Elektrotehnički fakultet
Univerzitet u Banjoj Luci

DODAVANJE MUZIČKIH EFEKATA U AUDIO SIGNAL KORIŠTENJEM RAZVOJNOG OKRUŽENJA ADSP-21489

iz predmeta

SIGNALI ZA DIGITALNU OBRADU SIGNALA

Student:
Tamara Lekić 11113/19

Mentori:
prof. dr Mladen Knežić
prof. dr Mitar Simić
ma Vedran Jovanović
dipl. Inž. Damjan Prerad

Februar 2024. godine

1. Opis projektnog zadatka

U sklopu projektnog zadatka je potrebno realizovati sistem za dodavanje muzičkih efekata u audio signal korištenjem razvojnog okruženja ADSP-21489. Osnovni dijelovi realizacije se mogu sumirati u narednih par koraka:

- Generisanje ulaznih i izlaznih signala u Pythonu, korištenjem odgovarajućih biblioteka.
- Implementacija istih efekta na ADSP procesoru.
- Analiza performansi koja obuhvata izvršavanje optimizacije svih koraka kroz koje navedeni sistem prolazi, sa vođenjem računa o zauzeću memorijskih resursa i brzini izvršavanja. U izvještaju je potrebno priložiti rezultate za različite pristupe optimizaciji koda, te obrazložiti dobijene rezultate.

2. Izrada projektnog zadatka

2.1. Korišteni hardverski i softverski reusrsi

Hardverski resursi korišteni pri izradi projektnog zadatka su: ADSP-21489 razvojna platforma i personalni računar (x86-64 procesor sa Windows 11 OS). Korištena softverska okruženja su: Cross Core Embedded Studio (CCES u nastavku teksta) i Visual Studio (Code). Svi efekti su implementirani u C programskom jeziku i u Pythonu.

2.2. Realizacija funkcionalnosti

Realizacija u Python programskom jeziku se može podijeliti u sledeće korake:

1. Generisanje i čuvanje audio signala – generisan je audio signal koji sadrži četiri sinusna tona različitih frekvencija (200 Hz, 400 Hz, 500 Hz, i 700 Hz), svaki trajanja 0.5 sekundi, sa frekvencijom odmjeravanja od 11025 Hz. Zatim se kreira signal ***input_signal*** koji je dopunjen nulama do dužine tri sekunde pri datoj frekvenciji odmjeravanja (11025 * 3), što odgovara ukupnom broju odmjeraka. Dobijeni signal je zapisan u WAV datoteku pod nazivom *"input_signal.wav"* i može se reprodukovati kao audio zapis sa zadatom frekvencijom odmjeravanja.

2. Upis generisanog signala u header fajl - Funkcija ***create_c_header*** implementirana u Pythonu generiše C zaglavlje koje sadrži niz sa audio podacima, frekvenciju odmjeravanja i dužinu niza. U zaglavlju se definišu makro direktive za frekvenciju odmjeravanja i dužinu audio podataka, a zatim se audio podaci upisuju u niz ***audio_data*** kao niz float vrednosti, formatirani sa osam decimala i grupisani po 10 u liniji za bolju čitljivost.

3. Implementacija audio efekta – Za svaki od četiri implementirana efekta implementirane su funkcije koje vrše primjenu odgovarajućeg audio efekta nad ulaznim signalom. Funkcije su implementirane tako da se pri njihovom pokretanju čuva audio signal nad kojim je primjenjen efekat.

4. Učitavanje binarnog fajla i poređenje rezultata - Nakon što je implementiran isti efekat u C programskom jeziku, izvršen na ADSP-21489 razvojnoj ploči, dobijen je izlazni .bin fajl u kom se nalazi izlazni signal nad kojim je primijenjen efekat. Funkcijom ***read_from_C*** vršimo čitanje audio odmjeraka i vršimo konverziju iz bajtova u numpy niz.

Realizacija funkcionalnosti u C programskom jeziku se može podijeliti u sledeće korake:

1. Implementiranje funkcije za isti audio efekat kao u Pythonu.

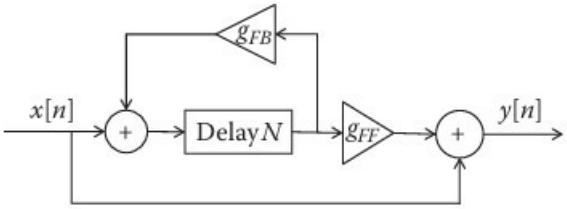
2. Upis izlaznog signal nad kojim je primijenjen efekat u binarni fajl.

3. Vršenje optimizacije i profilisanje koda

U sklopu ovog projektnog zadatka implementirana su četiri audio efekta (efekat kašnjenja sa povratnom spregom (eho), tremolo, flanger i wahwah). Ukratko će biti objašnjen svaki od ovih efekata u nastavku teksta.

Echo efekat

Efekat kašnjenja s povratnom petljom, prikazan na Slici 2.1, predstavlja audio proces u kom povratna petlja uzrokuje da se zvuk neprekidno ponavlja, i ako je pojačanje povratne sprege manje od 1, eho će postajati sve tiši. Teoretski se eho ponavlja zauvijek, ali s vremenom postaje toliko tih da je ustvari nečujan. Ulazni signal se odlaže za određeni broj odmjeraka N i zatim miješa s izvornim signalom, stvarajući početni eho. Povratna petlja zatim šalje dio izlaznog signala nazad na ulaz, stvarajući niz dodatnih eha. Kontrolom faktora pojačanja povratne petlje (označenog kao "g_{FB}" na Slici 2.1), moguće je regulisati koliko dugo će se eho ponavljati prije nego što postane nečujan.



Slika 2.1 - *Blok šema za implementaciju echo efekta*

Kašnjenje sa povratnom spregom umjesto skladištenja originalnog ulaznog signala u bafer, skladišti zbir ulaznog signala i zakašnjenog. Dakle, osnovni efekta kašnjenja se implementira bez povratne sprege i kod njega u bafer se upisuje originalni ulazni signal. Prikaza jednačina na osnovu koje je implementiran algoritam je dat sa (2.1), dok je implementacija u C programskom jeziku prikazana na Slici 2.2.

$$y(n)=x[n]+g_{FF}\,d[n] \text{ where } d[n]=x[n-N]+g_{FB}\,d[n-N] \tag{2.1}$$

```
for (int i = 0; i < signal_length; i++) {
    float input_sample = input_signal[i];
    int buffer_index = i % DELAY_SAMPLES;
    float delayed_sample = delayBuffer[buffer_index];

    if (i >= DELAY_SAMPLES) {
        delayBuffer[buffer_index] = input_sample + feedback_gain * delayed_sample;
        output_signal[i] = input_sample + feedforward_gain * delayed_sample;
    } else {
        output_signal[i] = input_sample;
    }
```

Slika 2.2 – *Implementacija efekta kašnjenja sa povratnom spregom u C-u*

Tremolo efekat

Tremolo je efekat koji djeluje na amplitudu audio signala, stvarajući ritmičke promjene u glasnoći koje percipiramo kao „drhtanje“. Osnovni princip tremolo efekta je množenje ulaznog signala sa periodičnim, polako promjenljivim signalom. U jednačini (2.2) to je signal m[n]. Dakle, ovaj efekat se dobija na osnovu formule (2.2) i možemo tumačiti da predstavlja promjenljivo pojačalo čije je pojačanje na signalu od n odmjeraka dato sa m[n].

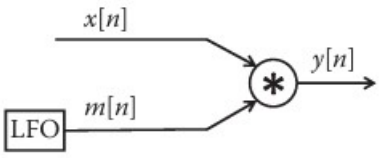
$$y[n]=m[n]x[n] \tag{2.2}$$

Kada je amplituda signala m[n] > 1, tada je amplituda izlaznog signala veća od amplitude ulaznog signala. Vrijedi i obrnuto, ako je amplituda signala m[n] < 1 tada je amplituda izlaznog signala manja od amplitude ulaznog signala. Zvuk tremolo efekta koji nastaje dolazi iz činjenice da je pojačanje periodično promijenljivo tokom vremena . Modulišući signal m[n] (2.3) se generiše pomoću oscilatora niskih

frekvencija (Low-Frequency Oscillator - LFO). α je dubina modulacije i $\omega_{LFO} = 2\pi f_{LFO}/fs$ učestanost gde je fs frekvencija odmjeravanja i f_{LFO} frekvencija oscilacije tremolo efekta, koja varira od 0.5 do 20 Hz.

$$m[n] = 1 + \alpha \sin(\omega_{LFO}) \tag{2.3}$$

Prikazana je implementacija ovog efekat u C programskom jeziku (Slika 2.4), dok je na Slici 2.3 prikazana struktura tremolo efekta. U ovoj konkretnoj implementaciji, za svaki odmjerak ulaznog signala, izračunava se novi odmjerak izlaznog signala tako što se originalni odmjerak ulaznog signala množi sa faktorom koji varira između 1 i 1 + alpha.



Slika 2.3 - *Prikaz strukture kojom se implementira tremolo efekat*

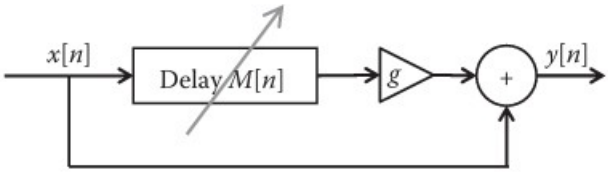
```
for (int i = 0; i < signal_length; i++) {
    output_signal[i] = (1 + alpha * sinf(2 * PI * Flfo * i / SAMPLE_RATE)) * input_signal[i];
}
```

Slika 2.4 - *Prikaz implementacije tremolo efekta u C-u*

Flanger efekat

Flanger efekat stvara karakterističnu "whoosh" zvuk. Osnovna implementacija flangera koristi varijabilnu dužinu kašnjenja koja se kontroliše pomoću niskofrekventnog oscilatora, a tipične dužine kašnjenja kreću se od 1 do 10 ms. Jednačina (2.4) je jednačina na osnovu koje je implementiran ovaj efekat, dok je na Slici 2.5 prikazana blok šema.

$$y[n] = x[n] + g * x[n - M[n]] \tag{2.4}$$



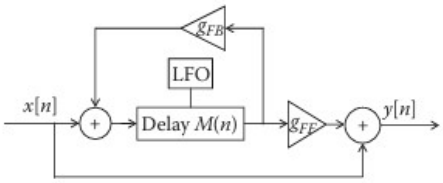
Slika 2.5 - Blok šema osnovnog flanger efekta

Na Slici 2.6 je implementacija u C-u. Osnovna implementacija flanger efekta u datom kodu radi na sledeći način: Izračunava se kašnjenje za svaki odmjerak koje je periodična funkcija niskofrekventnog oscilatora, stvarajući time promjenljivo kašnjenje koje osciluje. Zatim tu vrijednost kašnjenja koristimo da izvršimo linearnu interpolaciju između uzoraka ulaznog signala, stvarajući tako zakašnjeni signal. Zakašnjeni signal se dodaje na ulazni signal sa pojačanjem (gain), čime dobijamo zvuk flanger efekta. S obzirom da ovaj efekat zahtjeva kašnjenje koje se mijenja vremenom, da bi se postigla glatka promjena u kašnjenju korištena je linearna interpolacija, jer originalni signal nije bio čist.

```
const float omega = 2 * PI * Flfo / SAMPLE_RATE;
#pragma vector_for
for (int i = 0; i < signal_length; i++) {
    float d = DELAY_SAMPLES * 0.5 * (1 + sin(i * omega));
    int d_int = (int)d;
    float frac = d - d_int;

    if (i - d_int - 1 >= 0) {

        float delayed_sample = input_signal[i - d_int] * (1 - frac) + input_signal[i - d_int - 1] * frac;

        output_signal[i] = input_signal[i] + (gain * delayed_sample);
    } else {
        output_signal[i] = input_signal[i];
    }
}
```

Slika 2.6 - *Prikaz implementacije flanger efekta u C-u*

Flanger sa povratnom spregom (feedback) uvodi dodatni put povratne informacije koji pojačava izlaz kašnjenja nazad na njegov ulaz, dajući intenzivniji zvuk. Na Slici 2.7 prikazan je flanger sa povratnom spregom, dok je jednačina data sa (2.5). Možemo vidjeti da kada je pojačanje povratne veze g_{FB} jednako nuli, dobijamo osnovni flanger efekat bez povratne veze, što se i očekuje.



Slika 2.7 - *Blok šema flanger efekta sa povratnom informacijom*

$$y[n] = g_{FB}\, y[n - M[n]] + x[n] + (g_{FF} - g_{FB})\, x[n - M[n]] \qquad (2.5)$$

Na Slici 2.8 prikazana je implementacija.

```
float omega = (2 * PI * Flfo) / SAMPLE_RATE;
//#pragma vector_for
for (int i = 0; i < signal_length; i++) {
    float current_delay = (1 + sinf(omega * i)) / 2 * DELAY_SAMPLES;
    int delay_index = (int)current_delay;

    int buffer_index = i % (DELAY_SAMPLES + 1);
    float delayed_sample = delayBuffer[buffer_index];

    delayBuffer[buffer_index] = input_signal[i] + feedback_gain * delayed_sample;

    output_signal[i] = input_signal[i] + feedforward_gain * delayed_sample;
}
```

Slika 2.8 - *Implementacija flanger efekta sa povratnom vezom*

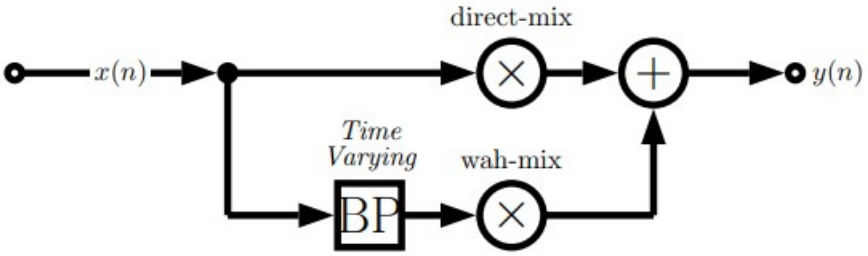
Kašnjenje se realizuje slično kao kod osnovnog flanger efekta, na osnovu periodične funkcije niskofrekventnog oscilatora. Parametar o*mega* predstavlja odnos između frekvencije niskofrekvencijskog oscilatora i frekvencije odmjeravanja. U sklopu ove implementacije korišten je bafer za kašnjenje *delayBuffer,* u kom se čuvaju odmjerci signala koji treba da budu zakašnjeni. Parametar *feedback_gain* kontroliše pojačanje u povratnoj vezi. Uzorak iz bafera se množi sa tim parametrom i dodaje na trenutni ulazni odmjerak, prije nego što se upiše u bafer. Parametar *feedforward_gain* kontroliše direktno dodavanje zakašnjenog odmjerka na izlazni signal. Kombinovanjem sa originalnim signalnom stvara se konačni flanger efekat sa povratnom spregom.

WahWah efekat

WahWah efekat je zvučni efekat koji proizvodi promjenu u tonu imitirajući zvučni efekat sličan ljudskom glasu koji govori "wah". Na Slici 2.9 je prikazana blok šema WahWah efekta. Osnovna implementacija WahWah efekta koristi promjenljivi filtar propusnik opsega, čija se centralna frekvencija mijenja. Da bi se postigao karakterističan zvuk WahWah efekta, filter se obično prilagođava da ima prilično oštru Q vrednost (faktor kvaliteta), što rezultuje užim i izraženijim frekvencijskim opsegom koji se pojačava ili smanjuje dok se filter pomiče kroz spektar. Jednačine na osnovu kojih implementiramo algoritam su date sa (2.6) za signale i (2.7) za koeficijente F1 koji zavisi od centralne frekvencije i Q1 koji zavisi od faktora prigušenja damp.

$$\begin{aligned} y_l(n) &= F_1 y_b(n) + y_l(n-1) \\ y_b(n) &= F_1 y_h(n) + y_b(n-1) \\ y_h(n) &= x(n) - y_l(n-1) - Q_1 y_b(n-1) \end{aligned} \tag{2.6}$$

$$F_1 = 2\sin\left(\frac{\pi f_c}{f_s}\right) \wedge Q_1 = 2\,damp \tag{2.7}$$



Slika 2.9 – *Blok šema wahwah efekta*

Sama implementacija algoritma u Pythonu i C-u ima par razlika. U Pythonu je algoritam implementiran na osnovu gore navedenih jednačina. Petlja izračunava F1, yl, yb i yh za svaki odmjerak. Centralna frekvencija Fc se prilagođava izvan petlje i generiše se kao niz, koji je skraćen na dužinu signala. Petlja u C-u uključuje logiku za prilagođavanje centralne frekvencije Fc sa svakom iteracijom, što znači da je promjena ove frekvencije integrisana unutar glavne petlje za obradu i Fc se povećava ili smanjuje unutar petlje. Provjera vrijednosti centralne frekvencije Fc za promjenu smijera se radi za svaku iteraciju. Na Slici 2.10 je prikazana implementacija WahWah efekta u C programskom jeziku.

```
float Q1 = 2 * DAMP;
float Fc = MINF;
float F1 =(float) 2 * sinf((PI * Fc) / SAMPLE_RATE);
float yl = F1 * input_signal[0];
float yb = output_signal[0] = F1 * input_signal[0];
float delta = WAHF / SAMPLE_RATE;
float maxy = fabs(output_signal[0]);

for (int i = 1; i < length; i++) {
    if (Fc + delta > MAXF || Fc + delta < MINF) {
        delta = -delta;
    }
    Fc += delta;
    F1 = 2 * sinf((PI * Fc) / SAMPLE_RATE);
    float yh = input_signal[i] - yl - Q1 * yb;
    output_signal[i] = F1 * yh + yb;
    yl = F1 * output_signal[i] + yl;
    yb = output_signal[i];
```

Slika 2.10 - *Implementacija WahWah efekta*

3. Rezultati i zaključak

Verifikacija funkcionalnosti implementiranih efekata sprovedena je na dva audio signala. Jedan je gore pomenuti generisani signal ***input_signal***, a drugi je učitani signal ***input*** iz fajla pod nazivom „acoustic.wav“ koji takođe ima frekvenciju odmjeravanja 11025Hz. Svi rezultati obrade su priloženi u sklopu foldera Python. Implementiranom funkcijom ***signal_error(audio_py, audio_C)*** u Pythonu napisana je da bismo uporedili dva audio signala (jedan generisan u Pythonu, drugi u C-u) i vizualizovali njihovu razliku u obliku greške. U nastavku će biti prikazane greške za svaki od implementiranih efekata samo za ulazni audio signal **input_signal** zbog sličnosti rezultata i za drugi testni signal.

Greška je izračunata kao apsolutna vrijednost razlike između svakog odmjerka u dva niza. Na x-osi su indeksi odmjeraka, dok y-osa pokazuje veličinu greške za svaki uzorak. Ako su dva signala identična, linija greške bi bila ravna linija na nuli, inače na grafu visina stubića ukazuje na veličinu greške za svaki odmjerak. Međutim, na rezultatima vidimo varijacije u vrijednostima grešaka, što pokazuje da postoje razlike između dva signala.

Ove razlike mogu biti posljedica:

- Numeričke razlike u implementaciji – različiti programski jezici i biblioteke mogu imati različite preciznosti i metode za izračunavanje matematičkih operacija, što može rezultirati malim numeričkim razlikama.

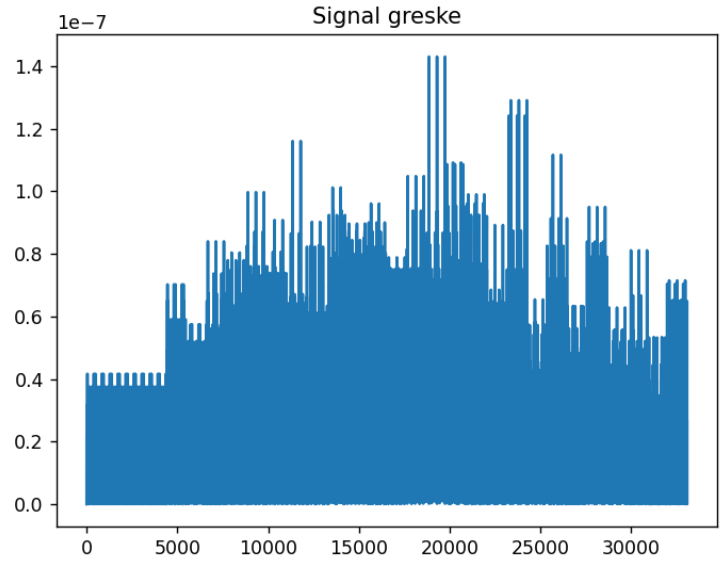
- Tipovi podataka i preciznost – Ako jedan jezik koristi 32-bitne floating point vrijednosti, a drugi 64-bitne, može doći do razlike zbog preciznosti.

- Razlike u algoritmu – Razlike u algoritamskoj logici ili redoslijedu operacija, koje mogu uticati na ukupnu grešku.

Na grafu, visina stubića ukazuje na veličinu greške za svaki odmjerak. Činjenica da greške nisu konstantno nula ukazuje da postoji razlika u izlaznim signalima. Ovi pikovi mogu biti mali i nečujni ljudskom uhu. Na osnovu rezultata za efekte kašnjenja, tremolo, flanger i djelimično wahwah može se smatrati da dvije implementacije proizvode veoma sličan rezultat. U slučaju rezultata za flanger efekat sa povratnom petljom dobijena je nešto veća greška, a razlog tome može biti nešto od gore navedenog.

```
signal_error(delay_with_feedback,delay_with_feedback_c)
✓ 0.3s
```

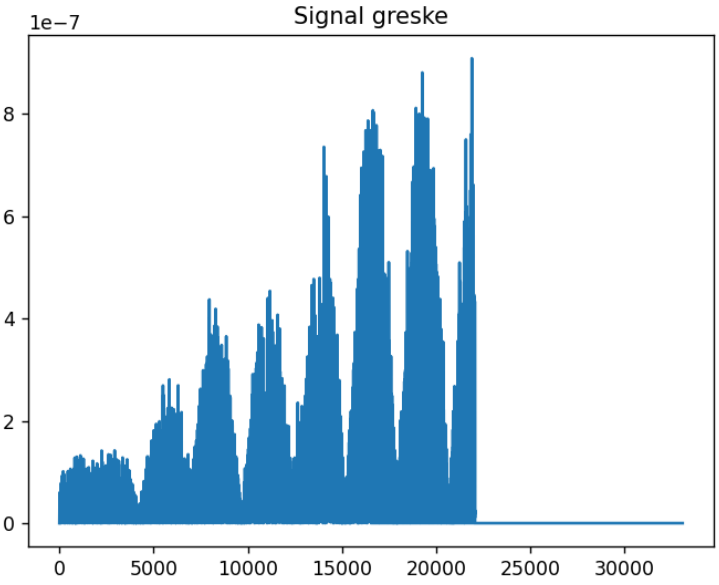
Figure 3



Slika 3.1 – *Prikaz greške za efekat kašnjenja sa povratno petljom*

```
signal_error(tremolo,tremolo_c)
✓ 0.2s
```

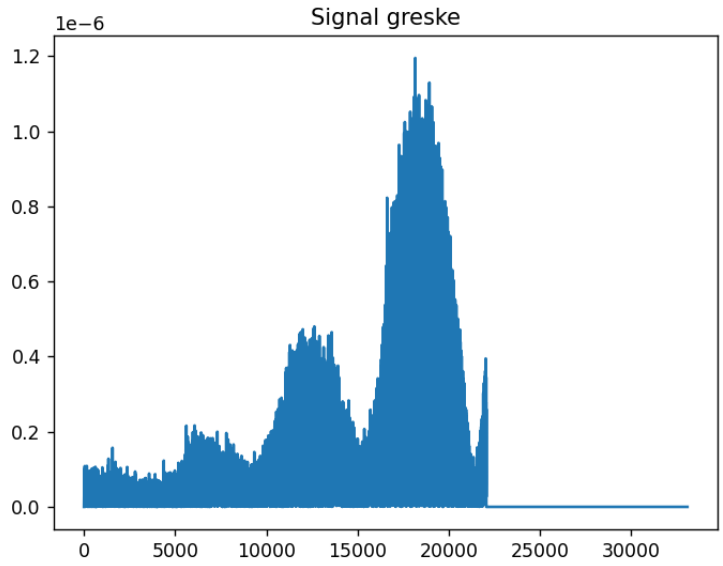
Figure 6



Slika 3.2 – *Prikaz greške za tremolo efekat*

```
signal_error(flanger,flanger_c)
✓ 0.3s
```

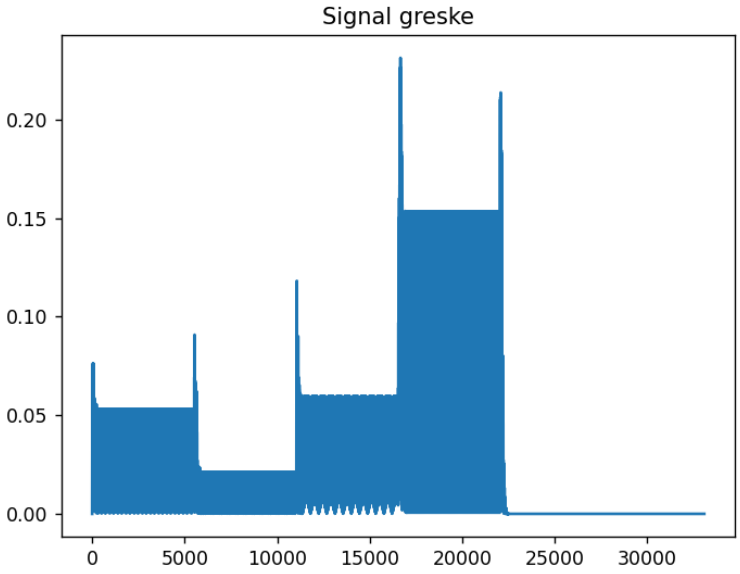
Figure 9



Slika 3.3 – *Prikaz greške za flanger efekat*

```
signal_error(flanger_feedback,flanger_feedback_c)
✓ 0.3s
```

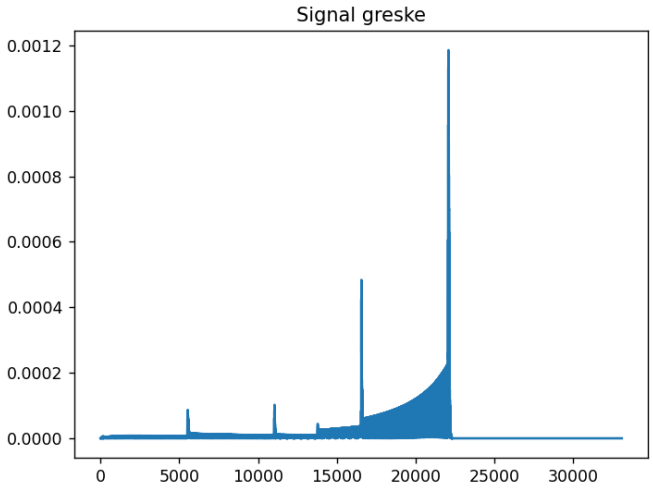
Figure 12



Slika 3.4 – *Prikaz greške za flanger efekat sa povratnom spregom*

```
signal_error(wahwah,wahwah_c)
✓ 0.1s
```

Figure 15



Slika 3.5 – *Prikaz greške za wahwah efekat*

Zauzeće memorije

ADSP-21489 ima internu statičku memoriju (SRAM) za instrukcije i podatke koja je podijeljena u 4 bloka. Na razvojnoj ploči ADSP-21489 postoji eksterna memorija sa kojom je procesor povezan preko eksternog porta. Prilikom izrade zadatka nizovi korišteni za skladištenje izlaznih podataka pokušano je da budu statički definisani. Međutim, pri pokušaju debug-ovanja projekta javile su se greške koje su ukazale da je došlo do preteka memorijskih blokova koji se mogu identifikovati u okviru projekta u CCES u fajlu /system/startup_ldf/app.ldf . Kako bi se riješio problem i obezbijedila dovoljna količina memorije za skladištenje izlaznih signala definisana je korisnička memorijska sekcija seg_sram u eksternoj SRAM memoriji dodavanjem sljedećeg koda. Prikazano na Slici 3.6.

```
dxe_seg_sram{
  INPUT_SECTIONS($OBJECTS(seg_sram))
} > mem_sram
```

Slika 3.6 – *Definisanje sekcije seg_sram*

Zauzeće procesorskih resursa i optimizacija

U Tabeli 3.1 su dati svi efekti implementirani u okviru projektnog zadatka i broj procesorskih ciklusa potrebnih za njihovo izvršavanje bez uključene optimizacije i korištenjem različitih tehnika optimizacije. Rezultati su dobijeni na testnom signalu *input_signal* čiji su odmjerci čitani iz header fajla *audio_data.h*. Procesorski ciklusi su izmjereni korištenjem funkcija iz *cycle_count.h* zaglavlja. U prvoj koloni tabele su rezultati dobijeni bez optimizacije. U drugoj koloni su rezultati dobijeni uključivanjem optimizacije kompajlera po brzini. U trećoj koloni su rezultati dobijeni sa uključenom optimizacijom kompajlera po brzini i primijenjenom pretprocesorske direktive #pragma vector_for. #pragma vector_for daje naredbu kompajleru da vektorizuje petlju čime je omogućeno paralelno izvršavanje više od jedne iteracije u petlji.

Tabela 3.1 - *Broj ciklusa potrebnih za izvršavanjae algoritama uz različite tehnike optimizacije*

EFEKAT	Bez optimizacije	Kompajlerska optimizacija	#pragma vector_for
DELAY_WITH_FEEDBACK	6656203	2329151	2329151
TREMOLO	7212601	4479066	2825398
FLANGER	7807579	4809650	4809647
FLANGER_WITH_FEEDBACK	8732380	2274074	1833112
WAHWAH	10187304	6118908	4878652

U Tabeli 3.2 je procentualno prikazana optimizacija. Kao što vidimo za algoritme dela

Tabela 3.2 - *Procentualni prikaz optimizacije*

EFEKAT	Bez optimizacije / Kompajlerska optimizacija (%)	Kompajlerska optimizacija/ kompajlerska optimizacija i #pragma vector_for (%)	Bez optimizacije/#pragma vector_for (%)
DELAY_WITH_FEEDBACK	65.01	0	65.01
TREMOLO	37.9	36.92	60.83
FLANGER	38.4	0.00006	38.39
FLANGER_WITH_FEEDBACK	73.96	19.39	79.01
WAHWAH	39.94	20.27	52.11

Na osnovu podataka iz priloženih tabela, vidi se da kompajlerska optimizacija i upotreba vektorskih instrukcija značajno smanjuju broj ciklusa potrebnih za izvršavanje, što ukazuje na poboljšanje performansi. U pogledu kompajlerske optimizacije posebno je vidljivo poboljšanje za efekat kašnjenja, gdje optimizacija rezultira smanjenjem broja ciklusa za 65%. Dodatna optimizacija upotrebom #pragma_for pruža dodatno

poboljšanje performansi za neke algoritme. Na primjer, tremolo i wahwah efekti pokazuju značajno smanjenje u broju ciklusa kada kombinujemo kompajlersku optimizaciju sa vektorskim instrukcijama. U slučaju efekta za kašnjenje sa povratnom petljom i flanger efekta sa povratnom petljom vidimo da nema poboljšanja sa vektorizacijom. Razlog ovome je što između iteracija petlje postoji zavisnost podataka gdje trenutna iteracija zavisi od prethodne i to onemogućava vektorizaciju jer ona zahtijeva da se operacije nad više elemenata mogu izvršiti nezavisno i paralelno. Pored toga uslovni iskazi nisu prijateljski u pogledu vektorizacije, posebno ako uslov zavisi od indeksa petlje. Pokušano je ručno odmotavanje petlje u slučaju efekta flanger sa povratnom petljom, međutim to je dalo veoma loše rezultate tj. pogoršanje.

Pri izradi, prvobitno efekati koji koriste povratnu petlju (flanger i efekat kašnjenja) bafer je bio implementiran korištenjem dinamičke alokacije memorije. Poboljšanje performansi u ovom slučaju je dobijeno korištenjem statički alociranog bafera. Statička alokacija memorije se obavlja u vrijeme kompajliranja, pa se memorija za bafer dodjeljuje prije pokretanja programa što donosi predvidivost u pogledu upotrebe memorije i bržeg pristupa memoriji. U pogledu performansi pristupa memoriji i brzini dodatno poboljašnje bi se postiglo na primjer inicijalizacijom bafera čija se veličina definiše, jer sam pristup eksternoj memoriji je sporiji od interne memorije.

Dodatno je implementiran prikaz progresa obrade korištenjem dioda na ploči. Na početku izvršavanja glavnog programa sve LED diode se isključuju, zatim se uključuje prva dioda, a zatim nakon završetka svakog od efekata se uključuje po jedna dioda. Pored toga je implementirano kaskadno i paralelno vezivanje više efekata. Kod kaskadnog vezivanja audio signal prolazi kroz jedan efekat, a zatim izlaz tog efekta postaje ulaz u sledeći efekat. Prednost ovakvog kombinovanja efekata je što svaki efekat može modifikovati signal na osnovu promjena koje su napravili prethodni efekti i precizno možemo kontrolisati redoslijed u kojem se efekti primjenjuju. Nedostatak je akumulacija buke i distorzije sa svakim dodatnim efektom. U slučaju paralelnog vezivanja audio signal se dijeli na više putanja, a zatim se procesuirani signali ponovo kombinuju (miksaju) prije izlaska. Prednosti su što se efekti primjenjuju na originalni signal, kao i fleksibilnost jer svaki efekat radi nezavisno jedan od drugog. Nedostatak je što zahtijeva više resursa, jer svaki lanac u vezi može nezavisno obrađivati signal.

Moguća poboljšanja obuhvata dalja optimizacija koda za smanjenje potrošnje memorije i upotrebe procesorskih resursa.

4. Literatura

1. Joshua D. Reiss, Andrew P. McPherson, *Audio Effects Theory, Implementation and Application*, Taylor & Francis Group, 2015.

2. Udo Zölzer, *Digital Audio Signal Processing Second Edition*, John Wiley & Sons, 2008.

3. Vladimir Risojević, *Multimedijalni sistemi*, Elektrotehnički fakultet Univerziteta u Banjoj Luci, 2018.

4. Materijali dostupni na moodle stranicama predmeta Osnovi digitalne obrade signala i Sistemi za digitalnu obradu signala

5. CCES C/C++ Compiler Manual for SHARC Processors: https://www.analog.com/media/en/dsp-documentation/software-manuals/ccessharccompiler-manual.pdf

6. ADSP-21489 procesor:https://www.analog.com/media/en/dsp-documentation/processor-manuals/ADSP214xx_HRM_rev0.3.pdf