Elektrotehnički fakultet
Univerzitet u Banjoj Luci

IZVJEŠTAJ PROJEKTNOG ZADATKA

iz predmeta

SISTEMI ZA DIGITALNU OBRADU SIGNALA

Student:
Emanuela Buganik 1209/18

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

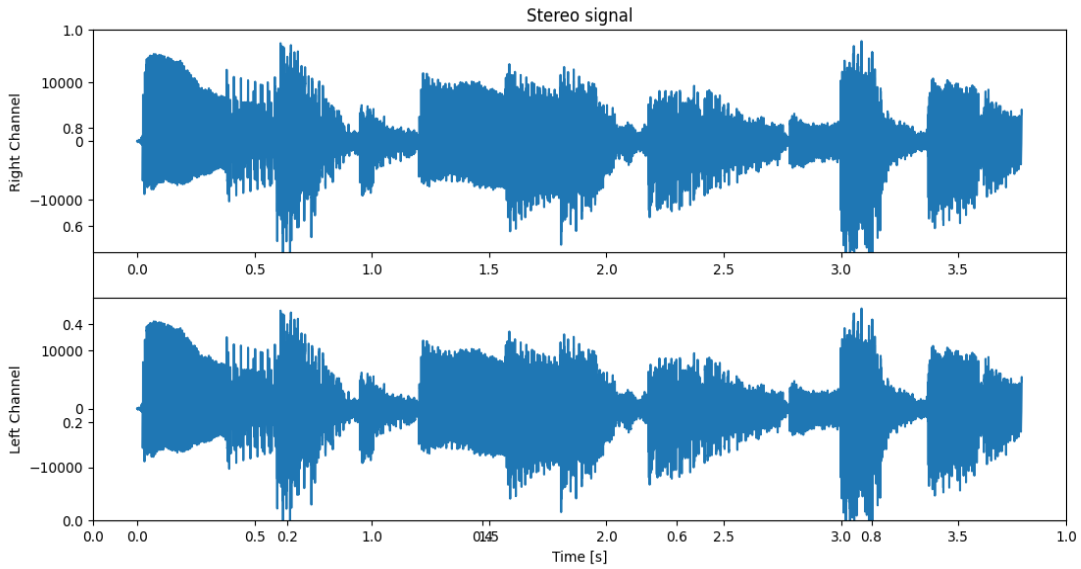
Februar 2024. godine

1. Opis projektnog zadatka

Cilj izrade projektnog zadatka je implementacija muzičkih efekata koji se mogu primijeniti na audio signal, korištenjem razvojnog okruženja ADSP-21489. Prvi dio izrade, ujedno i referenca za dalji rad odnosi se na generisanje referentnog signala i implementaciju gitarskih efekata (iz grupa definisanih projektnim zadatkom ili proizvoljno), a u Python programskom jeziku. Drugi dio izrade odnosi se na implementaciju istih efekata, ali u C programskom jeziku i unutar *CrossCore Embedded Studio* programskog paketa za ciljanu platformu. Odmjerci signala koji je korišten za analizu u prvom dijelu izrade upisuju se u file koji se zatim učitava u CCES kao *header* file i omogućava obradu istih. Moguće je reprodukovati rezultate obrade signala i u jednom i u drugom slučaju, eksportovati kao *.wav* file, a vršeno je i profilisanje koda s ciljem da se uoče potencijalni *bottleneck*-ovi i ukoliko je moguće, izvrši optimizacija koda na DSP-u. Efekti o kojima će biti riječ u narednom poglavlju su efekti iz prve i druge grupe - manje zahtijevni filtri *Delay* i *Echo* te umjereno zahtijevni *Tremolo*, *Flanger* i *Bit Crusher*.

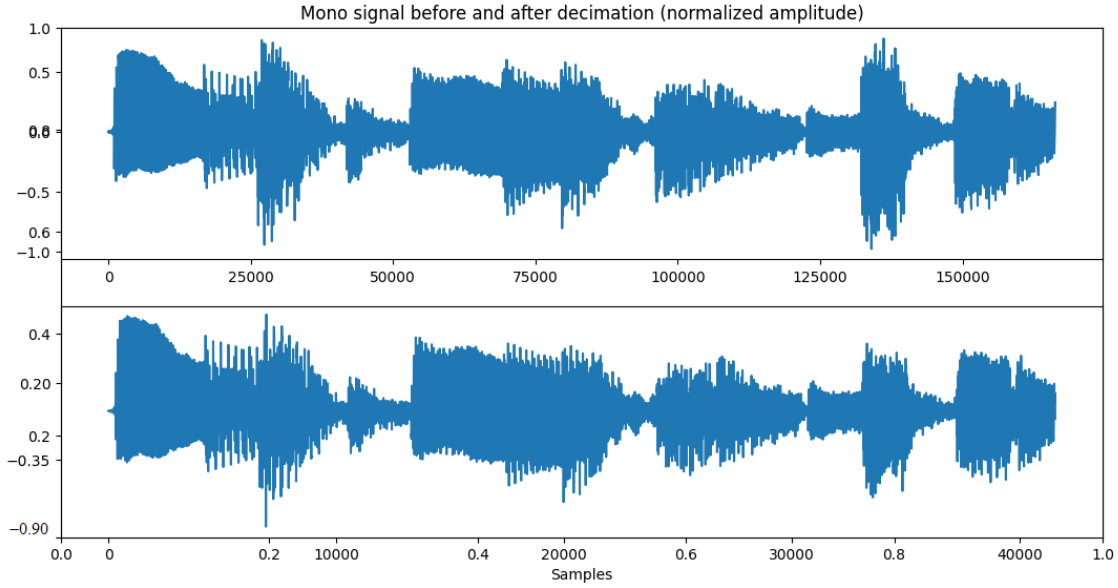
2. Izrada projektnog zadatka

S obzirom na primjenu efekata koje je potrebno implementirati, kao audio signal koji će se koristiti za obradu u daljem radu koristi se stereo zvuk gitare u trajanju od 3,77 s, a čija je frekvencija odmjeravanja prvobitno 44 100 Hz. Jedna od opcija da se analizira stereo signal jeste izdvajanje kanala, primjena efekata, a zatim ponovno objedinjavanje u 2D niz. Međutim, kako je prikazano i na narednoj *Slici 2.1*



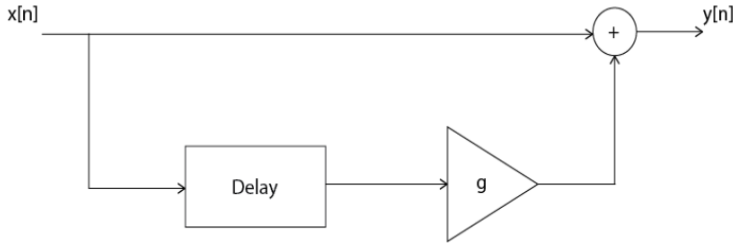
Slika 2.1 Stereo signal

da se zapaziti kako uglavnom nema razlike između lijevog i desnog kanala, te se postupak daljeg rada pojednostavljuje i tako radi sa mono signalom koji je nastao usrednjavanjem kanala stereo signala. Pored ovog razmatranja, u dalji rad se polazi idući koji korak unaprijed, pa se razmatra i kako uvesti odmjerke ovog sada mono signala u CCES, imajući na umu da se radi o signalu sa pomenutom frekvencijom odmjeravanja, gdje je broj odmjeraka 166 267. Postavlja se pitanje ima li smisla raditi sa velikim brojem odmjeraka kada radimo sa ograničenim signalom, uzimajući u obzir i da je potrebno omogućiti i učitavanje odmjeraka (pitanje skalabilnosti unutar *header*-a?), smještanje u memoriju, ispis obrađenih odmjeraka u *file* i slično. Sljedeći metod koji se može iskoristiti da pojednostavimo rad jeste *decimacija* (eng. downsampling), kojom za neki cjelobrojni faktor D možemo da smanjimo kako frekvenciju odmjeravanja nekog već odmjerenog signala, tako i njegov broj odmjeraka. U ovom slučaju izvršena je decimacija sa faktorom četiri, čime je nova frekvencija odmjeravanja 11 025 Hz, a broj odmjeraka 41 567. Korištena je python funkcija *decimate* (u okviru biblioteke *SciPy*), a prethodno je spesifikovan i niskopropusnik u vidu *Hann*-ovog prozora. Naredna slika prikazuje promjenu, tačnije mono signal prije i nakon decimacije.



Slika 2.2 Mono signal prije i nakon decimacije

***Delay* efekat**, kao prvi efekat koji je obrađen realizuje se korištenjem FIR filtra, kako se efekat zakašnjenog signala implementira sabiranjem originalnog signala sa svojom pomjerenom, zakašnjenom verzijom, koja je pomnožena faktorom g, kako je prikazano i na blok dijagramu na *Slici 2.3*. Naivna implementacija u Pythonu prikazuje da se pomjerena verzija signala može ostvariti pomjeranjem originalnog signala za broj odmjeraka koji određuju kašnjenje.



Slika 2.3 Blok dijagram *Delay* efekta

U ovom pa i narednim primjerima realizacije efekata, vrijednost faktora g je između -1 i 1, broj odmjeraka za pomjeraj (n_0) određuje se kao umnožak frekvencije odmjeravanja i vremena kašnjenja u sekundama.

$$y[n] = x[n] + \ g * x[n - n_0] \tag{2.1}$$

S obzirom na to da će nam Python poslužiti kao referenca, za implementaciju *Delay* efekta u C-u neophodna su nam dva niza – pomoćni koji predstavlja zakašnjenu verziju originalnog signala te niz koji predstavlja rezultat obrade. Iako je realizacija ovog projektnog zadatka svedena na odmjerke konačnog broja, pritom i redukovane prethodno pomenutim postupkom decimacije, nizove odmjeraka koje ćemo koristiti je svakako potrebno „pripisati“ nekoj memoriji zbog zauzeća koje interna memorija ne može podnijeti. Za početak analize, pomenuti nizovi su definisani unutar korisnički kreirane sekcije, koja pripada SRAM.

```
#pragma section("seg_sram")
float32_t audio_w_effect[NUM_SAMPLES];
#pragma section("seg_sram")
float32_t temp_array[NUM_SAMPLES];
```

```
  dxe_seg_sram
{
  INPUT_SECTIONS($OBJECTS(seg_sram))

} > mem_sram
```

Slika 2.4 Prikaz smještanja nizova u korisnički definisanu sekciju (lijeva slika) te definisanje iste u system/startup_ldf/app.ldf (desna slika)

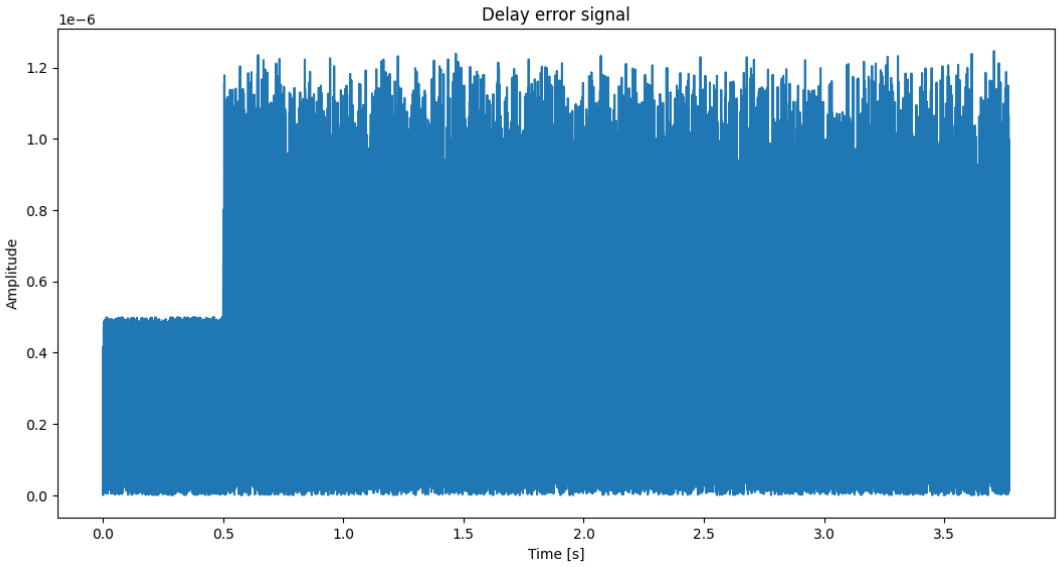
NUM_SAMPLES je makro unutar *audio.h* file-a (u kojem su eksportovani odmjerci) i definiše broj odmjeraka zvuka gitare, a nad kojim će se vršiti testiranje algoritama. Tip podataka sa kojim očekujemo raditi je float, kako bismo prilikom operacija nad odmjercima ostvarili što veću preciznost i rezultate koji će biti približni onom dobijenom u *Pythonu*, iako, već smo na samom početku radili sa zaokruživanjem na šest decimala što će se u nastavku rada dovesti u pitanje.

Konkatenaciju koja je sprovedena u Python kodu u C kodu zamijenjena je upisivanjem odmjeraka u pomoćni niz od pozicije *delay_samples* do NUM_SAMPLES, kako bismo osigurali da se nad nizovima u nastavku može odvijati operacija sabiranja i skaliranja, te kako bi konačan rezultat bio iste dužine kao i ulazni niz.

```
void delay_effect(const float32_t* restrict audio_input, float32_t* restrict audio_output, float32_t delay_s, float32_t gain)
{

    int delay_samples = conv_fix(delay_s*SAMPLE_RATE);
    int i,k;

    for(i = delay_samples; i<NUM_SAMPLES ; i++)
  {
      temp_array[i] = audio_input[i-delay_samples];
  }

  for(k=0; k<NUM_SAMPLES; k++)
  {
      audio_output[k] = audio_input[k]+gain*temp_array[k];
  }
}
```

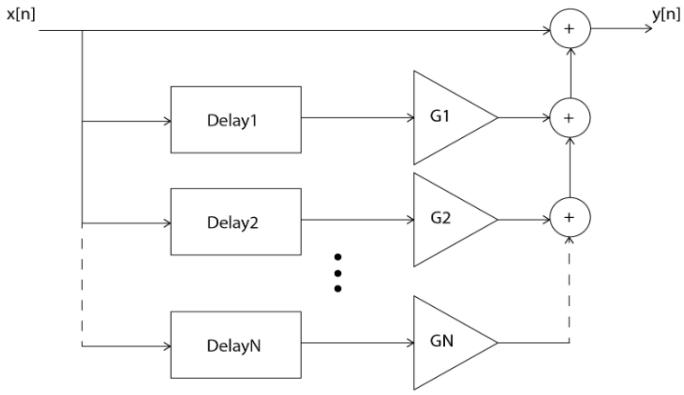
Slika 2.5 Implementacija *Delay* efekta u C kodu

Kao što je učinjeno i u Pythonu, svi argumenti funkcije, osim frekvencije odmjeravanja (koja je navedena kao makro) prosljeđuju se kao float vrijednosti. U C-u stavljamo naglasak na to da je u pitanju doista 32-bitna vrijednost. Na slici 2.5 da se primijetiti upotreba *intrinsic* funkcije ili ugrađene funkcije, koja omogućava efikasnije iskorištavanje hardverskih resursa, u ovom slučaju za dobijanje broja odmjeraka za koje se vrši pomjeranje. S obzirom na korišćenje pokazivača na nizove tipa *float*, u ovoj i narednim funkcijama koristi se i kvalifikator *restrict* kod funkcijskih parametara, kako bi pomogao kompajleru u razrješavanju potencijalnog pristupanja istim memorijskim lokacijama. Kroz dvije jednostavne *for* petlje vrši se prvo dobijanje zakašnjenih odmjeraka, a zatim skaliranje i dodavanje polaznom signalu. Ovakva, naivna realizacija funkcije iziskuje čak 4 351 940 ciklusa izvršavanja, a kada bi se se petlje sažimale u jednu, s obzirom na to da je krajnja vrijednost NUM_SAMPLES, svega 4 156 747. O potencijalnim poboljšanjima i primijenjenim tehnikama za smanjenje broja ciklusa biće više rečeno na kraju ovog poglavlja na temu svih audio efekata.



Slika 2.6 Signal greške kod *Delay* efekta

Sljedeći efekat koji možemo nadovezati na Delay jeste ***Echo efekat***, višestruki efekat kašnjenja koji se može realizovati kroz kašnjenje na više linija ili *stage*s, kako je prikazano na narednoj slici:



Slika 2.7 Blok dijagram *Echo* efekta

Kako se da i primijetiti, *Echo* efekat se sastoji od dodavanja originalnog audio signala N broju zakašnjenih verzija istog, gdje faktor skaliranja (G_i, gdje je i = 1, 2, 3, …) opada sa svakom linijom kašnjenja. Očito je da je za slučaj N=1, u pitanju gorepomenuti slučaj *Delay* efekta.

$$y[n] = x[n] + \sum_{i=1}^{N} G_i * x[n - n_0] \qquad (2.2)$$

Ovo znači da za npr. N=3, vrijednost izlaza zavisi od vrijednosti kašnjenja na izlazima na linijama kašnjenja, tj. prethodnim vrijednostima izlaza koji se superponiraju.

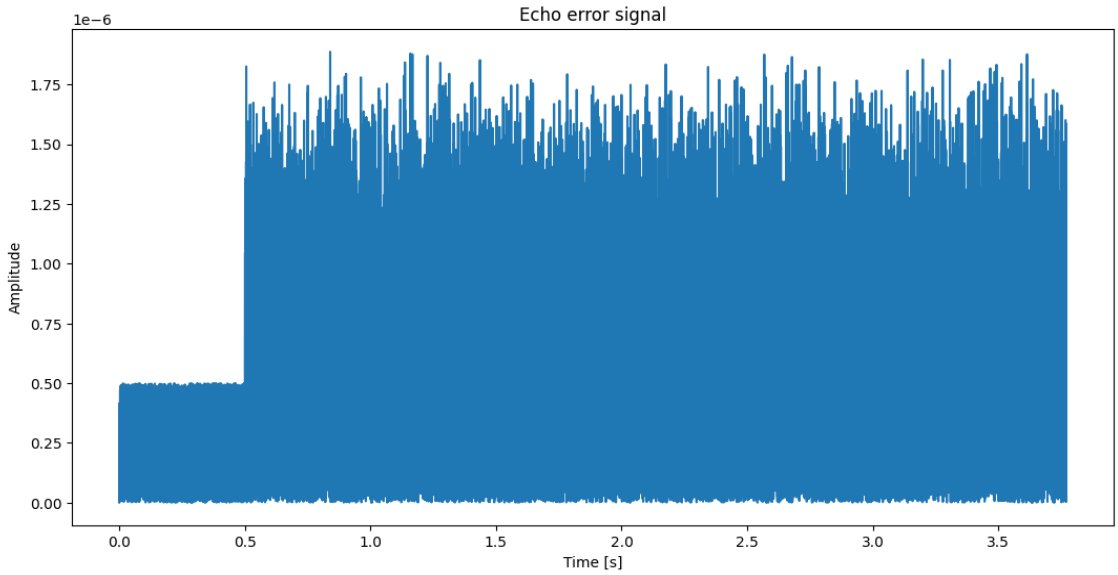
Dio koda na slici 2.8 je ujedno i dio funkcije *Echo* koji iziskuje najviše izvršavanja, što je na neki način i očekivano i sa povećanjem broja linija kašnjenja i broj ciklusa potencijalno raste, a neophodno je proći kroz sve odmjerke signala i sabrati ih sa zakašnjenom verzijom.

```
__builtin_assert(stages> 1); /*for stages = 1, it's better to call delay function*/

for(i=1; i<stages+1; i++)
    {
        stage_amp = (float32_t)(gain/i);

        for(k=0; k<NUM_SAMPLES; k++)
            {
            audio_output[k]+=stage_amp*temp_array[k];
            }
    }
```

Slika 2.8 Efekat kašnjenja na linijama kašnjenja u C-u

Iako se pokušaj optimizovanja ovog dijela koda pokazao kao zahtjevniji i kao što će rezultati kasnije da prikažu gdje jedino kompajlerska optimizacija ima mogućnost da značajnije smanji broj ciklusa izvršavanja, ovaj kod prikazuje dobru praksu koju navodi i *Manual* za Sharc kompajler, a to je da u slučaju ugniježđenih petlji unutrašnja uvijek ima više ciklusa izvršavanja u odnosu na spoljašnju, jer je očekivano da će ondje da se provede najviše vremena. Varijanta u prethodnim verzijama projekta da se obrne redoslijed dovela je do velikih povećanja u broju ciklusa.

__builtin_assert je funkcija koja može znatno pomoći kompajleru u radu, te mu tako pruža informaciju o vrijednosti parametara koji se prosljeđuju u funkciju, što na primjer on sam ne može da zaključi. Tako jei u ovom slučaju očekivano da vrijednost parametra *stages* bude veća od jedan, jer bi u suprotnom bio slučaj *Delay* efekta i nema smisla pozivati ovu funkciju za izvršavanje.

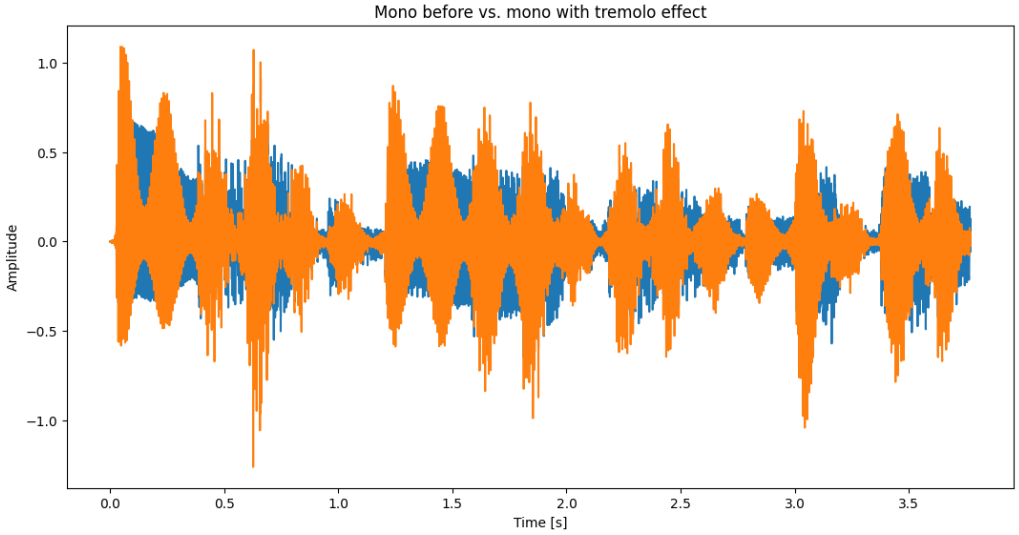


Slika 2.9 Signal greške kod *Echo* efekta

Tremolo efekat iz druge grupe efekata realizuje se kao vid amplitudske modulacije, što znači da se amplituda ulaznog signala (eng. *carrier*) mijenja sa promjenama LFO (oscilatora na niskim frekvencijama). Narednom jednačinom to se može prikazati na sljedeći način:

$$y[n] = x[n] * m[n] \tag{2.3}$$

Gdje je $m[n]$ određen sa $1 + \text{depth} * \sin(2\pi\frac{f_{LFO}}{fs}\text{n})$



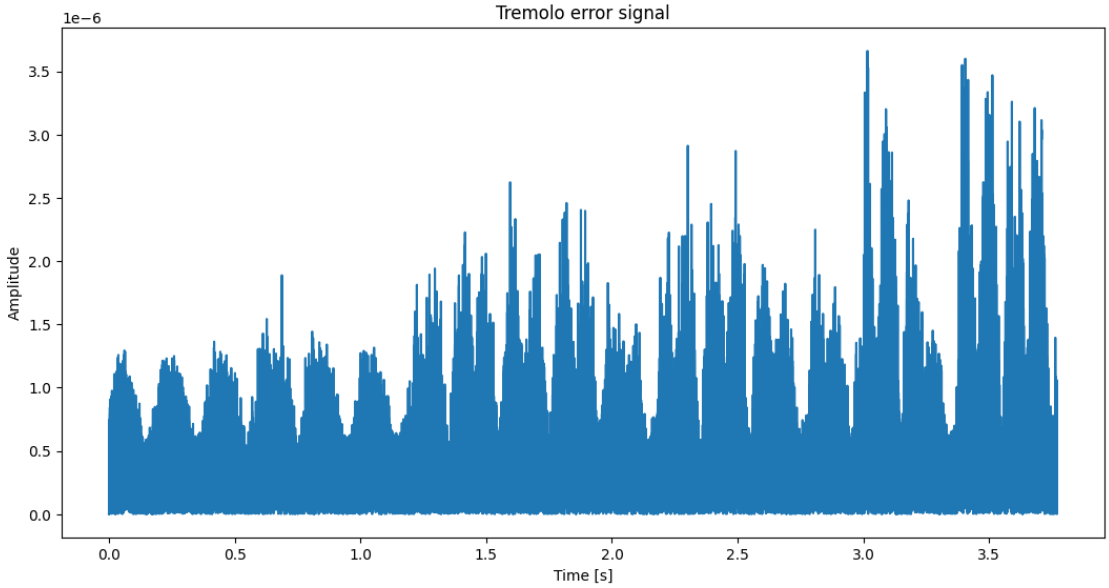
Slika 2.10 Signal prije i nakon primjene *Tremolo* efekta u Pythonu

Depth određuje amplitudu signala modulatora, uopšte – dubinu amplitudske modulacije, a f_{LFO} frekvenciju modulatora (obično u opsegu od 1 Hz do 25 Hz). Vidimo da se za prvobitnu realizaciju *Tremolo* efekta na osnovu jednačine (2.3) vrlo jednostavno može implementirati *Tremolo* efekat:

```
for(i=0; i<NUM_SAMPLES; i++)
{
    audio_output[i] = audio_data[i]*(1+depth*sinf(2.0*PI*i*rate/SAMPLE_RATE));
}
```

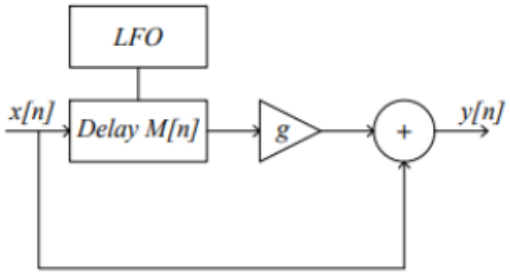
Slika 2.11 Amplitudska modulacija unutar *Tremolo* funkcije u C kodu

O potencijalnim poboljšanjima ovog algoritma biće priče u nastavku, a kao stavke koje će se svakako dodatno ispitati jesu korištenje *inline* funkcija ili definisanje funkcija kao makroi, s obzirom na to se da se izračunavanje vrši za svaki odmjerak u nizu audio_data (veliki broj poziva za kraće implementacije funkcija). Na slici 2.9 prikazan je signal greške obrađenog signala u Pythonu i signala na ADSP procesoru.

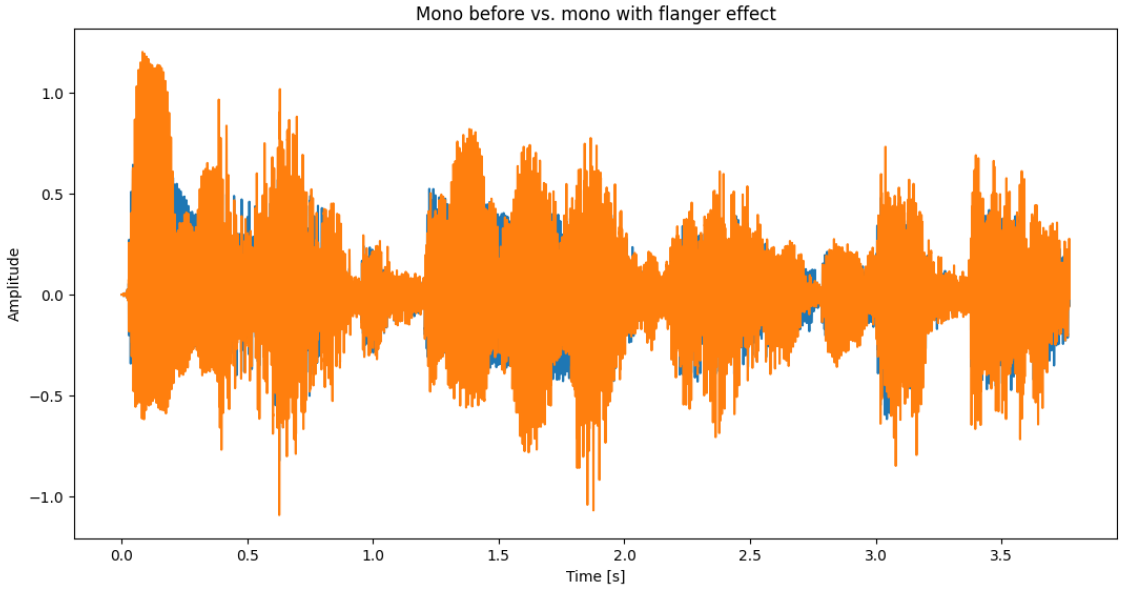


Slika 2.12 Signal greške *Tremolo* efekta

Osnovna ideja iza ***Flanger*** efekta je stvaranje konstruktivne i destruktivne interferencije, dodavanjem originalnog zvuka zakašnjeloj verziji zvuka, čije je kašnjenje određeno funkcijom vremena. Pomenuta funkcija vremena je u ovom slučaju LFO, kao u primjeru realizacije efekta *Tremolo*, a čija frekvencija modulatora iznosi između 0.1 Hz do 10 Hz.



Slika 2.13 Blok dijagram *Flanger* efekta



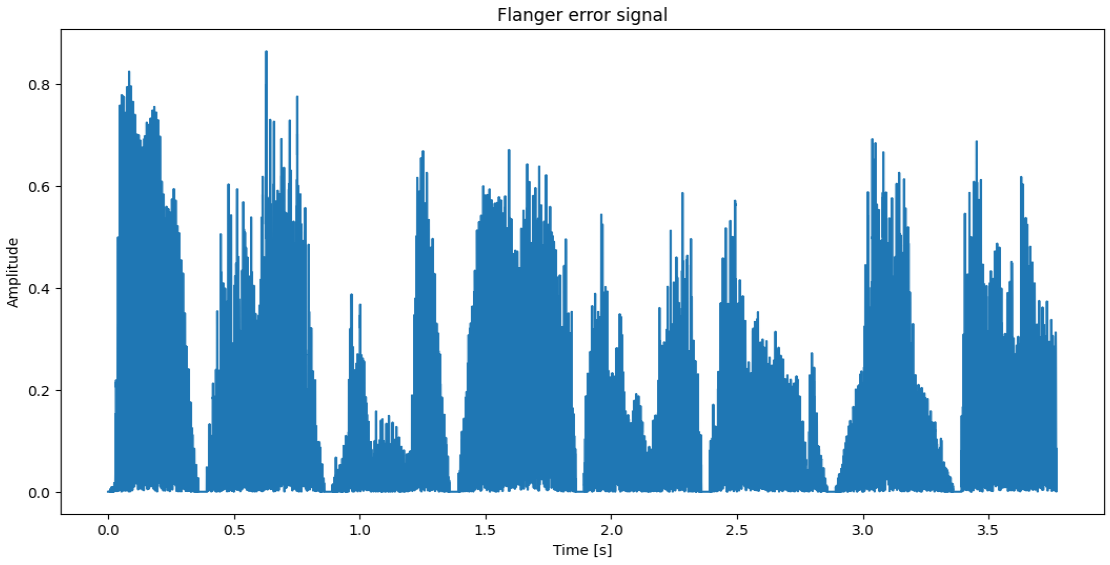
Slika 2.14 Signal prije i nakon primjene *Flanger* efekta u Pythonu

A, kako je realizovano kašnjenje u funkciji vremena može se vidjeti na slici 2.15 koja predstavlja dio implementacije optimizovane metode u C kodu.

```
    for(i = 0 ; i<NUM_SAMPLES; i++)
{
        d = conv_fix(delay_samples*(1 + sinf(lfo_w*i)));
        if(expected_true(i-d > 0))
        {
            audio_output[i]= audio_data[i]+ gain*audio_data[i-d];
        }
        else
        {
            audio_output[i] = audio_data[i];
        }
    }
```

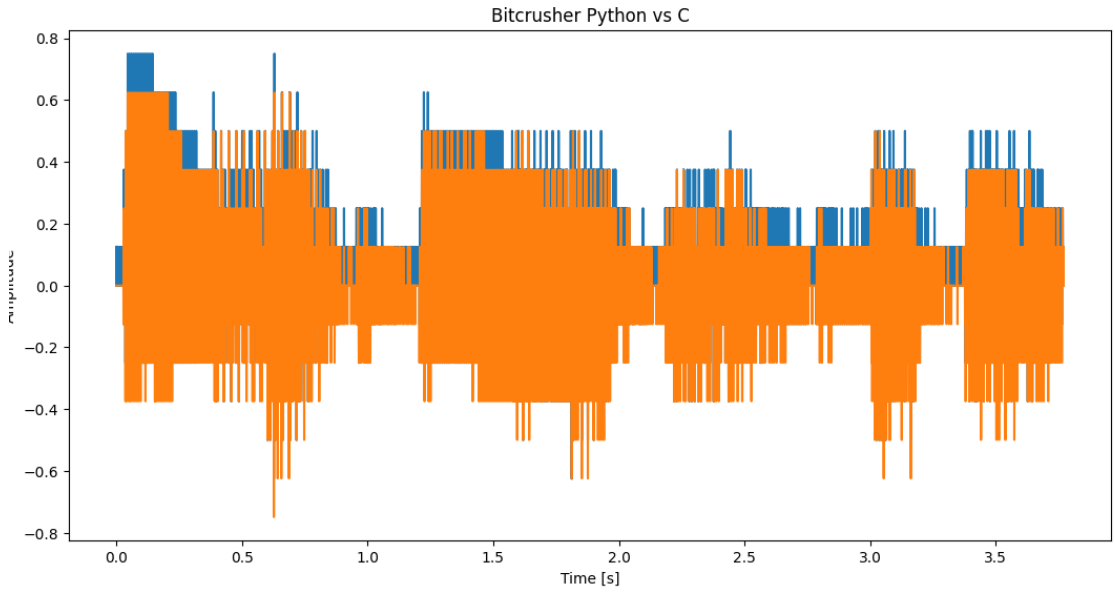
Slika 2.15 Implementacija funkcije *Flanger*

S obzirom na korišteno uslovno grananje unutar petlje, savjetuje se korištenje funkcija *expected_true (ili expected_false),* kojima se omogućava kompajleru da predvidi rezultat uslova i na taj način ubrza izvršavanje koje nameće grananje. Ovdje je potreba za poređenjem vrijednosti odmjerka za koji se vrši pomjeranje u odnosu na trenutnu vrijednost iteratora. Implementacija LFO je kao kod *Tremolo* efekta, ali ovog puta vrši modulisanje vrijednosti prethodno proračunatog broja odmjeraka za kašnjenje.



Slika 2.16 Signal greške kod *Flanger* efekta

Posljednji implementiran efekat je ***Bit Crusher***, poznatiji i kao *lo-fi* (eng. low-fidelity) efekat. Sa opcijama kao što su smanjenje frekvencije odmjeravanja (poznato i kao eng. „downsampling“ i „rate crush“) te smanjenje rezolucije (poznato i kao eng. „bit depth“ i „bit crush“), ovim efektom se postiže ciljana distorzija audio zapisa. Oba režima rada su implementirana u Python i C kodu.



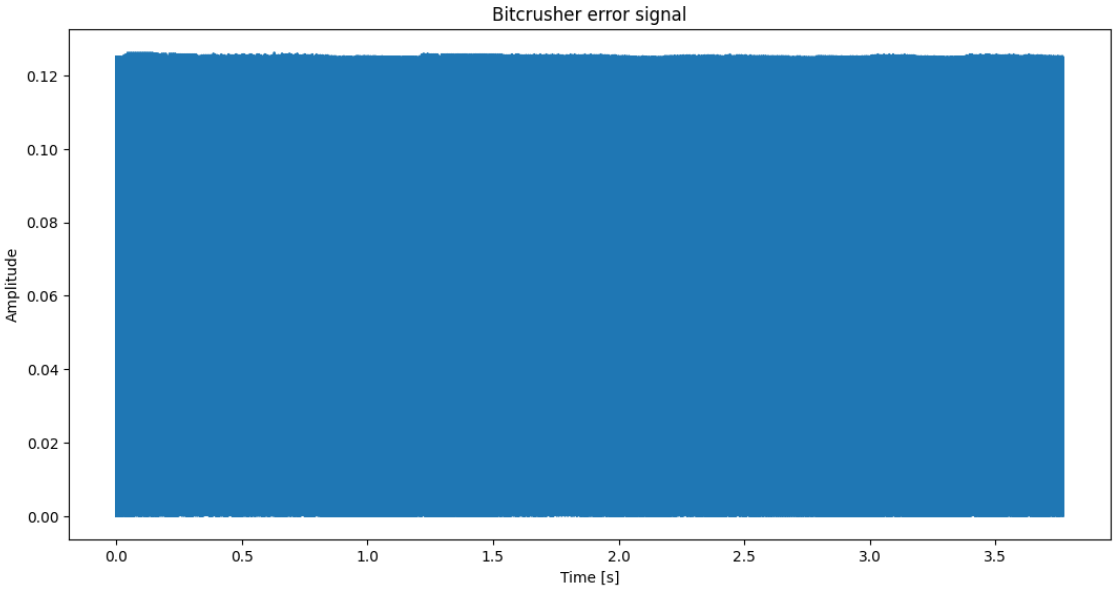
Slika 2.17 Rezultat izvršavanja *Bit crusher* efekta u Pythonu i u C-u

Iako je u funkciji ponovo korištena decimacija, verzija na kojoj smo se zadržali prilikom testiranja algoritma je ona bez niskopropusnog filtra. Signal koji mu je propušten već je filtriran u Pythonu, kao što je prethodno i decimovan, a ponovnim filtiranjem audio na izlazu izgubi svojstvenost efekta. Ipak, kao mogućnost u nekim drugim realizacijama i za potrebe drugih audio zapisa ovog projektnog zadatka unutar funkcije u Pythonu ostavlja se mogućnost filtiranja filtrom konačnog impulsnog odziva, a onda i generisanje koeficijenata filtra koji se mogu eksportovati u header file i zatim učitati u C-u. U C-u je preporučeno koristiti funkciju iz *filter.h* biblioteke – firf, koja može da bude efektivnija kada je u pitanju određivanje izlaza iz filtra, nego u slučaju funkcije *convolve*.

```
for(i = 0, j = 0; i < NUM_SAMPLES; i += DEC_FAC, j++)
{
    audio_output[j] = audio_data[i];
}
for (i=0; i<j; i++)
{
    audio_output[i] = (int)(max_value*audio_output[i])/(float32_t)max_value;
}
```

Slika 2.18 Prvobitna implementacija opcija podešavanja za *Bit Crusher* u C kodu

Za potrebe implementacije koda u C-u, bilo je neophodno dodati još jedan niz u korisničku sekciju koji će sada da ima dužinu NUM_SAMPLES/DEC_FAC, gdje DEC_FAC makro određuje cjelobrojni faktor za decimaciju, odnosno funkciju *Bit Crusher*. Kao što se da primijetiti na slici 2.13, prva petlja izvršava tipičan način rada decimacije, a to je uzimanje svakog DEC_FAC odmjerka iz prvobitnog niza, dok druga petlja radi kao kvantizer. S obzirom na to da je nakon prvobitne decimacije frekvencija odmjeravanja 11 025 Hz, a broj odmjeraka 41 567, nakon odmjeravanja u Bit Crusher funkciji broj odmjeraka iznosi 13 856, a frekvencija 3 675 Hz, čime smo prilično postigli željenu distorziju audio signala.



Slika 2.19 Signal greške kod *Bit Crusher* efekta

Nakon prezentovanja implementiranih efekata osvrnućemo se na profilisanje koda i optimizacije. Za profilisanje svih algoritama koristi se zaglavlje *cycle_count.h*, koje sadrži metode START_CYCLE_COUNT, STOP_CYCLE_COUNT te PRINT_CYCLES, kojima ćemo dobiti, a zatim i ispisati informacije o broju ciklusa koje DSP ostvari između poziva pomenutih funkcija. Prvo će biti izloženi rezultati prvobitne, naivne implementacije (bez optimizacija), nakon čega će se uključiti kompajlerske optimizacije (optimizacije za maksimalnu brzinu –Ov100), dodati i pragme za optimizaciju, ali koristiti i benefiti koje nude *intrinsic* (built-in) funkcije.

Tabela 2.1 – Ciklusi izvršavanja na DSP-u (SRAM)	*Prvobitna implementacija*	*Ručno optimizovanje i ugrađene funkcije (% u odnosu na*	*#pragma vector_for* *10,100,1000*	*Kompajlerska optimizacija (% u odnosu na prvobitnu imp.)*

		prvobitnu imp.)		
Delay	4 351 940	4 351 340	4 351 946	2 501 522 (42.5 %)
Echo	14 224 287	14 224 207	14 224 209	8 840 594 (37.8 %)
Tremolo	6 027 155	5 611 561 (6.8 %)	5 611 557	4 710 935 (21.8 %)
Flanger	5 611 431	5 299 682 (5.5 %)	5 299 714	3 879 556 (30.8 %)
Bit Crusher	1 949 056	762 113 (60.8 %)	762 113	277 145 (85.8 %)

Rezultati tabele 2.1 odnose se na optimizaciju koja je rađena sa izlaznim nizovima koji su smješteni u SRAM memoriju. Kao što se da primijetiti, optimizacija kompajlera je značajno doprinijela smanjenju broja ciklusa u svih pet slučajeva, poredeći rezultate s početka tabele i kraja, najviše u slučaju *Bit Crusher* efekta – 85.8 % najmanje u slučaju efekta *Tremolo* - 21.8 %

Pomenuto ručno optimizovanje i korišćenje ugrađenih funkcija imalo je značajan uticaj u smanjenju ciklusa, osim nešto manje u slučaju *Delay* efekta, kao i efekta *Echo*, gdje se od ugrađenih funkcija za float vrijednosti koristio *conv_fix* (za pretvaranje iz float u cjelobrojne, int vrijednosti). Razmotrićemo dodatnu mogućnost smanjenja ciklusa za ova dva efekta i pokušaj izmjene algoritma u C-u.

I - Kada je u pitanju Tremolo efekat, u odnosu na naivnu implementaciju razmatran je i slučaj poziva funkcije koja određuje LFO kao inline ili kao funkcija definisana unutar makroa. U odnosu na naivnu implementaciju, a u slučaju definisanja parametara LFO modulatora, poziv funkcije kao inline iznosio je 6 650 632 ciklusa, dok je u slučaju funkcije definisane u define iznosilo 5 861 784 ciklusa. Slučaj koji je svakako donio najmanji broj izvršavanja kod ručnog optimizovanja jeste onaj kada se u odnosu na kod sa slike 2.11 izdvaja segment koji nema potrebe da se izračunava NUM_SAMPLES puta, a u konačnoj verziji nakon profilisanja to je varijabla *lfo_w*, čime se broj ciklusa smanjio za nešto manje od pola miliona. Jedan od korisnih principa je generalno zaobilazak pozivanja funkcija unutar petlje.

II - U kontekstu optimizovanja *Flanger* efekta pozivamo se na dio teksta koji se odnosi na kod sa slike 2.15. Iako smo iskoristili funkciju *expected_true* što je poboljšanje od oko 300 hiljada ciklusa (manje u odnosu na prvobitna izračunavanja), treba uzeti u obzir da se grananja unutar petlje generalno trebaju

zaobići, tačnije raditi obrnuto. U ovom slučaju zadržali smo se na gorepomenutoj implementaciji, kako je postojala zavisnost brojača od proračunate vrijednosti kašnjenja.

III - Za najoptimizovaniji slučaj, ujedno i posljednji implementiran efekat, značajne su optimizacije postignute ručnim preuređivanjem koda. Pomenuta je strategija razbijanja petlje na više manjih petlji koja može doprinijeti značajnim rezultatima u sklopu *softverskog pipeline*-a, ali kako smo imali dvije jednostavne petlje gdje jedna uzima svaki D-ti odmjerak, a naredna treba da prođe kroz sve odmjerke ponovo i izmijeni amplitudu, sažimanjem dvaju for petlji te primjenom ugrađenim funkcija *conv_fix* te *frecipsf* (kojom smo operaciju dijeljenja zamijenili operacijom množenja, kako je u pitanju funkcija koja vraća recipročnu vrijednost float podatka) postigli veću efikasnost u kompajlerskim performansama. Napominjemo – korištenje ovakvih funkcija nema uticaj na funkcionalno ponašanje koda koji je kompajliran. U odnosu na prvobitnu implementaciju, vidimo poboljšanje koje iznosi oko 60.8 %.

Pokušaj vektorizacije petlji nije imao značajnog efekta (u slučaju bez primjene kompajlerske optimizacije #pragma optimize_for_speed), jer i sa forsiranjem #pragma vector_for(n) pretprocesorskom direktivom kompajler neće omogućiti paralelno izvršavanje više (spesifikovanih n) operacija, kako unutar petlje postoji zavisnost podataka. Slična vector_for, pretprocesorska direktiva SIMD_for navodi nas na to da se nešto bolji rezultati eventualno mogu postići ukoliko promijenimo mjesto skladištenja nizova koji su već pomenuti. U tom slučaju, misli se na promjenu sa eksterne SRAM na eksternu SDRAM memoriju, koja omogućava SIMD režim rada. Slično onome što je rađeno na početku C koda, a to je stvaranje korisničkih sekcija unutar SRAM memorije, dovoljno je da se umjesto *seg_sram* upiše *seg_dram*, a u *.ldf* file-u smještenom u *system* folderu projekta doda novi segment koji se odnosi na novu memoriju, kao i sekcija u kojoj će nizovi biti smješteni.

Tabela 2.2 – Ciklusi izvršavanja na DSP-u (SDRAM)	*Prvobitna implementacija*	*Ručno optimizovanje i intrinsic funkcije (% u odnosu na prvobitnu imp.)*	*#pragma vector_for 10, 100, 1000 (% u odnosu na optimizovan slučaj)*	*Kompajlerska optimizacija (% u odnosu na prvobitnu imp.)*
Delay	3 889 614	3 889 558	3 889 558	2 804 966 (27.8 %)
Echo	14 641 670	14 641 511	14 641 449	10 389 456 (29 %)
Tremolo	5 861 687	5 401 615 (8.0 %)	5 401 631	5 401 441 (7.8 %)

Flanger	5 091 843	4 780 096 (6.12 %)	4 676 211 (2.17 %)	3 325 370 (34.6 %)
Bit Crusher	1 530 662	969 945 (36.6 %)	588 919 (39.2 %)	106 201 (93 %)

Iz tabele 2.2 vidimo da je broj ciklusa u prvobitnoj implementaciji umanjen za nešto ispod milion ciklusa, ali da sama kompajlerska optimizacija nakon ručnog optimizovanja i korišćenja ima nešto manju efikasnost u odnosu na onu sa početka, tj. da je procentualno veće kompajlersko optimizovanje nastalo u slučaju sa tabelom 2.1. Vektorizacija petlji imala je nešto veće promjene kod *Flanger* i *Bit Crusher* efekta u ovom slučaju, kod jednog za 100 000 ciklusa, kod drugog za 380 000 ciklusa, što je procentualno izraženo u odnosu na prethodno optimizovane rezultate.

Isto tako, primijetimo da je prilično teško smanjiti broj ciklusa kod *Delay* efekta, ali pogotovo kod *Echo*, odnosno nema većeg pomjeraja. Sljedeći pokušaj za smanjenjem broja ciklusa bio je u pokušaju spajanja dvije petlje u kodu prije dijela koda prikazanog na slici 2.5:

```
/*for(i=0; i<NUM_SAMPLES; i++)*/

    for(i=0,k=delay_samples; i<NUM_SAMPLES; i++, k++)
{
    audio_output[i] = audio_data[i];
    temp_array[k] = audio_data[k-delay_samples];
}

/*for(k=delay_samples; k<NUM_SAMPLES; k++)
```

Slika 2.20 Sažimanje dvije u jednu petlju

Ispostavlja se da se na ovaj način, u slučaju SRAM memorije, za slučaj ručnog optimizovanja i ugrađenih funkcija dobijen rezultat u broju ciklusa 14 132 920, što je poboljšanje za oko 90 000 ciklusa u odnosu na rezultat u tabeli 2.1. (procentualno, učinak veoma mali – 0.64 %) Pored toga, nakon primijenjenih kompajlerskih optimizacija, broj ciklusa sveden je na 8 833 106 (samo par hiljada manje u odnosu na rezultat kompajlerske optimizacije u tabeli 2.1), dok je u radu sa SDRAM memorijom primijećeno pogoršanje u broju ciklusa u odnosu na iznesene vrijednosti. Iako to u ovom slučaju nije rađeno, u narednoj iteraciji optimizovanja, mogli bismo smjestiti i ulazni niz u niz koji je u sekciji za SDRAM memoriju i ponovo izvršiti testiranja. Slična izmjena primijenjena i na *Delay* efekat takođe u slučaju SRAM memorije čime se dobiju nešto manji rezultati (za oko 200 hiljada ciklusa manje u odnosu na prikazano u tabeli 2.1).

3. Zaključak

Za kraj izrade ovog projektnog zadatka napravićemo kratak osvrt na rad i dobijena rješenja. Istraživanje nekih najosnovnijih algoritama za implementaciju danas najpoznatijih efekata dalo je značajan uvid u to šta zapravo znači podešavanje parametara jedne gitarske pedale. S druge strane, potreba za implementacijom na DSP-u navodi nas da u cilju što boljih performansi radimo na optimizaciji pomenutih efekata i njihovom takoreći pojednostavljenju za što efikasnije izvršavanje. Poznavanje samog rada DSP sistema, korištenog kompajlera, kao i preporučenih principa optimizacije i strategija, vidimo da se to može postići. Svakako se nameće kao potreba i korištenje optimizacija koje nudi i kompajler, ali dobro je poraditi i na nekim drugim načinima implementacije algoritama, kao i razmatranju koja je memorija pogodnija (SRAM ili SDRAM te PM i DM, s obzirom na mogućnosti koje nudi SDRAM u vidu SIMD), jer kao što se da primijetiti, imala je uticaj u konačnom broju ciklusa. Upoznali smo se djelimično i sa zahtijevima MISRA:C-2004 standarda koji nalaže da se ne koristi dinamička alokacija memorije, pa smo isti princip i primijenili korištenjem statički alociranih nizova, a s obzirom na to da je poznat broj odmjeraka sa kojima se radi. Izbor tipa podataka za rad takođe je imao efekat na procesirane odmjerke, a u ovom slučaju rađeno je sa float tipom što opet otvara mogućnosti korišćenja ugrađenih funkcija za isti, a koje mogu i u drugim slučajevima da poboljšaju rad korištenog kompajlera i iskoristivost resursa. Signali greške to takođe pokazuju, ali daju naznaku da greška ipak postoji.

Prisjetimo se pristupa i sa početka izrade, a to je korištenje decimacije u signalu i prethodno potreba za filtriranjem, odnosno potreba da smanjimo broj odmjeraka koje ćemo analizirati. Recimo da smo na taj način izgubili i neke detalje signala i što je svakako imalo efekat na ono što smo na izlazu mogli reprodukovati, iako time omogućavamo manje memorijsko zauzeće. U pitanju su donekle oprečni zahtjevi, ali nije nemoguće raditi na poboljšanju, kako bi ciklusi izvršavanja na jednom DSP sistemu, ali i rezultat obrade bili minimalni i uspješni.

4. Literatura

[1] Materijali sa predavanja i laboratorijskih vježbi iz predmeta Sistemi za digitalnu obradu signala, Elektrotehnički fakultet, Univerzitet u Banjaluci

[2] CrossCore Embedded Studio, *CCES 2.9.0 C/C++ Complier Manual for SHARC Processors*, Revision 2.2, Analog Devices, May 2019.

[3] CrossCore Embedded Studio, *CCES 2.9.0 C/C++ Library Manual for SHARC Processors*, Revision 2.2, Analog Devices, May 2019.

[4] Alfredo Ricci Vasquez - Juan Carlos Bucheli Garcıa, *Implementation of sound effects in DSP*

[5] Bitcrushing and downsampling: how to add grit to any sound: *https://www.edmprod.com/bitcrushing/#h-a-quick-physics-lesson*

5. Dodaci

Pored toga što je moguće pozivati funkciju za funkcijom i upisivati obrađene odmjerke u odgovarajuće *.txt* file-ove, paljenjem dioda se može ustanoviti u kojem dijelu obrade se trenutno nalazimo – podešavanje parametara, ulazak u funkciju, izlazak iz funkcije, početak upisa u file, te provjera ispisa trećine i polovine odmjeraka, a onda i završetak obrade. S obzirom na formirane nizove unutar korisničkih sekcija, isti se mogu koristiti za kaskadno uvezivanje efekata (odnosno, da izlaz jednog bude ulaz u drugi, pri čemu se korištenje efekta *Bit Crusher* preporučuje kao posljednja stavka u nizu korištenih efekata). Zakomentarisan primjer nalazi se u C kodu.