

Elektrotehnički fakultet

Univerzitet u Banjoj Luci

**IZVJEŠTAJ PROJEKTNOG ZADATKA**

iz predmeta

**SISTEMI ZA DIGITALNU OBRADU SIGNALA**

Student: Mentori:

*Tanja Popović 1218/17* prof. dr Mladen Knežić

’ prof. dr Mitar Simić

ma Vedran Jovanović

dipl. inž. Damjan Prerad

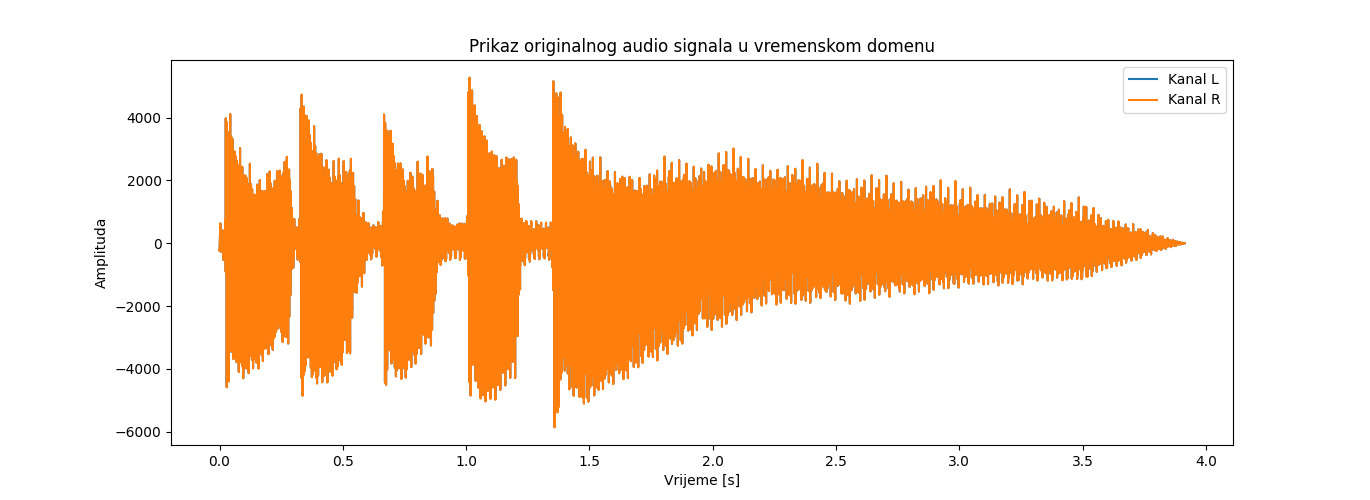
Februar 2024. godine

# Opis projektnog zadatka

U sklopu projektnog zadatka potrebno je realizovati sistem za dodavanje muzičkih efekata u audio signal korištenjem ADSP-21489 razvojnog okruženja. Ulazni signal, muzički efekti, kao i svi izlazni signali su prvenstveno generisani unutar Python programskog jezika (program effects.py) i služe kao referenca signalima implementiranim u okviru ciljnog programskog paketa CrossCore Embedded Studio. Nakon primjene muzičkog efekta, odmjerci izlaznog signala su upisani u odgovarajuće tekstualne datoteke koje se iz Pythona čitaju, prikazuju i eksportuju kao *wav* fajlovi tako da se primijenjeni efekti mogu čuti. Pored implementacije efekata, u sklopu CCESa izvršeno je i profilisanje koda, a signali su međusobno upoređeni pomoću signala greške. Implementirano je ukupno 11 muzičkih efekata: Delay, Echo, Compressor, Noise Gate, Envelope Filter, Volume Pedal, Tape Saturation, Octave Up, Tremolo, Override i Distorsion.

# Izrada projektnog zadatka

Za potrebe izrade projektnog zadatka, kao ulazni signal iskorišten je konkretan zvuk gitare – audio fajl pod nazivom *guitar\_sound.wav.* Radi se o stereo signalu, frekvencije odmjeravanja 44,1 kHz i trajanja 3,9 sekundi prikazan na Slici 2.1. Radi lakše implementacije zadatka, stereo signal je konvertovan u mono signal (*source\_audio\_mono.wav*) i kao takav predstavlja „sirovi“ ulazni signal na koji se primjenjuju audio efekti. Softverska implementacija audio efekata zasnovana je na jednačinama diferencija odgovarajućih sistema.



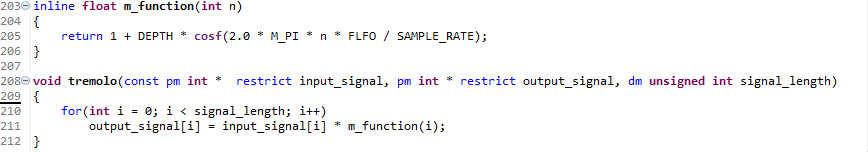
*Slika 2.1* – Originalni, stereo signal

Proces profilisanja koda je obavljen pomoću dva makroa koje sadrži zaglavlje *cycle\_count.h,* a prikazuju broj procesorskih ciklusa koji se potroše na izvršavanje određenog dijela koda. Broj procesorskih ciklusa koji se potroše na izvršavanje pojedinačnih funkcija koje predstavljaju muzičke efekte prikazan je u Tabeli 2.1. Profilisanje je izvršeno bez ikakve optimizacije, a kao što se može vidjeti najviše najviše procesorskih ciklusa treba za izvršavanje funkcije *tremolo*, koja će u ovom slučaju predstavljati fokus optimizacije – na ovaj način je detektovano usko grlo (eng. *bottle neck*) u kodu.

*Tabela 2.1* – Muzički efekti sa odgovarajućim brojem procesorskih ciklusa (bez optimizacije)

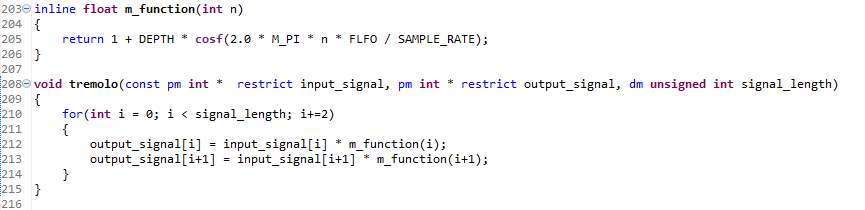
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Delay | Echo | Compressor | Noise Gate | Volume Pedal | Tape Saturation |
| 20 283 523 | 18 158 689 | 13 967 836 | 10 489 237 | 8 713 719 | 14 174 859 |
| Octave Up | Envelope Filter | **Tremolo** | Distorsion | Override |
| 9 324 359 | 17 490 248 | **25 134 797** | 12 695 583 | 21 415 226 |

Prvi korak pri optimizaciji je uključivanje kompajlerske optimizacije i dodavanje identifikatora *restrict* pored parametara funkcije. Kako se sa Slike 2.2 može vidjeti da se *tremolo* funkcija sastoji od *for* petlje sa velikim brojem iteracija, prilikom čega se u svakoj iteraciji poziva *m\_function* funkcija, ista je proglašena kao *inline*. Ovakav vid optimizacije smanjuje broj procesorskih ciklusa na 18 465 873.



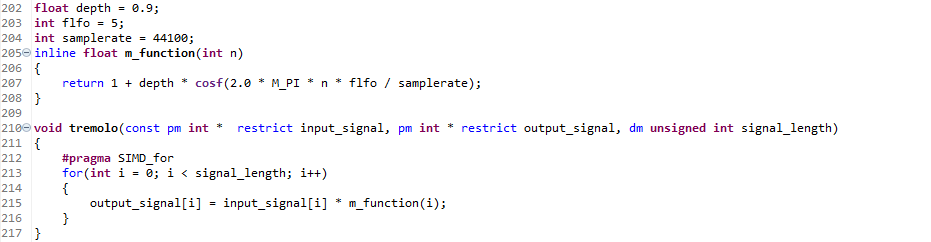
*Slika 2.2* – Implementacija Tremolo funkcije

Sljedeći pokušaj optimizacije jeste ručno odmotavanje petlje, tako da imamo manje iteracija ali je blok izvršavanja proširen. Primjer ručnog odmotavanja petlje unutar *tremolo* funkcije prikazan je na Slici 2.3. Broj ciklusa nakon ovakvog načina implementacije funkcije je smanjen na 16 731 696, a pored toga posmatrani su rezultati dodavanja *pragma* direktiva koje se odnose na odmotvanje petlji kao što je *loop\_unroll* sa parametrima 1000, 100 i 10. To je dalo sljedeće rezultate: 21 726 042, 18 310 214 i 17 690 884 za nabrojene parametre, respektivno. Ovakvi rezultati predstavljaju pogoršanje u odnosu na kompajlersku optimizaciju, a s obzirom na uputstvo u specifikaciji kompajlera da nije potrebno ručno odmotavati petlje već to prepustiti kompajleru, vratićemo se prvobitnoj implementaciji funkcije *tremolo* kao na Slici 2.2.



*Slika 2.3* – Ručno odmotavanje petlje unutar Tremolo funkcije

Kako su unutar CCESa parametri svih muzičkih efekata definisani kao makroi, uporedićemo brzinu izvršavanja funkcije ukoliko se parametri definišu kao globalne promjenjive. Pred toga, unutar *tremolo* funkcije je dodana druga pragma direktiva za optimizaciju petlji, koja se odnosi na vektorizaciju – SIMD\_for. Primjer implementacije ovakve funkcije prikazan je na Slici 2.4. Profilisanjem ovakve funkcije, dobili smo poboljšanje u odnosu na kompajlersku optimizaciju te sada broj ciklusa iznosi 11 994 354. Ukoliko želimo da uporedimo da li je izvršavanje funkcije brže sa globalnim promjenjivim ili makroima, uz vektorizaciju, dolazimo do sljedećeg rezultata – 9 598 584 ciklusa što predstavlja značajno manje procesorskih ciklusa u odnosu na početnu vrijednost, i najbolji rezultat optimizacije. Prikaz svih rezultata optimizacije funkcije *tremolo* prikazan je u Tabeli 2.2.



*Slika 2.4* – Vektorizacija petlje i dodavanje globalnih promjenjivih umjesto makroa

*Tabela 2.2* – Optimizacioni koraci za funkciju Tremolo i dobijeni rezultati

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Bez optimizacije | Kompajlerska optimizacija, inline, restrict | Ručno odmotavanje petlje | #pragma no\_vectorization | #pragma loop\_unroll 1000, 100, 10 | #pragma SIMD\_for, globalne promjenjive | #pragma SIMD\_for, makroi |
| 25 134 797 | 18 465 873 | 16 731 696 | 16 726 042 | 21 726 042 18 310 214 17 690 884 | 11 994 354 | 9 598 584 |

S obzirom da je prvenstveno kompajlerska optimizacija značajno smanjila broj ciklusa potrebnih za izvršavanje tremolo funkcije, a zatim i vektorizacija, posmatraćemo kako kompajlerska optimizacija, a kako vektorizacija utiče na ostale funkcije. Upoređivanjem rezultata u Tabelama 2.3 i 2.4, vidimo da vektorizacija ipak nije uvijek najbolja tehnika za optimizaciju. Broj ciklusa izvršavanja pojedinih funkcija je ostao isti, odnosno kompajler je vid optimizacije za te funkcije prepoznao bez vektorizacije (kao što postoji pragma direktiva no\_vectorization). Pored toga, forsiranje vektorizacije može dovesti do povećanja broja ciklusa prilikom izvršavanja funkcija kao što je to primjer sa Volume Pedal i Octave Up funkcijama.

*Tabela 2.3* – Broj ciklusa nakon kompajlerske optimizacije

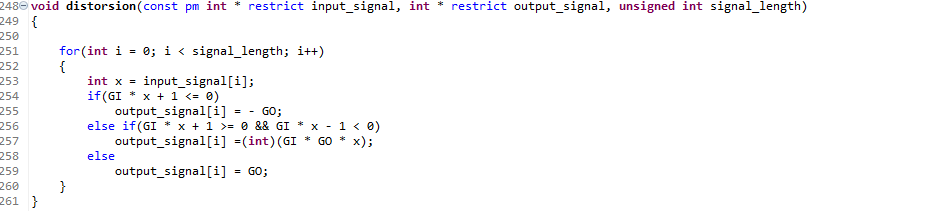
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Delay | Echo | Compressor | Noise Gate | Volume Pedal | Tape Saturation |
| 14 156 568 | 14 690 624 | 4 844 718 | 5 328 974 | 4 844 724 | 10 783 200 |
| Octave Up | Envelope Filter | Tremolo | Distorsion | Override |
| 4 844 704 | 7 438 216 | 18 465 978 | 7 344 552 | 15 929 756 |

*Tabela 2.4* – Broj ciklusa nakon kompajlerske optimizacije i vektorizacije

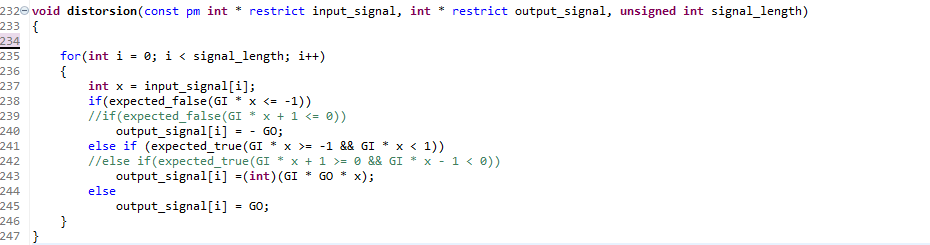
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Delay | Echo | Compressor | Noise Gate | Volume Pedal | Tape Saturation |
| 14 156 392 | 14 690 624 | 4 844 686 | 5 291 502 | 5 080 568 | 10 782 736 |
| Octave Up | Envelope Filter | Tremolo | Distorsion | Override |
| 5 080 134 | 7 438 200 | 9 598 584 | 7 344 572 | 15 929 376 |

U specifikaciji kompajlera se dodatno razmatraju slučajevi kako uslovna grananja unutar petlji takođe značajno utiču na brzinu izvršavanja programa. Savjetuje se da se petlje nalaze unutar grananja, a ne obrnuto, ukoliko je to dozvoljeno. U konkretnom slučaju, imamo primjer implementacije funkcija *distorsion* i *override*, gdje se unutar petlje nalaze uslovna grananja koja se zbog zavisnosti od iteratora ne mogu izbaciti izvan petlje, kao što je prikazano na Slici 2.5. Ukoliko se uslovna grananja ne mogu izbaciti, savjetuje se korištenje funkcija *expected\_false* i *expected\_true*, kojima se vrši predviđanje rezultata uslova i na taj način ubrzava njihovo izvršavanje. Pored toga, unutar uslovnog bloka za poređenje se savjetuje poređenje neke vrijednosti sa nulom, umjesto sa nekom drugom vrijednošću. Primjer izmjene implementacije funkcije *distorsion* prikazana je na Slici 2.6.

Kao što je prikazano u Tabeli 2.4, vektorizacija nije doprinijela poboljšanju vremena izvršavanja funkcije *distorsion*. Ukoliko se uslovi unutar grananja napišu na drugačiji način, kao što je prikazano na Slici 2.5, broj ciklusa potebnih za izvršavanje *distorsion* funkcije će se povećati na 7 370 468. S obzirom na to, implementacija će ostati kao na Slici 2.6, a uz dodavanje prethodno pomenutih funkcija rezultat izvršavanja iznosi 7 108 828 ciklusa. Ovim primjerom je pokazano da drugačiji načini implementacije mogu uticati na brzinu izvršavanja programa. Poboljšanje vremena izvršavanja *distorsion* funkcije prikazano je u Tabeli 2.5.



*Slika 2.5* – Implementacija distorsion efekta



*Slika 2.6* – Implementacija distorsion efekta dodavanjem prediktivnih funkcija

Tabela 2.5 – Rezultati izvršavanja *distorsion* funkcije

|  |  |  |
| --- | --- | --- |
| Kompajlerska optimizacija | Uslovno poređenje sa nultom vrijednošću | Prediktivne funkcije, poređenje sa nenultom vrijednošću |
| 7 344 572 | 7 370 468 | 7 108 828 |

# Zaključak

# Literatura