ad esempio, considerando un segnale sinusoidale in ingresso al distorsore con drive $\alpha = 5$, l'effetto prodotto è chiaramente visibile dalla seguente figura:

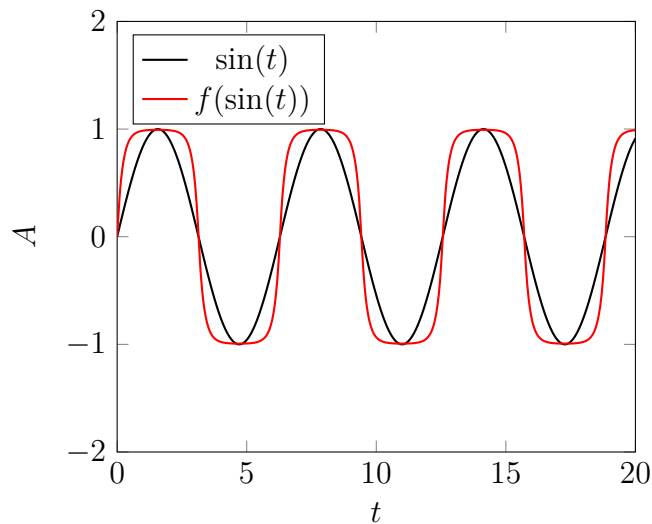


Figura 2.3: Risposta nel tempo del distorsore ad un ingresso sinusoidale

E' evidente inoltre come, all'aumentare del drive, le transizioni del segnale distorto diventano sempre più veloci, causando quindi una forte distorsione

armonica che, al contrario di molte altre applicazioni è ben accetta, dato che arricchisce di armoniche il suono in uscita.

La funzione distorcente (2.2) è stata favorita rispetto alle altre per via del suo compromesso tra overdrive e fuzz, con la possibilità di avere un suono quasi pulito per valori di α prossimi ad 1 e di avere potenti distorsioni al crescere di esso.

Nel codice Faust sviluppato viene anche utilizzato un *filtro passa-basso* in serie all'uscita del distorsore. Questo permette di avere un controllo sul cosiddetto "tone" e quindi di enfatizzare frequenze più basse per un suono più "cupo" o, in alternativa, di utilizzare tutto lo spettro di frequenze ricoperto dalla chitarra.

Il filtro passa-basso utilizzato è un filtro di Linkwitz–Riley del 4° ordine.

Seppur tradizionalmente usato come filtro crossover nei diffusori acustici, può essere tranquillamente utilizzato per gli scopi sopra descritti.

Esso è costituito da due filtri di Butterworth del 2° ordine in cascata, permettendo di ottenere uno slope di 80 *dB/decade*.

La scelta di questo filtro rispetto agli analoghi di Chebyshev ed ellittici, è basata sul fatto che non si hanno particolari esigenze di banda di transizione stretta. Viene preferito ad un singolo filtro di Butterworth a causa di bug delle funzioni Faust che implementano tali filtri, le quali non permettono il corretto funzionamento del codice.

Il range di frequenze fondamentali di una chitarra elettrica standard varia da 80 *Hz* a 1200 *Hz*, considerando le armoniche presenti a frequenze multiple delle fondamentali e la distorsione armonica provocata dal distorsore in sé, il range di frequenze si estende anche oltre i 10 *kHz*.

Il filtro avrà una frequenza di taglio controllabile variabile da 2.5 *kHz* (fondamentali) a 15 *kHz* (armoniche).

2.1.2 Chorus

Il *chorus* fa parte della categoria degli effetti basati su delay line, in essa ne fanno parte anche il *flanger* e il *vibrato*.

L'effetto chorus si può ascoltare quando segnali musicali di simile timbro e tonalità vengono suonati all'unisono. L'effetto è naturalmente percepibile quando ad esempio un gruppo di violinisti o cantanti suonano contemporaneamente: seppur di poco, il suono uscente da ogni strumento o corde vocali è diverso in termini di tempo e frequenza (da qui il nome dell'effetto).

L'effetto chorus simula proprio queste variazioni di tempo e di tonalità, dando quindi l'effetto di percepire più strumenti suonati contemporaneamente suo-

nandone uno solo.

Il funzionamento di principio si basa semplicemente nell'aggiungere al segnale originale una sua copia ritardata di M campioni, con M non costante. In formule:

$$y[n] = x[n] + gx[n - M(n)] \quad (2.3)$$

Per maggiore chiarezza, è possibile rappresentare la precedente mediante uno schema a blocchi:

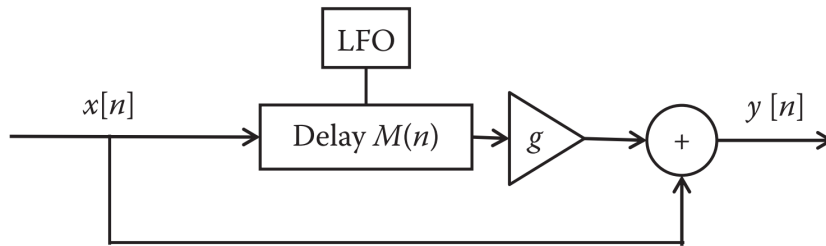


Figura 2.4: Schema a blocchi di un generico chorus

Con $M(n)$ funzione ritardante. Nel caso del chorus, il numero di campioni di ritardo M (lunghezza del delay) è variabile nel tempo. In particolare, viene utilizzato un *low-frequency oscillator* (LFO) con forma d'onda sinusoidale, non essendo però l'unica scelta possibile.

$$M(n) = d_{cost} + \frac{1}{2}A(1 + \sin(2\pi fn)) \quad (2.4)$$

Con d_{cost} delay costante (10 *ms*) e A ampiezza massima dell'oscillatore (parametro *depth*), variabile da 0 *ms* a 30 *ms*. L'aggiunta del ritardo costante consente di non scendere al di sotto del limite di 10 *ms*, oltre il quale si avrebbe l'effetto flanger. La frequenza di oscillazione f (parametro *speed*) è variabile da 0 *Hz* a 2 *Hz*.

Interpolatore lineare

La durata del delay è modulata dalla funzione $M(n)$. Come intuibile dalla notazione, essendo la funzione $M(n)$ a valori reali (non intero) e $x[n]$ definito solo per n interi, è chiaro che $x[n - M(n)]$ non può sempre esistere.

Una prima semplice soluzione potrebbe essere quella di arrotondare al numero intero di campioni più vicino ma sebbene tale approccio non sia computazionalmente esoso, non consentirebbe di raggiungere un sufficiente grado di fluidità (smoothness) di variazione del delay.

Per implementare un *delay frazionario* viene perciò scelta l'*interpolazione*, in

particolare quella lineare (o del primo ordine).

L'interpolazione si basa sullo stimare un valore di una funzione continua conoscendo solo valori discreti di essa. Considerando infatti un generico segnale $x[n]$ e un η ($0 \leq \eta \leq 1$) rappresentante la distanza di interpolazione tra il campione n e $n + 1$, si definisce il valore interpolato $\hat{x}[n + \eta]$ come:

$$\hat{x}[n + \eta] = x[n](1 - \eta) + x[n + 1] \cdot \eta \quad (2.5)$$

La delay line viene quindi realizzata interpolando linearmente tra il campione $n - \beta$ e $n - \beta - 1$, con β ritardo intero. Sapendo che il ritardo è dato dalla (2.4), si può facilmente ricavare β notando che in qualunque istante, essendo reale, M può essere scritto come somma della sua *parte intera* ($\lfloor M \rfloor$ o $\text{floor}(M)$) e della sua *parte frazionaria* o *mantissa* ($\{M\}$ o $\text{frac}(M)$):

$$M = \text{floor}(M) + \text{frac}(M) \quad (2.6)$$

La funzione *floor* arrotonda M con il più grande intero minore o uguale ad esso. Dualmente, si può introdurre la funzione *ceil*: arrotonda M col il più piccolo intero maggiore o uguale ad esso. Risulta quindi evidente che:

$$\beta = \text{floor}(M) \quad , \quad \beta - 1 = \text{ceil}(M) \quad (2.7)$$

Il parametro η nella (2.5) viene chiamato *delay frazionario* e si ottiene considerando la parte frazionaria di M , ossia:

$$\eta = \text{frac}(M) \quad (2.8)$$

Utilizzando quindi i risultati ottenuti fin'ora e applicandoli alla (2.5) si giunge all'espressione che rappresenta il segnale uscente dalla delay line:

$$u[n] = x[n - \text{floor}(M)](1 - \text{frac}(M)) + x[n - \text{ceil}(M)](\text{frac}(M)) \quad (2.9)$$

Equivalente al seguente schema a blocchi:

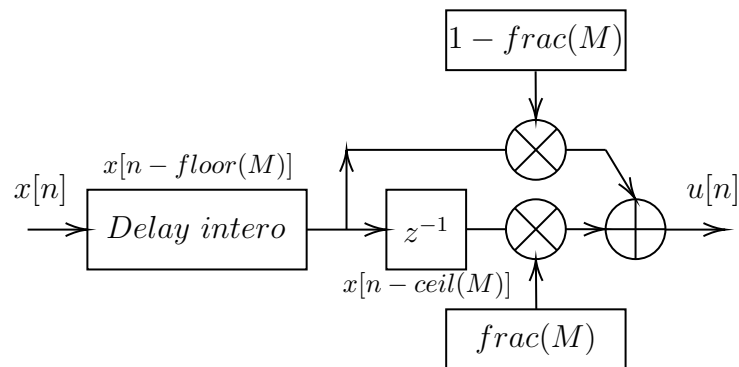


Figura 2.5: Schema a blocchi di una delay line interpolata linearmente

L'effetto acustico che produce un ritardo modulato è quello di un'alterazione del "pitch" del segnale originale che, miscelato con quest'ultimo, genera il caratteristico effetto di più strumenti suonati all'unisono precedentemente introdotto. La modulazione del delay porta infatti ad un allungamento o ad un accorciamento della forma d'onda, i quali si traducono in variazioni di frequenza. [11][9][12]

2.1.3 Riverbero

Si consideri un generico ambiente acustico al cui interno sono presenti una sorgente sonora (es. persona che parla) ed un ascoltatore. L'onda sonora emessa dalla sorgente non compie solamente un percorso diretto verso l'ascoltatore ma viene riflessa da pareti, pavimenti o oggetti presenti all'interno dell'ambiente stesso. L'ascoltatore riceve quindi copie ritardate e attenuate dell'onda originale emessa dalla sorgente. Queste copie danno origine al cosiddetto effetto *riverbero* e sono fondamentali per dare la percezione di spazialità. A differenza dell'eco, dove sono percepibili le distinte repliche ritardate (di almeno 40 ms), nel riverbero questo non è possibile in quanto esse arrivano all'ascoltatore con tempi molto più brevi.

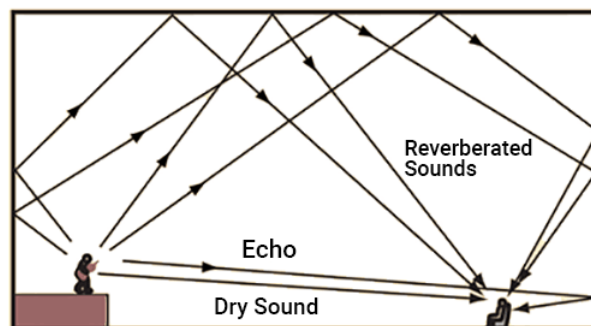


Figura 2.6: Raffigurazione di un ambiente in presenza di riverbero

Per poter emulare il riverbero all'interno di una stanza si è utilizzato "Freeverb" [11], un *riverberatore di Schroeder* di pubblico dominio sviluppato da "Jezar at Dreampoint" e largamente utilizzato, specialmente nei software di elaborazione audio open-source.

Il riverberatore di Schroeder è formato da più *riverberatori piani* di Schroeder connessi in parallelo, seguiti da filtri all-pass in cascata.

Il *riverberatore piano* non è altro che un filtro comb ricorsivo con la seguente funzione di trasferimento e corrispondente equazione alle differenze:

$$H(z) = \frac{-a + z^{-D}}{1 - az^{-D}} \quad (2.10)$$

$$y[n] = ay[n - D] - ax[n] + x[n - D] \quad (2.11)$$

Schematizzato si ha:

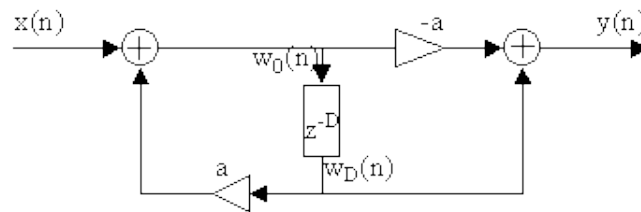


Figura 2.7: Schema di un riverberatore piano

Nel caso di Freeverb, il moltiplicatore a in retroazione viene sostituito da un filtro passa-basso $G(z)$, ottenendo così la seguente funzione di trasferimento:

$$LBCF_D^{f,d} = \frac{z^{-D}}{1 - z^{-D}G(z)} = \frac{z^{-D}}{1 - f \frac{1-d}{1-dz^{-1}} z^{-D}} \quad (2.12)$$

Il filtro passa-basso in retroazione ha lo scopo di ottenere una risposta più morbida per ogni replica del segnale in ingresso.

Il parametro d è detto "*damping*" ed il suo valore viene impostato di default a 0.2; esso controlla il $T60$ (tempo di decadimento di 60 dB, genericamente "tempo di riverbero") alle alte frequenze. Il parametro f (feedback), il fattore moltiplicativo del filtro passa-basso, è anche chiamato "*room*" e rappresenta la dimensione della stanza simulata nel riverbero. Dalla teoria dei controlli, il valore di f deve essere minore dell'unità per garantire la stabilità dell'effetto. Si può interpretare "room" come il $T60$ alle basse frequenze e, fisicamente, rappresenta la radice quadrata del coefficiente di riflessione alle basse frequenze di ogni muro presente nella stanza simulata.

I filtri all-pass in cascata hanno la seguente funzione di trasferimento approssimata:

$$AP_D^g \simeq \frac{-1 + (1 + g)z^{-D}}{1 - gz^{-D}} \quad (2.13)$$

Difatti questa rappresenta un all-pass solo per $g = \frac{\sqrt{5}-1}{2} \simeq 0.618$ ossia per l'inverso della sezione aurea.

In definitiva, ogni canale viene implementato seguendo lo schema a blocchi:

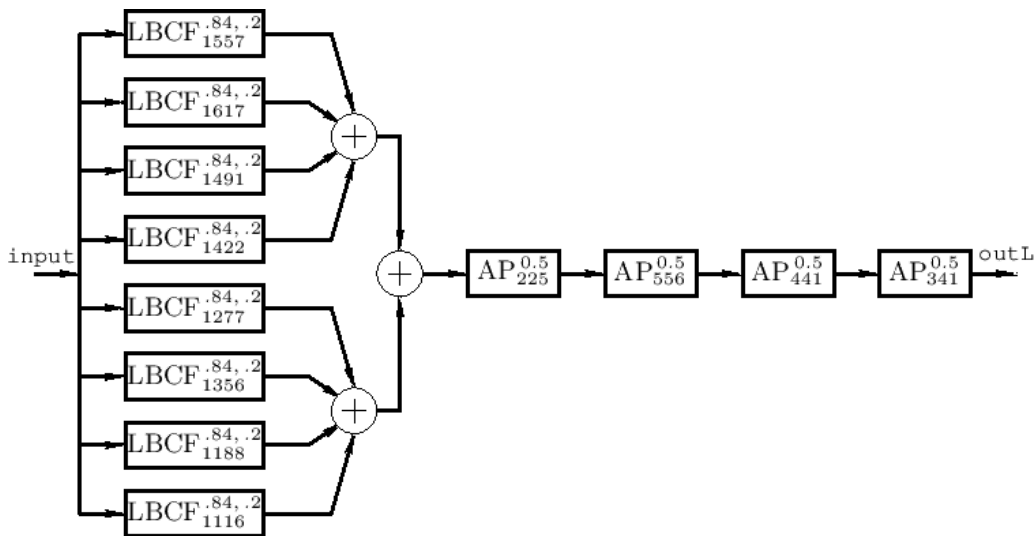


Figura 2.8: Schema a blocchi del "Freeverb"

Nel caso di riverbero stereo, ad uno dei due canali viene aggiunto un ulteriore ritardo pari al valore del parametro "*spread*" in modo da fornire un effetto stereofonico che enfatizzi la percezione di riverbero.

Il segnale riverberato in uscita dall'effetto viene miscelato con il segnale in ingresso moltiplicato per il fattore "*wet*" ($0 \leq wet \leq 1$). Se $wet = 1$ il segnale in uscita avrà solo la componente riverberata.

I valori dei ritardi D sono stati determinati a priori dallo sviluppatore. I parametri d ("damping"), g (polo/zero dei filtri all-pass) e "*spread*" sono impostati in compile-time mentre f ("room") e "*wet*" sono variabili dall'utente in run-time mediante i potenziometri.

2.2 Implementazione in Faust

In questa Sezione vengono implementati in Faust gli effetti progettati e dettagliati in Sezione 2.1. Il codice relativo completo verrà riportato solamente

a fine Capitolo.

2.2.1 Distorsione

Seguendo quanto formalizzato in 2.1.1, in Faust viene implementata la funzione distortante (2.2) utilizzando le funzioni presenti nella libreria `maths.lib`. In particolare, vengono inizializzati due `hslider` per il controllo dei parametri *drive* (variabile da 1 a 750) e *tone* (variabile da 2500 Hz a 15000 Hz).

In serie ad ogni `hslider` viene aggiunto il blocco `smoo`, facente parte della libreria `signals.lib`. Esso altro non è che un filtro passa-basso ad un sol polo con frequenza di taglio a 7 Hz che permette di rendere più "dolci" le rapide variazioni dello slider (controllato da un potenziometro) ed evitare quindi fastidiosi clipping.

Il filtro di Linkwitz-Riley utilizzato per modificare il *tone* viene implementato passandolo come parametro alla funzione `lowpassLR4`, presente nella libreria `filters.lib` e posizionato in serie all'uscita del blocco distortante precedentemente moltiplicato per una costante di attenuazione pari a 0.2. L'implementazione mediante schematizzazione a blocchi è la seguente:

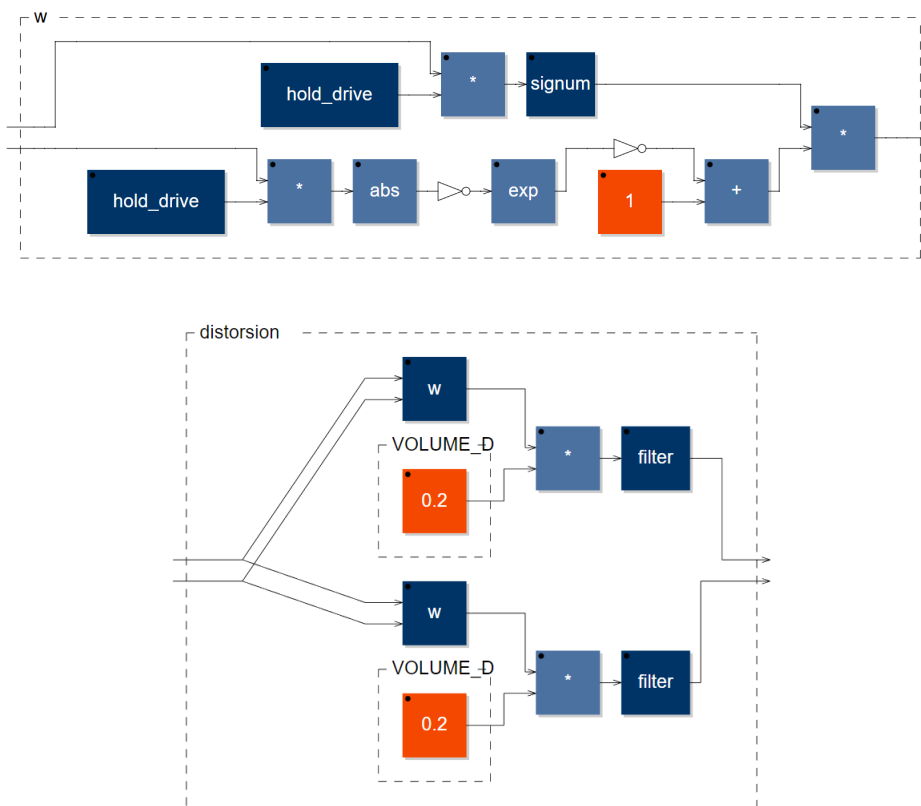


Figura 2.9: Implementazione Faust della distorsione mediante schemi a blocchi

2.2.2 Chorus

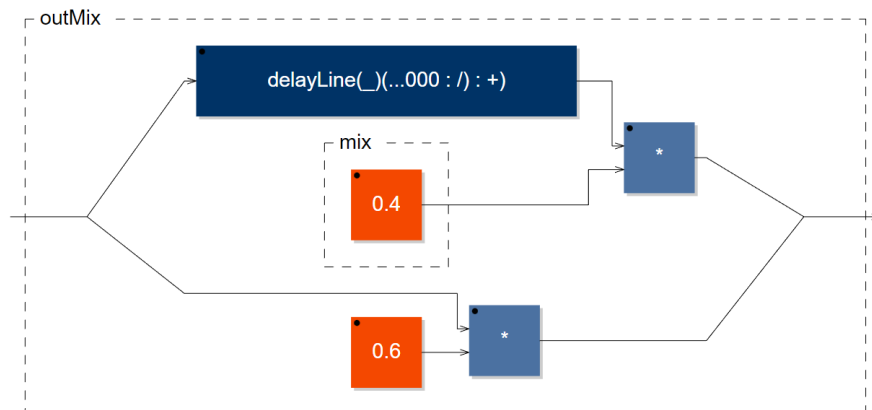
Come nella distorsione, i due parametri controllabili *speed* e *depth* possono essere variate mediante potenziometri, implementati con la funzione `hslider`. Come discusso in 2.1.2, *speed* è variabile da 0 *Hz* a 2 *Hz* mentre *depth* può essere variato da 480 campioni corrispondenti a 10 *ms* (delay costante) a 1440 campioni corrispondenti a 30 *ms* (nell'ipotesi di campionamento a 48 *KHz*). Nel tali valori sono normalizzati in modo da poterli rappresentare in millisecondi e quindi di variarne gli estremi in compile-time in modo più immediato. A differenza del distorsore, non è stato possibile inserire in cascata allo slider *speed* la funzione smussante a causa di bug interni al compilatore Faust che ne causano errori di computazione.

La modulante (2.4) viene implementata utilizzando l'oscillatore `oscsin` presente nella libreria `oscillators.lib`.

L'interpolatore lineare viene sviluppato sulla falsa riga dello schema riportato in Figura 2.5. Viene definita una funzione (`delayFin`) i cui argomenti sono il segnale da processare e la funzione ritardante, i due vengono processati mediante l'operatore `@` per il ritardo e mediante le funzioni primitive `ceil` e `floor`.

Il segnale con chorus viene poi miscelato con l'originale tramite un fattore moltiplicativo *mix*, impostato di default a 0.4.

Lo schema a blocchi risultante dalla compilazione in Faust non è affatto dissimile a quello proposto nella trattazione teorica:



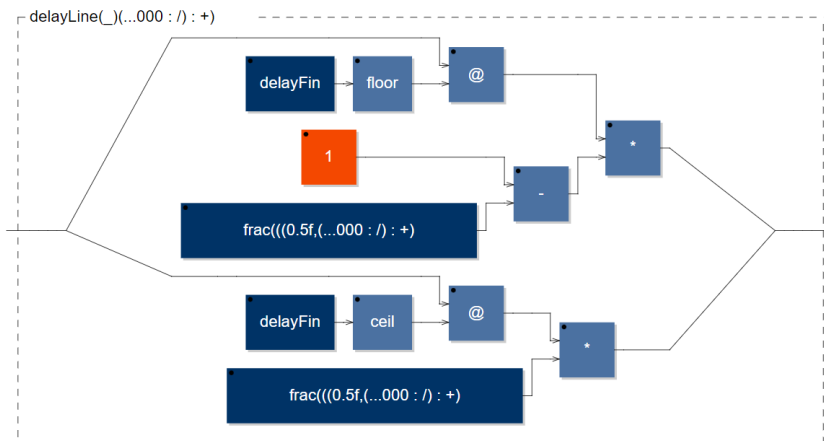


Figura 2.10: Implementazione Faust del chorus mediante schema a blocchi

2.2.3 Riverbero

La realizzazione interna del riverbero "Freeverb" è stata discussa in 2.1.3, lo sviluppo in Faust si limita ad assegnare i giusti parametri alla funzione `stereo_freeverb(fb1, fb2, damp, spread)` presente nella libreria `reverbs.lib`:

- `fb1`: parametro *room*. Controllato da un `hslider` variabile da 0 a 0.9 in real-time.
- `fb2`: parametro *g*. Fissato a 0.5 in compile-time.
- `damp`: parametro *damping*. Fissato a 0.2 in compile-time
- `spread`: parametro fissato a 23 campioni in compile-time, come suggerito dallo sviluppatore.

Nei pedali riverbero reperibili in commercio spesso vi si trova il parametro *wet* che, similmente al parametro *mix* del chorus, permette di controllare la quantità di miscelamento del segnale originale con quello riverberato. Così come tutti gli altri parametri, viene controllato mediante `hslider` variabile da 0 (per il solo segnale originale) a 1 (per il solo segnale riverberato).

Per completezza, viene riportato lo schema a blocchi generato alla compilazione del codice:

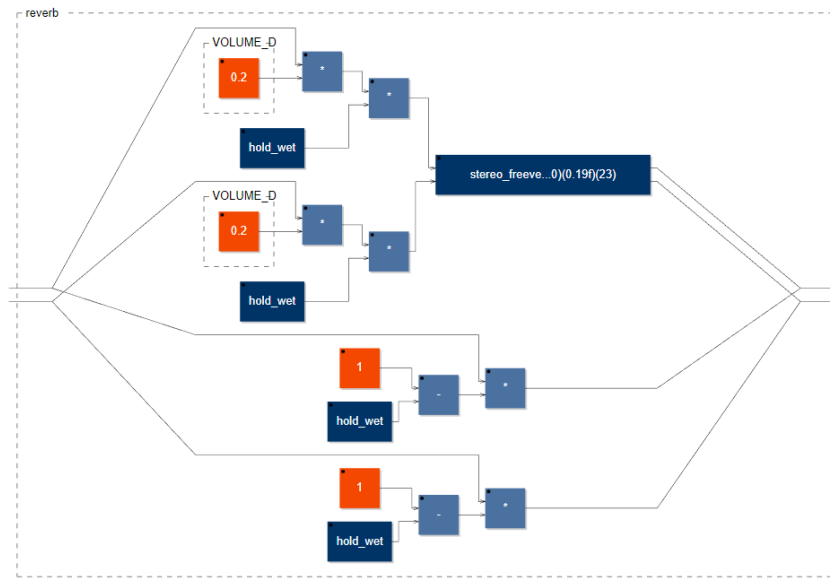


Figura 2.11: Implementazione Faust del riverbero mediante schemi a blocchi

2.2.4 Volume

In aggiunta ai 3 effetti viene anche implementato un semplice controllo del volume del segnale in uscita dalla catena cosicché si possa avere una maggior flessibilità nell'uso del multieffetto o, in alternativa, utilizzare il sistema come pedale volume.

La sua realizzazione in Faust avviene mediante uno slider variabile da 0 a 1 in moltiplicazione ai due segnali (canale destro e canale sinistro) in uscita dal multieffetto. Di seguito ne viene riportato lo schema a blocchi Faust che, oltretutto, coincide con quello dell'intero sistema.

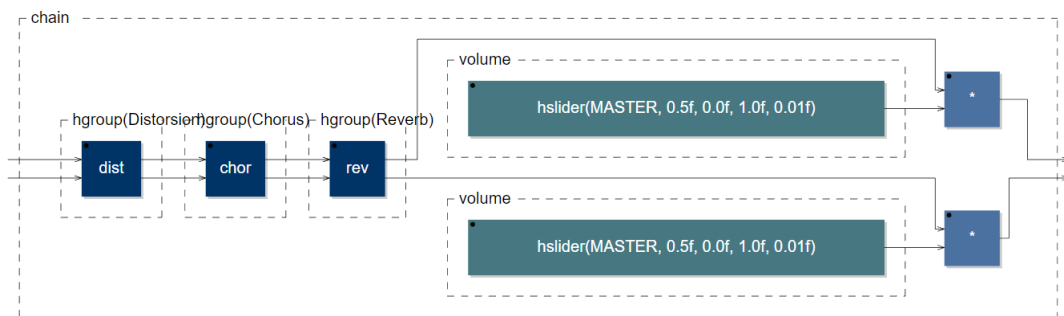


Figura 2.12: Implementazione Faust del controllo volume

2.3 Interfaccia di controllo

Il controllo del multieffetto e la gestione degli ingressi e delle uscite viene affidata alla Audio Project Fin, introdotta in 1.1. Nella figura di seguito se ne può vedere una schematizzazione di principio:

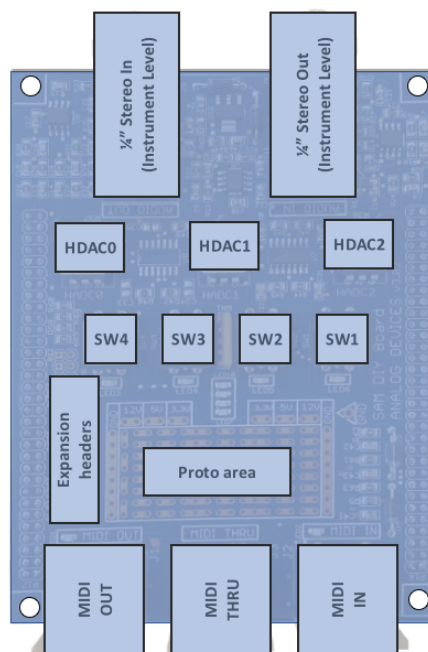


Figura 2.13: Suddivisione degli elementi costituenti la Audio Project Fin

I potenziometri (HDACx) e i pulsanti (SWx) possono essere usati per controllare i parametri di algoritmi implementati nella SHARC Audio Module mediante il protocollo MIDI. Questi sono mappati come controller MIDI in base alle seguenti tabelle:

Potenziometro	Numero del controller
HADC0	CC-2
HADC1	CC-3
HADC2	CC-4

Pulsante	Numero del controller
SW1	CC-102
SW2	CC-103
SW3	CC-104
SW4	CC-105

Tabella 2.1: Mapping tra dispositivi fisici e controller MIDI

Il numero del controller MIDI associato ad ogni dispositivo fisico viene inserito in Faust all'interno di ogni `hslider` sotto forma di metadata, seguendo la notazione `[midi:ctrl num]`. Ogni controller viene mappato in un range da 0 a 127. Per scelta costruttiva il mapping viene effettuato solamente in modo continuo, i pulsanti sono perciò mappati con valore 0 per l'assenza di pressione e valore 127 in presenza di pressione. In Faust si ricorre comunque agli slider, con estremi 0 e 1 e passo 1. Se ne può vedere un esempio di seguito:

```
1-hslider("D OFF/ON [style: knob] [midi:ctrl 104]",0,0,1,1)
```

Figura 2.14: Esempio di `hslider` utilizzato come pulsante

2.3.1 Gestione hardware dei parametri del multieffetto

Nella Sezione 2.2 sono stati descritti i 7 parametri controllabili dai potenziometri. L'insufficiente numero di quest'ultimi porta a dover definire un'interfaccia che preveda l'uso di un selettore, implementato mediante un pulsante, che permetta di poter selezionare la coppia di parametri da variare.

Il pulsante SW4 viene esclusivamente dedicato per questo scopo, difatti in base al numero di pressioni è possibile selezionare uno dei tre effetti da controllare, oltre alla possibilità di bypassare il multieffetto. Il bypass consente di trasferire in uscita il segnale in ingresso senza effettuarne elaborazioni, rendendo quindi il sistema "trasparente".

I restanti tre pulsanti vengono utilizzati per controllare l'accensione e lo spegnimento di ogni effetto.

Di seguito viene illustrata la gestione dei pulsanti:

- **SW4** : Selezione (s)
 - $s = 0$: Bypass
 - $s = 1$: Modifica parametri distorsione
 - $s = 2$: Modifica parametri chorus
 - $s = 3$: Modifica parametri riverbero
- **SW3** : Distorsione ON/OFF
- **SW2** : Chorus ON/OFF
- **SW1** : Riverbero ON/OFF

e dei potenziometri:

- **HADC0** : Master volume
- **HADC1** :
 - `distorsion.drive`, se $s = 1$
 - `chorus.depth`, se $s = 2$
 - `reverb.room`, se $s = 3$
- **HADC2**:
 - `distorsion.tone`, se $s = 1$
 - `chorus.speed`, se $s = 2$
 - `reverb.wet`, se $s = 3$

Viene perciò riservato il potenziometro HADC0 al controllo del volume mentre lo scopo di HADC1 e HADC2 viene modificato a seconda del valore del selettore (SW4). Tale valore non modifica il funzionamento degli altri tre pulsanti, che sono esclusivamente impiegati per l'attivazione e lo spegnimento dei singoli effetti.

2.3.2 Gestione software dei parametri del multieffetto

A causa del paradigma funzionale di Faust, concetti classici presenti nella programmazione imperativa (es. in C) non possono essere sfruttati alla stessa maniera. In Faust infatti non è possibile ridefinire una variabile, né tantomeno utilizzare costrutti logici (*if* su tutti) nel modo tradizionale. Sebbene la libreria `basics.lib` metta a disposizione la funzione *if* ed esternamente sia strutturata in modo analogo a quella presente nei linguaggi imperativi, internamente viene implementata mediante la funzione primitiva `select2` (brevemente riassunta in 1.2.2) la quale elabora sempre tutti e due i segnali di ingresso, anche quando la condizione in argomento non è verificata, comportando un significativo onere computazionale aggiuntivo. L'utilizzo di questa funzione è però inevitabile, in quanto chiave di ogni possibile implementazione di istruzioni condizionali. Per lo sviluppo di algoritmi Faust destinati all'utilizzo mediante IDE o architetture di alto livello, tali funzioni non comportano evidenti problematiche; se destinati invece a sistemi embedded è necessario effettuare valutazioni di ottimizzazione delle risorse utilizzate.

La gestione dei controlli in Faust viene effettuata mediante un sample and hold

(funzione `sAndH(trig)` della libreria `basics.lib`). L'unico parametro presente è *trig*, ossia la condizione di trigger tale per cui se essa vale 0, il valore corrente del segnale in ingresso viene mantenuto e portato in uscita, se vale 1 la funzione viene bypassata.

Per ogni parametro controllabile viene definita una variabile ottenuta portando il relativo slider in ingresso al sample and hold, il quale mantiene il valore corrente del parametro solamente quando il valore del selettore (*s*) è relativo all'effetto da variare. Quest'ultima condizione viene implementata utilizzando la funzione `select2` che seleziona uno tra due degli ingressi costanti di ampiezza 0 e 1 al verificarsi dell'uguaglianza tra il valore di *s* corrente e quello relativo all'effetto di cui variarne il parametro. Di seguito se ne può vedere un esempio:

```
hold_drive = drive:sAndH((0,1:select2(sel==1)));
```

Figura 2.15: Esempio di sample and hold

sel è il valore corrente del selettore, è realizzato tramite la funzione `counter(trig)` il cui trigger è lo slider corrispondente al pulsante SW4. Viene poi effettuata una operazione di modulo (*mod4*) così da resettare il contatore ogni quattro pressioni.

L'attivazione dei singoli effetti è demandata ai singoli pulsanti implementati tramite `hslider`, come accennato ad inizio Sezione e riportato nell'esempio di Figura 2.14. In particolare viene utilizzata la funzione `bypass_fade(n,b,e)`, i cui argomenti *b* ed *e* sono rispettivamente l'istruzione di controllo (sliders sopra citati) ed il circuito da bypassare (i singoli effetti). Tale funzione, oltre al bypass dei singoli effetti, applica un crossfade di *n* campioni, con *n* impostato a 500 (equivalenti a 10 *ms*, per una frequenza di campionamento di 48 *kHz*). Il crossfade permette una transizione più "morbida" tra il segnale originale e quello elaborato con effetti, evitando quindi possibili clipping fastidiosi all'ascolto.

2.3.3 Gestione grafica dei parametri del multieffetto

Parallelamente all'utilizzo mediante la SHARC Audio Module e la Audio Project Fin, il multieffetto può essere utilizzato direttamente dall'ambiente di sviluppo online di Faust, ragion per cui è stata implementata una comoda interfaccia grafica per poter controllare i parametri in simil modo.

Nella GUI i controlli vengono raggruppati per effetto mediante la funzione `hgroup`. Per maggiore chiarezza, viene anche visualizzato il numero corrente

del selettore (funzione `hbargraph`).

Se ne può vedere l'interfaccia grafica nella figura si seguito:



Figura 2.16: Faust GUI del multieffetto

Lo knob "control" è la mappatura del pulsante SW4 (di selezione). Come evincibile dalla Figura, a causa della metodologia utilizzata per implementare i controlli, congiuntamente al paradigma di Faust, lo knob viene automaticamente inserito anche in ogni raggruppamento. Questa situazione non voluta comporta che ogni qualvolta l'utente voglia utilizzare il selettore, debba variare allo stesso modo anche quelli presenti nei vari raggruppamenti.

2.4 Codice completo

Per completezza, di seguito viene riportato il codice completo del multieffetto.

```

1 import("stdfaust.lib");
2 import("maths.lib");
3 import("filters.lib");
4 import("oscillators.lib");
5 import("reverbs.lib");
6 import("misceffects.lib");
7 import("basics.lib");
8
9 /*****CONTROLLI*****/
10
11 hold_drive = drive:sAndH((0,1:select2(sel==1)));
12 hold_tonelp = tonelp:sAndH((0,1:select2(sel==1)));
13 hold_depth = depth:sAndH((0,1:select2(sel==2)));
14 hold_speed = speed:sAndH((0,1:select2(sel==2)));
15 hold_room = coeffLPF:sAndH((0,1:select2(sel==3)));
16 hold_wet = wet:sAndH((0,1:select2(sel==3)));
17 bypassctrl = 0,1:select2(sel==0);
18
19 /*****DISTORSORE*****/
20
21 drive = hslider("DRIVE[style:knob][midi:ctrl 3]",200,1,750,10):si.smoo;
22 tonelp = hslider("TONE[style:knob][midi:ctrl 4]",10000,2500,15000,1):si.smoo;
23 VOLUME_D = 0.2;
24 w=(*(hold_drive)):signum*((*(hold_drive)):abs:*(-1):exp:*(-1)+1);
25 filter = fi.lowpassLR4(hold_tonelp);
26 distorsion = _,_<:(w*VOLUME_D:filter),(w*VOLUME_D:filter);
27
28 /*****CHORUS*****/
29
30 A_MAX = 30;
31 DELAY_CONST = 10;
32
33 speed = hslider ("SPEED [style:knob][midi:ctrl 3]" , 0.4 , 0 , 2 , 0.001):si.smoo;
34 depth = hslider ("DEPTH [style:knob][midi:ctrl 4]" , 0.33 , 0 , 1 , 0.001);
35 mix = 0.4;
36
37 //Delay variabile
38 varDelay = (0.5*(1+oscsin(hold_speed)))*(depth*A_MAX*SR/1000);
39
40 //Delay costante
41 constDelay = DELAY_CONST*SR/1000;
42
43 //Delay finale
44 delayFin=varDelay+constDelay;
45
46 //Interpolatore lineare come delay line
47 delayLine(x,d) = x <: @(d:floor)*(1-frac(d)), @(d:ceil)*(frac(d)) :> _;
48 //Mix segnale processato con quello in ingresso
49 outMix = _<: delayLine(_,delayFin)*mix, _*(1-mix) :> _;
50

```

Capitolo 2 Design e implementazione degli effetti

```
51 chorus = outMix,outMix;
52
53 /*****RIVERBERO*****/
54
55 VOLUME_R = 0.2; //Fissato
56 coeffLPF = hslider("ROOM[style:knob][midi:ctrl 3]",0.7,0,0.9,0.01) : si.smoo;
57
58 wet = hslider("WET[style:knob][midi:ctrl 4]", 0.33,0,1,0.001): si.smoo;
59
60 coeffAPL = 0.5;
61
62 dampingAPF = 0.2;
63
64 spread = 23;
65
66 reverb = _,_<:(*(VOLUME_R)*(hold_wet),*(VOLUME_R)*(hold_wet):stereo_freeverb(hold_room,
67     coeffAPL, dampingAPF, spread)),*(1-hold_wet),*(1-hold_wet)):>_,_;
68
69
70 /*****CONNESSIONI*****/
71
72 NF = 500; |
73
74 volume = hslider("[1]MASTER [style: knob] [midi ctrl 2]",0.5,0,1,0.01);
75
76 //Controlli ON/OFF(bypass) degli effetti.
77 dist = bypass_fade(NF,1-hslider("D OFF/ON [style: knob] [midi:ctrl 104]",0,0,1,1),distorsion);
78 chor = bypass_fade(NF,1-hslider("C ON/OFF [style: knob] [midi:ctrl 103]",0,0,1,1),chorus);
79 rev = bypass_fade(NF,1-hslider("R ON/OFF [style: knob] [midi:ctrl 102]",0,0,1,1),reverb);
80
81
82 /*****MAIN*****/
83
84 chain = hgroup("[2]Distorsion",dist):hgroup("[3]Chorus",chor):
85     hgroup("[4]Reverb",rev):*(volume),*(volume);
86
87 sel = counter(hslider("[0]CONTROL [style: knob] [midi:ctrl 105]",0,0,1,1)) : %(4);
88 dispSel = sel : hbargraph("SELEZIONE[style:numerical]",0,3);
89
90 process = bypass_fade(NF,bypassctrl,chain),dispSel;
```

Figura 2.17: Codice Faust del multieffetto

Capitolo 3

Porting su SHARC Audio Module

In questo Capitolo vengono discusse le modalità di porting dell'algoritmo Faust su SHARC Audio Module, con particolare riferimento alle problematiche riscontrate e alle loro relative soluzioni.

3.1 Compatibilità

Sin dalla versione 2.6.1 di Faust, risalente al 2018, l'ambiente di sviluppo CrossCore Embedded Studio, più specificatamente il Bare Metal Framework (1.1.3), non è compatibile con Faust. Difatti se si esporta il codice direttamente nella IDE con l'opzione "*sam*" (SHARC Audio Module) e lo si carica nell'ambiente di sviluppo sopra citato, al momento della compilazione verranno generati numerosi errori che non ne permettono il build. Il motivo è da ricercarsi nei continui aggiornamenti di Faust e nel mancato interesse da parte di Analog Devices di continuarne lo sviluppo in quella direzione.

Per aggirare questo problema, il codice Faust va compilato (convertito in C++) utilizzando una vecchia versione di Faust compatibile, ad esempio proprio la 2.6.1.

3.1.1 Installazione di Faust su macchina Linux

Tutte le releases di Faust sono interamente disponibili e consultabili nell'apposita repository GitHub. Per poter compilare utilizzando una precedente versione si deve perciò necessitare di una macchina Linux¹ con il software Git installato.

E' inoltre necessario installare il pacchetto `build-essential` contenente l'utilità `make`, varie librerie ed il compilatore `gcc/g++`. Anche quest'ultimo, a causa della non retrocompatibilità della versione di C++ utilizzata nello sviluppo della release 2.6.1 di Faust con quella attualmente disponibile, necessita di

¹In tutta la trattazione si farà riferimento alla distribuzione Ubuntu.

Capitolo 3 Porting su SHARC Audio Module

essere installato ad una precedente versione. Le versioni di `gcc` anteriori alla 7 (verrà utilizzato `gcc-5`) si trovano all'interno della repository *xenial*. Questa va inserita nel file `sources.list` presente in APT (Advanced Package Tool, il sistema di gestione dei pacchetti in Ubuntu) mediante i seguenti comandi da terminale:

```
sudo vim /etc/apt/sources.list
deb http://dk.archive.ubuntu.com/ubuntu/ xenial main
deb http://dk.archive.ubuntu.com/ubuntu/ xenial universe
```

Durante l'esecuzione dei comandi non è rara la comparsa di errori riguardanti l'assenza di alcune chiavi pubbliche GPG (GNU Privacy Guard). L'aggiunta di queste ultime va effettuata con una specifica richiesta ad un server, con il comando:

```
sudo apt-key adv --recv-keys --keyserver keyserver.ubuntu.com
XXXXXXXXXXXXXXXX
```

Le cui 'x' indicano la chiave mancante ottenuta dall'errore originato. Successivamente si può procedere con l'installazione:

```
sudo apt update
sudo apt install g++-5 gcc-5
```

E con la modifica della versione del compilatore di default:

```
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/
gcc-5 5
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/
g++-5 5
```

La versione 2.6.1 di Faust può essere scaricata ripristinando una vecchia commit effettuata su GitHub attraverso le seguenti istruzioni:

```
git clone https://github.com/grame-cncm/faust.git
git reset --hard 97c682909
```

Lo standard C++ 17 utilizzato da Faust non coincide con quello presente di default in `gcc/g++`. Nella directory `build` della repository Faust vi si trova perciò il file *CMakeLists.txt*, il quale contiene la stringa `set (CMAKE_CXX_STANDARD 11)` che, settata al valore 17, ne modifica lo standard utilizzato.

A causa delle diverse distribuzioni e versioni Linux installate, potrebbero verificarsi problemi di mancanza di librerie. Nel caso specifico, la libreria *libmicrohttpd* è necessaria per la corretta installazione di Faust, difatti se mancante occorre installarla dall'opportuno sito web.

Altro errore molto comune riguarda il mancato casting tra alcuni tipi di dati presenti nei file di installazione dedicati all'HTTPD. Tale errore viene corretto nelle successive versioni di Faust che però non possono essere installate, date le finalità precedentemente descritte che invaliderebbero il downgrade. Per aggirare tale problema è sufficiente sostituire i file *HTTPDServer.cpp* e *HTTPDServer.h* presenti al percorso `/architecture/httpdlib/src/httpd/` con quelli disponibili, allo stesso indirizzo, nelle ultime releases di Faust.

Dopo aver concluso le operazioni preliminari è possibile procedere con l'installazione eseguendo i comandi `make` e `sudo make install` all'interno della directory Faust.

3.1.2 Compilazione/conversione dell'algoritmo

All'interno dell'IDE Faust è possibile, mediante l'opzione *"Download/Save as a local file"*, salvare l'algoritmo in locale in formato *.dsp*. Per convertire questo file in C++ si fa uso del tool `faust2sam` che, come accennato in 1.2.4, crea una cartella *.zip* contenente tre files (*fast_pow2.h*, *samFaustDSP.h* e *samFaustDSP.cpp*) compatibili con l'ambiente CrossCore Embedded Studio. Il comando da eseguire è:

```
faust2sam -midi nomeFile.dsp
```

dove l'opzione `-midi` consente di abilitare l'uso del protocollo MIDI.

Comparando il codice generato da `faust2sam` con quelli forniti da esempio dalla Analog Devices, si nota come ogni file *samFaustDSP.h* generato da diverse compilazioni differisca dagli stessi generati in locale con la procedura sopra descritta. La differenza, che ne comporta il mancato build all'interno di CrossCore Embedded Studio, sta nell'assenza della stringa `"myDsp* fDSP"` alla riga 47, all'interno della parola chiave `private`. Occorre perciò aggiungere tale stringa per garantirne la corretta compilazione.

3.2 Building e booting

Si riporta ora la procedura di compilazione e di caricamento dell'algoritmo convertito nella SHARC Audio Module.

3.2.1 Operazioni preliminari

Il processo di porting vero e proprio inizia con la creazione di un nuovo progetto Bare Metal Framework all'interno di CrossCore Embedded Studio, effettuabile selezionando il medesimo dall'apposita icona presente nella barra superiore.

Alla creazione viene chiesto se è previsto l'uso della Audio Project Fin (sì, nel caso in esame), del modulo A2B e di Faust. Per quest'ultimo è possibile selezionare se far elaborare l'algoritmo soltanto dal Core 1 o se da ambedue i cores (il Core 0 non si occupa dell'elaborazione audio). L'algoritmo multieffetto sviluppato è pensato per essere eseguito soltanto in un core (Core 1) perché Faust, essendo ad alto livello e non specifico per questa piattaforma, non consente lo scambio di informazioni tra i cores. Sebbene l'utilizzo di entrambi i cores ne raddoppi la latenza, le capacità computazionali vengono evidentemente aumentate, permettendo di implementare algoritmi Faust complessi e computazionalmente onerosi [13].

Come ultima opzione di progetto viene chiesta la frequenza di campionamento e la dimensione del buffer, impostati entrambi con i valori di default (48 *kHz* per la prima e 32 campioni per il secondo).

Alla creazione del progetto viene creata una cartella all'interno della workspace directory al cui interno si trovano varie sottocartelle, tra cui le tre dedicate ai singoli core. Per ognuna di esse sotto la cartella `src` si trova la cartella `faust`. L'importazione dell'algoritmo Faust convertito avviene quindi copiando i tre file `fast_pow2.h`, `samFaustDSP.h` e `samFaustDSP.cpp` nella suddetta directory. La procedura va ripetuta per ogni core utilizzato.

All'interno della cartella `src` si trovano diversi files dedicati al processing audio e all'elaborazione MIDI. In particolare è presente il file `callback_audio_processing.cpp` dove al suo interno sono pre-implementate varie funzioni, ognuna dedicata ad una specifica sezione del processing audio. La funzione `void processaudio_callback(void)` contiene l'algoritmo di elaborazione vero e proprio. Al suo interno infatti è presente un ciclo `for` che itera sulla lunghezza del buffer audio impostata alla creazione del progetto. Nell'utilizzo con Faust, tale ciclo si occupa solamente di indirizzare l'audio

elaborato con Faust ai buffer di uscita. Nelle recenti versioni del Bare Metal Framework, all'interno del *for* è anche presente un'istruzione condizionale:

```
if (true) {

    // Copy incoming audio buffers to the effects input buffers
    copy_buffer(audiochannel_0_left_in, audio_effects_left_in,
                AUDIO_BLOCK_SIZE);
    copy_buffer(audiochannel_0_right_in, audio_effects_right_in,
                AUDIO_BLOCK_SIZE);

    // Process audio effects
    audio_effects_process_audio_core1();

    // Copy processed audio back to input buffers
    copy_buffer(audio_effects_left_out, audiochannel_0_left_in,
                AUDIO_BLOCK_SIZE);
    copy_buffer(audio_effects_right_out, audiochannel_0_right_in,
                AUDIO_BLOCK_SIZE);

}
```

Questa, avendo sempre valore `true` di default, abilita un preset di effetti di esempio. Se non eliminata (o perlomeno modificata a `false`), l'algoritmo Faust caricato sulla SHARC Audio Module verrà sovrapposto alla catena di default già presente.

L'ultima (opzionale) operazione preliminare consiste nell'abilitare l'ottimizzazione del compilatore. Può essere attivata seguendo il percorso:

```
Properties - C/C++ Build - Settings - CrossCore SHARC C/C++
Compiler - General
```

E spuntando l'opzione *Enable optimization (-O)*.

3.2.2 Building

Per compilare l'algoritmo è sufficiente effettuare un `clean`, ossia un'opzione di CrossCore Embedded Studio che permette di scartare eventuali residui di build precedentemente effettuati e di compilare il progetto. L'operazione di building non è particolarmente rapida, difatti possono occorrere diversi minuti per compilare tutti i cores.

3.2.3 Debugging

Il caricamento dell'algoritmo sulla SHARC Audio Module richiede il collegamento di questa al PC tramite l'emulatore ICE-1000. Quest'ultimo va collegato all'apposita porta "DEBUG" a 10 pin presente sulla board e tramite USB al computer.

Prima dell'avvio del debug, occorre configurarne i parametri. Sotto la voce `Run - Debug configurations - Application with CrossCore Debugger` è possibile infatti scegliere il processore e l'emulatore utilizzato, nonché l'aggiunta di breakpoint automatici ed il target che, per gli scopi di questo elaborato, deve essere Core 0.

Dopo aver avviato il debug, il LED di stato dell'ICE-1000 passerà da verde a viola, indicandone la corretta esecuzione.

Nella nuova schermata di debug è quindi possibile, mediante il tasto "Resume", avanzare i breakpoint automatici e quindi caricare completamente l'algoritmo sulla board e testarne il funzionamento.

3.2.4 Booting

Il programma caricato sulla SHARC Audio Module durante il debugging è volatile. Scollegando e ricollegando l'alimentazione infatti, l'algoritmo non sarà più presente in memoria.

Il caricamento del programma sulla memoria flash permette di risolvere questo problema. La procedura inizia con la creazione di una LDR (*loader image*) tramite il debugger. L'uploading avviene eseguendo l'applicazione `cldp.exe` tramite terminale con il seguente comando:

```
cldp -proc ADSP-SC589 -emu ICE-1000 -core 1 -driver
"C:\Analog Devices\SAM_BareMetal_SDK-Rel2.1.2\extras\
flash-programmer\Supporting_Files\sam_dpia_Core1.dxe"
-core 1 -cmd prog -erase affected -file "<workspace
directory>\Debug\NomeFile_Core1.ldr"
```

[14]

Nuovi recenti sviluppi da parte di Analog Devices permettono l'inserimento di un bootloader all'interno della memoria flash della SHARC Audio Module che consentirebbe il caricamento del programma direttamente dalla porta USB Micro Type B senza l'uso dell'emulatore ICE-1000 [15].

La procedura è abbastanza lunga e richiede una serie di operazioni preliminari che esulano dalle finalità sperimentali e prototipali di questo elaborato.

Conclusioni

Lo sviluppo del multieffetto presentato in questo elaborato mostra come Faust sia un linguaggio che permette una facile e intuitiva scrittura di algoritmi, anche in chiave di implementazione su sistemi embedded. Quest'ultimo aspetto è ampiamente supportato da tools dedicati che ne permettono la compatibilità con numerose piattaforme. Il suo paradigma ad alto livello e la logica basata su schemi a blocchi lo classificano come una valida alternativa a C++. Tuttavia Faust, come la maggioranza dei linguaggi ad alto livello (pur essendo compilato e non interpretato), pecca di efficienza computazionale che ne può compromettere l'uso su sistemi integrati. L'algoritmo sviluppato infatti, se testato mediante l'IDE online, è ben funzionante e ricrea abbastanza fedelmente la risposta di un generico multieffetto commerciale. Testato su SHARC Audio Module invece, all'ascolto è chiaramente udibile una distorsione che è dovuta, con ogni probabilità, a MIPS overflow (superamento delle massime risorse computazionali). Tale situazione non viene risolta né abilitando l'opzione di ottimizzazione del compilatore, né riducendo la frequenza di campionamento alla minima adottabile (44.1 kHz) e né aumentando la dimensione del buffer al massimo supportato (128 campioni). Il motivo è anche da ricercarsi nella gestione dell'interfaccia di controllo che fa uso, come illustrato in 2.3.2, di funzioni che processano comunque i segnali ad esse in ingresso anche se non utilizzati, comportando uno spreco di risorse non indifferente. Una possibile soluzione è sicuramente quella di utilizzare entrambi i core per processare l'audio e nell'integrazione del codice Faust con codice C++ per ottimizzare le risorse a disposizione. L'interazione tra i due core è certamente interessante e possibili realizzazioni a basso livello possono essere oggetto di sviluppi futuri.

I problemi di compatibilità evidenziati nel Capitolo 3 mostrano come la procedura di setup iniziale e di porting dell'algoritmo non sia particolarmente rapida e richieda una discreta conoscenza dei sistemi Linux, il che è in contrasto l'intuitiva procedura di sviluppo di algoritmi con Faust.

Si è voluto mettere l'accento sulle problematiche riscontrate in quanto questo elaborato si pone anche come supporto per chiunque decida di realizzare algo-

Capitolo 3 Porting su SHARC Audio Module

ritmi tramite le piattaforme qui utilizzate e come incentivo agli sviluppatori di queste a continuare con la ricerca e lo sviluppo in tal senso.

Bibliografia

- [1] Tom Evans and Mary Anne Evans. *Guitars: Music, History, Construction and Players from the Renaissance to Rock*. Grosset & Dunlap, 1977.
- [2] Júlio Ribeiro Alves. *The History of the Guitar: Its Origins and Evolution*. Huntington, 2015.
- [3] Ian Lang. *Digital Guitar Effects Pedal*. 2018.
- [4] SHARC Audio Module (ADZS-SC589-MINI) - Documentazione ufficiale. <https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/sharc-audio-module.html>. Consultato: 12/11/2022.
- [5] SHARC Audio Module Audio Project Fin - Documentazione ufficiale. <https://wiki.analog.com/resources/tools-software/sharc-audio-module/hardware/audioproj-fin>. Consultato: 12/11/2022.
- [6] ICE-1000 - Documentazione ufficiale. <https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/emulators.html#eb-overview>. Consultato: 12/11/2022.
- [7] Faust - Documentazione ufficiale. <https://faustdoc.grame.fr/manual/introduction/>. Consultato: 13/11/2022.
- [8] Yann Orlarey, Dominique Fober, and Stephane Letz. Syntactical and Semantical Aspects of Faust. *Soft Computing*, 8:623–632, 09 2004.
- [9] Joshua D Reiss and Andrew McPherson. *Audio effects: theory, implementation and application*. CRC Press, 2014.
- [10] Udo Zölzer, Xavier Amatriain, Daniel Arfib, Jordi Bonada, Giovanni De Poli, Pierre Dutilleux, Gianpaolo Evangelista, Florian Keiler, Alex Loscos, Davide Rocchesso, et al. *DAFX-Digital audio effects*. John Wiley & Sons, 2002.

Bibliografia

- [11] Julius O. Smith. *Physical Audio Signal Processing*. W3K Publishing, 2010 edition.
- [12] Dattorro, Jon. Effect design, part 2: Delay line modulation and chorus. *Journal of the Audio engineering Society*, 45(10):764–788, 1997.
- [13] Alosi Amerigo. *Studio e sviluppo di algoritmi di elaborazione del segnale musicale su piattaforma HW SHARC Audio Module Evaluation Board* [tesi di laurea triennale]. Ancona: Università Politecnica delle Marche, 2022.
- [14] Creating a Boot Image (Dual-Core ADSP-SC5xx Processors). <https://wiki.analog.com/resources/tools-software/crosscore/cces/getting-started/boot-app-sc5xx>. Consultato: 02/12/2022.
- [15] Sharc Audio Module - Program Flash. <https://wiki.analog.com/resources/tools-software/sharc-audio-module/advanced-audio-projects/program-flash>. Consultato: 02/12/2022.