VEŽBA 3 – Izrada DSP aplikacije 1. deo – metodologija razvoja DSP aplikacija

3.1 Uvod

Na digitalnim signal procesorima se realizuje širok spektar aplikacija: audio i video koderi, algoritmi zaštite govora, modemi, eliminacija šuma, prepoznavanje govora itd. Širok spektar algoritama podrazumeva i širok spektar digitalnih signal procesora: od jeftinih sistema za masovnu namenu kod kojih je cena najvažniji faktor, do procesora sa visokim performansa na kojima se realizuju složeni i veoma zahtevni algoritmi. Algoritmi se najčešće razvijaju na procesorima opšte namene u nekom od viših programskih jezika (C ili C++) ili u formi matematičkog modela. Na taj način se dobija referentni kod algoritma, koji nije zavisan od fizičke platforme i predstavlja osnovu za razvoj softverskih aplikacija za DSP procesore.

Proces razvoja softvera za platforme sa ograničenim resursima je složen posao i podrazumeva poznavanje aritmetike, hardverskih proširenja i instrukcionog skupa ciljne platforme. Ovaj proces neposredno utiče na kvalitet i cenu krajnjeg proizvoda. Tokom izrade ove vežbe objasniće se jedna metodologijom razvoja algoritama na digitalnim signal procesorima sa aritmetikom u nepokretnom zarezu. Pokazaće se kako se iterativnim postupkom referentni kod prilagođava (modifikuje) tako da može da se prevede za ciljnu platformu. Detaljnije će se obraditi aritmetika procesora CS48x, na koji način se vrši provera ispravnosti aplikacija, i kako se ta provera može automatizovati.

3.2 Metodologija razvoja DSP aplikacija

Proces razvoja softverskih aplikacija predstavlja struktuirani skup aktivnosti koji za cilj ima razvoj softverskog proizvoda. Softverski proizvod se sastoji od razvijenih programa i dokumentacije. Dokumentacija podrazumeva spisak zahteva koje softverska aplikacija treba da ispuni, opis dizajna aplikacije i korisnička uputstva. Kao što je rečeno, proces razvoja softvera je složen proces i neposredno utiče na kvalitet i cenu krajnjeg proizvoda. Pod efikasnim razvojem softvera podrazumevamo da se sa konačno definisanim obimom resursa, u predviđenom roku, postigne proizvod visokog kvaliteta. Pri tome, kriterijumi kvaliteta koji određuju dobro razvijen softverski proizvod su:

- funkcionalnost – ispunjavanje svih predviđenih zadataka
- pouzdanost – otpornost na greške i mogućnost oporavka usled otkaza
- upotrebljivost - odgovarajuća korisnička sprega i adekvatna dokumentacija
- efikasnost – korišćenje minimalnog skupa resursa
- održavanje – lakoća ispravke grešaka i prilagođavanja softvera izmenjenim zahtevima kupca
- pokretljivost – mogućnost prilagođavanja softverskog proizvoda različitim okruženjima

Uopšteno, modeli razvoja softvera su apstrakcije koje pomažu u procesu razvoja softvera. Model predstavlja formalan opis faza u razvoju softvera i postupaka karakterističnih za svaku od tih faza. Odabira se u zavisnosti od prirode projekta i aplikacije, tehničke orijentacije ljudi koji će učestvovati u razvoju, kao i metoda i alata koji će se koristiti.

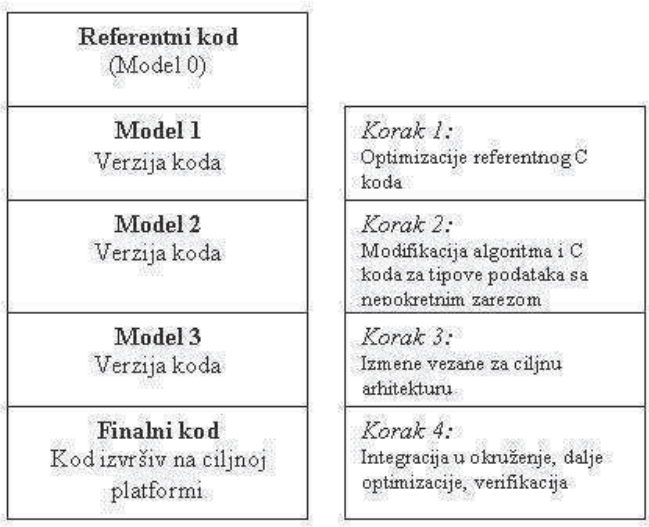
Proces razvoja softverskih aplikacija za sisteme zasnovane na digitalnim signal procesorima sadrži određene specifičnosti u odnosu na procese razvoja na opštenamenskim platformama. Fizička arhitektura ciljnog procesora značajno definiše kritične zahteve: memorijska ograničenja, tačnost izlaznih odbiraka, dužinu programa i vreme izvršenja. Ovi zahtevi nisu nezavisni, pa se tokom razvoja mora voditi računa o svim resursima platforme. Recimo, poboljšanje tačnosti pri obradi odbiraka dovodi do većeg broja instrukcija. Ili, smanjenje vremena izvršenja može da se izvede na račun povećanja memorijskih zahteva.

Posle više od 15 godina iskustva u razvoju softvera na raznim digitalnim signal procesorima, na Odseku za računarsku tehniku i računarske komunikacije (Fakultet tehničkih nauka u Novom Sadu) je razvijena metodologija za razvoj aplikacija na ovakvim sistemima. Ova metodologija je primarno razvijena za procesore sa aritmetikom u nepokretnom zarezu i oslanja na C kompajler za DSP, ali je velikim

delom primenjiva i kada se pristupa razvoju zasnovanom isključivo na asemblerskom jeziku.

Metodologija propisuje korake koji olakšavaju i ubrzavaju razvoj i održavanje koda. Prvi korak je analiza ulaznih ispitnih vektora, analiza zadatog algoritma i, ukoliko je potrebno, formiranje referentnog C koda. Rezultat ovog koraka jeste referenti kod, odnosno **Model 0** i skup izlaznih referentih vektora koji služe za proveru ispravnosti svake naredne faze. Sledeći korak, **Model 1,** uvodi funkcionalne optimizacije u C kod. **Model 2** podrazumeva prilagođenje koda aritmetici ciljne arhitekture, dok **Model 3** predstavlja potpuno prevodiv C kod za namensku platformu. Nakon uspešno formiranog Modela 3 sledi detaljna procena utroška resursa (odnosno profilisanje). Ukoliko su rezultati profilisanja zadovoljavajući, sledi povezivanje koda sa programskim okruženjem ciljne platforme i pisanje prateće dokumentacija. Model 0, Model 1 i Model 2 se prevode opštenamenskim kompajlerom, dok se Model 3 prevodi kompajlerom za ciljnu platfromu.

Treba napomenuti da razvoj aplikacije za ciljnu platformu predstavlja iterativni postupak. Ako se na osnovu procene utroška resursa zaključi da rešenje ne zadovoljava unapred poznate kriterijume, pristupa se modifikovanju rešenja polazeći od Modela 1.



Slika 3.1 - Tok implementacije softvera na digitalnim signal procesorima sa aritmetikom u nepokretnom zarezu

3.2.1 Model 0 – Referentni kod

U okviru izložene metodologije podrazumevano je da je početna tačka razvoja referentni algoritam realizovan kao C programski kod. Ovo je i najčešća realizacija algoritama, jer je mnogo lakše i brže napisati algoritamski kod kada se ne vodi računa o ograničenjima fizičke platforme. Ako je algoritam definisan u nekoj drugoj formi (recimo u obliku standarda, nekog drugog programskog jezika ili matematičkog modela), ova metodologija zahteva da se prvo napravi C referetni kod, koji se potom može prilagoditi različitim DSP platformama. Namena referentnog koda je ispitivanje i verifikacija algoritma, pa je po pravilu neefikasan za ciljnu platformu. Odnosno, kod pisanja referentog koda se vodi računa isključivo o ispravnosti rešenja, kvalitetu i tačnosti izlaznih vrednosti, dok se problemi koji se tiču implementacije zanemaruju. Referentni rezultati, odnosno izlazni vektori koje generiše referenti kod (Model 0) služe za ispitivanje svake sledeće izmene u kodu. Poređenjem referentnih rezultata sa rezultatima modifikovanog koda možemo proveriti da li je izmena dovela do narušavanja semantike koda, pa samim tim uočiti i otkriti greške u ranoj fazi razvoja softvera.

U nekim slučajevima algoritam je realizovan u asemblerskom jeziku za neku drugu platformu. Ako programski prevodilac podržava opciju koja se naziva „dekompajliranje“, referentni C kod se može generisati na osnovu datog asemblerskog koda (što znatno skraćuje vreme potrebno za formiranje referentnog modela).

Kao sto je rečeno, izlazne vrednosti iz Modela 0 predstavljaju referentne vrednosti, i za izlaz svake naredne faze smatramo da je ispravan ako odgovara ovim vrednostima. Naglašavamo da se Model 0 ne sme menjati u daljem toku izrade aplikacije.

3.2.2 Model 1 – Funkcionalna optimizacija C koda

Pošto se referentni kod (Model 0) po pravilu realizuje bez znanja o osobinama i ograničenjima ciljne platforme, prvi korak u daljem razvoju je funkcionalna optimizacija C koda prilagođena DSP procesoru. Ova optimizacija podrazumeva izmene u referentnom kodu u skladu sa hardverskim proširenjima procesora: broju registra, veličini memorije, veličini steka, radu adresnog generator i hardverskim petljama. Ove izmene se svode na organizaciju koda i podatka, tako da nakon prevođenja utrošak procesorskih resursa bude optimalan i u skladu sa zahtevima.

Treba naglasiti da su funkcionalne modifikacije, u ovoj fazi razvoja, takve da se Model 1 (kao i Model 0) prevodi opštenamenskim kompajlerom. Nakon primene optimizacionih tehnika rezultat izvršavanja Modela 1 mora biti identičan na nivou bita rezultatu izvršavanja Modela 0.

U daljem tekstu je dat pregled tehnika optimizacije koje se primenjuju u ovoj fazi razvoja koda.

3.2.2.1 Organizacija podataka

Organizacija podataka, odnosno korišćenje globalnih ili lokalnih promenjivih, prosleđivanje parametara funkcijama, korišćenje struktura podataka, raspored nizova i sl, značajno utiče na kvalitet prevedenog C koda. Kod razvoja softvera za opštenamenske procesore se najčešće ne razmišlja na koji način su resursi raspoređeni nakon prevođenja koda. Kao što je poznato, programski jezik C dozvoljava korišćenje lokalnih i globalnih promenljivih, pri čemu životni vek lokalnih promenljivih traje samo u domenu u kom su definisane (npr: u telu funkcije). Resursi koji se prevođenjem dodeljuju lokalnim promenljivim su registri procesora ili lokalni stek, dok se globalne promenljive smeštaju u radnoj memoriji.

DSP procesori su platforme sa ograničenim resursima i imaju mali broj registara (recimo, dva skupa od po 8 registara) i mnogo manji stek od opštenamenskih procesora (npr: 16 ili 32 memorijske reči). Isto tako, smeštanje promenjivih na stek obično iziskuje više instrukcija obrade nego neposredno smeštanje u memoriju podataka (jer je potrebno pristupiti steku i premestiti promenjive u memoriju ili registre pre obrade podataka). Sa druge strane, neracionalno korišćenje globalnih promenljivih može da rezultuje popunjavanjem memorije promenljivim čiji je životni vek zapravo mali ili se uopšte ne koriste, a dovodi i do narušavanja čitljivosti i razumljivosti programskog koda.

Zbog svega ovoga je u ovoj fazi razvoja potrebno organizovati podatke tako da nakon prevođenja koda utrošak procesorskih resursa bude optimalan.

3.2.2.1.1 Smanjenje broja argumenata kod funkcija obrade

Jedan od primera korišćenja globalnih promenljivih za optimizaciju koda je smanjenje broja argumenata koji se prosleđuju funkcijama. Parametrizovane funkcije obezbeđuju modularnost koda, ali prosleđivanje argumenata zahteva dodatne instrukcije prilikom poziva funkcije:

- dodeljivanje vrednosti parametrima funkcije u skladu sa pozivnom konvencijom,
- čuvanje trenutne vrednosti korišćenih registara pre poziva funkcije,
- vraćanje tih vrednosti nakon izvršenja funkcije.

Ako broj argumenata funkcije prelazi broj slobodnih/dozvoljenih procesorskih registara, dolazi do smeštanja parametara na stek. Kao što je rečeno, korišćenje steka dodatno uvećava broj instrukcija prilikom poziva funkcije i pristupa parametrima unutar tela funkcije.

U sledećem primeru prikazano je kako se smanjuje broj parametara upotrebom globalnih promenljivih i/ili upotrebom definisanih simbola.

Primer:

```
int foo(int window_size, int sample_rate,
        int number_of_channels);
```

Kod nakon optimizacije:

```
#define window_size 512
#define number_of_channels 7
      int sample_rate 32000;

      int foo();
```

Detaljan opis prosleđivanja parametara kod procesora CS48x dat je u poglavlju koje sadrži opis CCC kompajlera, u odeljku koji sadrži opis pozivne konvencije.

3.2.2.1.2 Uklanjanje lokalnih struktura

U referentnom kodu se često koriste velike strukture u kojima se opisuje trenutno stanje aplikacije ili pojedinačnih procesnih modula. Ponekad se takve strukture prosleđuju kao parametri funkcija, a češće se prosleđuje pokazivač na strukturu. Ovakvo prosleđivanje rezultuje generisanjem neefikasnog koda i samim tim ga treba izbegavati. Poželjno je uočiti takve strukture i napraviti njihove globalne instance.

Primer:

```
      int foo (t_lvl_mod_cfg S)

      {
            ...
            x = S.scale_factor;
            ...
      }
```

Kod nakon optimizacije:

```
      t_lvl_mod_cfg g_S;
      int foo ()
      {
            ...
            x = g_S.scale_factor;
            ...
      }
```

Kada se funkciji prosleđuje pokazivač na strukturu, potrebno je obratiti pažnju da li je u kodu realizovana jedna ili vise struktura. Samo ukoliko se koristi ista struktura, moguće joj je neposredno pristupati u telu funkcije.

Primer:

```
int foo (t_lvl_mod_cfg* pS)
{
      ...
      x = pS->scale_factor;
      ...
}

void main ()
{
      ...
      return foo (&g_S);
}
```

Kod nakon optimizacije:

```
int foo ()
{
      ...
      x = g_S.scale_factor;
      ...
}
```

3.2.2.2 Pristupanje podacima

Jedno od hardverskih proširenja digitalnih signal procesora je i jedinica za generisanje adresa podataka. Adresni generator je u stanju da obavi izvestan broj aritmetičkih operacija i da generiše adresu na kojoj se nalazi podatak. Po pravilu podržava kružno ili modulo adresiranje, kao i režim bit-obrnutog pristupa koji se koristi kod FFT-a.

U ovoj fazi razvoja softvera, operacije pristupa podacima potrebno je prilagoditi radu adresnog generatora. Ovo prilagođenje podrazumeva zamenu posrednog pristupa elementima niza, koristeći indeks i C-ovske operatore pristupa, pristupom preko pokazivača na element. Na ovaj način se omogućava C kompajleru da ovu instrukciju prevede u kraći i brže izvršiv kod. Takođe, olakšava se proces ručnog pisanja asemblerskog koda na osnovu referentnog, jer je ovako napisan C kod lakše predstaviti ponuđenim instrukcijskim skupom. Od svih pomenutih tehnika, ova optimizacija najviše utiče na veličinu generisanog koda i brzinu izvršavanja.

Primer:

```
int i, j;
for ( i = 0; i < n ; i++ )
{
      for ( j = 0; j < m ; j++ )
      {
            a[i] += 2 * b[j];
      }
}
```

Kod nakon optimizacije:

```
int* p_a;
int* p_b;
for ( p_a = a; p_a < a+n ; p_a++ )
{
      for (p_b = b; p_b < b+n ; p_b++)
      {
            *p_a += 2 * *p_b;
      }
}
```

3.2.2.3 Optimizacija programskih petlji

Velik deo obrade digitalnih signala se zasniva na obradi nizova podatka, tipičan primer za ovo je digitalni filter. Samim tim, najveći deo procesorskih ciklusa tokom izvršavanja DSP aplikacija troši se na izvršavanje programskih petlji. Ušteda jednog instrukcionog ciklusa unutar tela petlje dovodi do uštede *N* instrukcijskih ciklusa, gde *N* predstavlja broj iteracija petlje po jedinici obrade.

Primer:

```
for ( i = 5; i < n ; i++ )
{
      x = y + z;
      a[i] = 6 * i *x * x;
}
```

Kod nakon optimizacije:

```
x = y + z;
int t = x*x;
for ( i = 5; i < n ; i++ )
{
      a[i] = 6 * i * t;
}
```

Deo koda unutar programskih petlji ponekad je nezavisan od trenutne iteracije ili čitave petlje, pa nema potrebe da se izvršava više puta. Ovaj kod naziva se nezavisni, odnosno invarijantni kod. Primer ovakvog koda jeste dodeljivanje konstantne vrednosti nekom resursu. Umesto da se ta dodela obavlja u svakoj iteraciji petlje, može se izvršiti samo jednom, pre ulaska u petlju. Ovaj pristup produžava životni vek promenljivih i dodatno opterećuje resurse, ali ubrzava izvršenje celokupnog koda jer skraćuje telo petlji.

3.2.3 Model 2 – Prilagođenje koda aritmetici DSP-a

Referentni kod algoritma je najčešće zasnovan na aritmetici sa pokretnim zarezom, koja se bez modifikacija ne može prevesti kompajlerom za procesore sa aritmetikom u nepokretnom zarezu. U ovoj fazi razvoja potrebno je funkcionalno optimizovan kod prilagoditi aritmetici ciljne platforme, ali tako da je i dalje prevodiv opštenamenskim kompajlerom.

S obzirom da C standard ne sadrži podršku za tipove u nepokretnom zarezu, način na koji je ta aritmetika podržana zavisi od implementacije C kompajlera. *Embedded C standard* je proširenje C standarda za namenske procesore, koje propisuje C-ovske tipove podataka u nepokretnom zarezu i operacije nad njima. Ako kompajler za ciljnu platformu podržava ovo proširenje, referentni kod se modifikuje tako da se obrada zasniva na tipovima podataka u nepokretnom zarezu. Ako kompajler ne podržava ovo proširenje, obrada podataka se mora prilagoditi celobrojnoj aritmetici.

Da bi modifikovan kod bio i dalje prevodiv opštenamenskim kompajlerom, često se uvode klase koje služe za emulaciju tipova u nepokretnom zarezu. Na ovaj način se obezbeđuje kod čija aritmetika u potpunosti prati aritmetiku ciljnog procesora, a i dalje se može prevoditi i kontrolisano izvršavati istim prevodiocem kao i Model 0 i Model 1 (što omogućava lakše otklanjanje grešaka i poređenje međurezultata obrade sa referentnim kodom) .

Kompajler koji se koristi za razvoj aplikacija na procesorima CS48x (nazvan CCC kao skraćenica od Cirrus C Compiler), podržava proširenje C standarda za namenske procesore, pa je nakon analize koda potrebno zameniti sve tipove podataka u pokretnom zarezu sa odgovarajućim tipovima u nepokretnom zarezu. U okviru skupa alata za razvoj softvera za platformu CS48x date su C++ klase koje služe za emulaciju tipova u nepokretnom zarezu: *fract* i *accum*. Preciznost ovih tipova odgovora preciznosti tipova u nepokretnom zarezu kod CCC kompajlera.

Najveći problemi prilikom formiranja Modela 2 su posledica različite preciznosti kod aritmetike sa nepokretnim zarezom u odnosu na aritmetiku sa pokretnim zarezom. U daljem tekstu su predloženi neki od načina za prevazilaženje ovih problema.

Potrebno je napomenuti da promena preciznosti često dovodi do toga da Model 2 ne može da bude bit-identičan sa Modelom 1.

3.2.3.1 Prebacivanje konstanti (koeficijenata) iz pokretnog u nepokretni zarez

3.2.3.1.1 Problem kod koeficijenata čija je vrednost van zadatog opsega

Kao što je rečeno, platforma CS48x je 32-bitna platforma sa aritmetikom u nepokretnom zarezu, pri čemu je opseg vrednosti tipova u nepokretnim zarezom [-1, 1). Konstante koje su van ovog opsega je potrebno skalirati, da bi se omogućile aritmetičke operacije nad ispravnim vrednostima.

Skaliranje se vrši promenom prezentacije brojeva u nepokretnom zarezu. Kod binarnog prikaza brojeva u nepokretnom zarezu, svaki bit pre zareza predstavlja stepen dvojke, dok svaki bit posle zareza predstavlja recipročnu vrednost stepena dvojke. Tip *fract* u okviru C proširenja za namenske platforme predstavljen je u formatu s1.31, otuda i potiče pomenuti interval [-1, 1). Ukoliko podelimo taj broj sa 2, smatramo da broj ima istu vrednost u prezentaciji s2.30, i tako dalje. Sve operacije skaliranja u kodu (koje odgovaraju deljenju ili množenje sa odgovarajućim stepenom dvojke) se realizuju kao logički pomeraj.

Primer:

```
// 48 kHz
{  3.580469254416130e-014,
                          0, 9.338952139211192e-001,                          0, 1.034871980488338e+000,
   -8.851313402201296e-001, 1.881032966150447e+000, 1.053627350897576e+000, 2.052424807501546e+000,
   -9.073018742276428e-001, 1.903448412575458e+000, 1.011402336878282e+000, 2.010233857798241e+000,
   -9.356046394317145e-001, 1.931455342504913e+000, 9.662620695152961e-001, 1.965132140417648e+000,
   -9.775108620840036e-001, 1.973263048666602e+000, 9.384420320790645e-001, 1.937337213794234e+000,
}
```

U datim primeru vidimo da neke konstante ne upadaju u interval [-1, 1). Neke su malo vece od 1, ali postoje i one koje su vece od 2. Konstante koje se nalaze u intervalu od [1,2) treba da podelimo sa 2, a konstante koje se nalaze u intervalu od [2,3) treba da podelimo sa 4, pa je u ovom primeru rešenje podela niza na dve celine:

Grupa konstanti skaliranih sa faktorom 2:

```
FRACT_NUM( 0.00000000000000000), FRACT_NUM( 0.46694760676473379), FRACT_NUM( 0.00000000000000000),
FRACT_NUM(-0.44256567023694515), FRACT_NUM( 0.94051648303866386), FRACT_NUM( 0.52681367564946413),
FRACT_NUM(-0.45365093694999814), FRACT_NUM( 0.95172420609742403), FRACT_NUM( 0.50570116844028234),
FRACT_NUM(-0.46780231967568398), FRACT_NUM( 0.96572767104953527), FRACT_NUM( 0.48313103476539254),
FRACT_NUM(-0.48875543102622032), FRACT_NUM( 0.98663152428343892), FRACT_NUM( 0.46922101592645049),
```

Grupa konstanti skaliranih sa faktorom 4:

```
FRACT_NUM( 0.25871799513697624),
FRACT_NUM( 0.51310620177537203),
FRACT_NUM( 0.50255846465006471),
FRACT_NUM( 0.49128303490579128),
FRACT_NUM( 0.48433430353179574)
```

Na ovaj način će koeficijenti biti pravilno smešteni u memoriju procesora (ili registre koji su isto 32-bitni), ali ih je potrebno vratiti na početnu vrednost pre obrade. Dat je primer izmene koda koji vrši računanje upotrebom skaliranih koeficijenata.

Primer:

```
    for(j = 0; j < N; j+=2)
    {
        yn = xn1*b1; //koeficijent b1 je pomeren u desno za 2 bita
        yn << 1; // vrati jedan bit
        yn += xn2 * b2; // koeficijenti b2, a1 i a2 su pomereni za 1
bit
        yn += yn1 * a1;
        yn += yn2 * a2;
        yn <<= 1; // pomeri za jos jedan bit
        yn = (DSPaccum)*pIn + yn;
    }
```

3.2.3.2 Problem konstanti čija je vrednost isuviše mala

Zaseban problem jeste pojava da je vrednost konstante u nepokretnom zarezu isuviše mala da bi bila predstavljena sa datim tipom podataka. Preciznost tipa u nepokretnom zarezu može se predstaviti rastojanjem Δ između bilo koja dva susedna broja predstavljena ovim tipom. Ako je bilo koji tip u nepokretnom zarezu predstavljen u formatu <*a.b*>, gde je *a* broj bita kojim je predstavljen celobrojni deo broja, a *b* broj bita kojim je predstavljen razlomljeni deo broja, tada je $\Delta=2^{-b}$. Konstanta ima isuviše malu vrednost da bi se predstavila određenim tipom ukoliko je njena vrednost manja od Δ za taj tip. U zavisnosti od samog algoritma, ovaj problem se rešava na različite načine. U nekim slučajevima je dovoljno ovakav broj zaokružiti na najbližu vrednost. Međutim, u određenim slučajevima ovakav pristup dovodi do gubitka preciznosti i ispravnosti rezultata.

Ovaj problem se često javlja u slučajevima kada je potrebno rezultat neke obrade pomnožiti sa koeficijentom pojačanja. Ukoliko je vrednost tog koeficijenta manja od Δ, i zaokružena na 0 tada je konačan rezultat obrade uvek 0. Jedan od načina da

se ovaj problem reši jeste primena koeficijenta pojačanja po međufazama obrade. Neophodno je podeliti obradu na određene faze, potom razložiti konačni koeficijent tako da rezultat množenja novim koeficijentima po fazama daje rezultat ekvivalentan proizvodu ukupnog koeficijenta i izlazne vrednosti obrade, i zatim izvršiti množenje sa odgovarajućim novim koeficijentom nakon svake faze obrade.

Na prethodno prikazanom primeru prvi koeficijent u matrici predstavlja koeficijent pojačanja i njegova vrednost je manja od Δ za tip formata 1.31. Obrada predstavlja filtriranje IIR filterom reda 2N realizovanog upotrebom N filtera drugog reda. Kao pojedinačne faze obrade uzimaju se sekcije drugog reda. S obzirom da je svaka od faza obrade linearna, koeficijent pojačanja se može razložiti na N koeficijenata, gde svaki koeficijent ima vrednost N-tog korena iz originalnog koeficijenta.

$$g \cdot \prod_{k=1}^{N} F_k(x) = \prod_{k=1}^{N} \left(\sqrt[N]{g} F_k(x) \right)$$

U okviru samog koda potrebno je zameniti stari koeficijent pojačanja novom vrednošću koja je jednaka njegovom N-tom korenu, i premestiti operaciju množenja koeficijentom pojačanja nakon izvršenja svake od faza obrade kao što je prikazano na primeru ispod.

Primer:

```
for(n = 0; n < N; n++)
{
      ... // obrada
}

pOut = pOutBuf;
for (j=0 ; j < M j++ )
      *pOut++ = *pOut * gain;
```

Kod nakon izmene (*gain* = N-ti koren od originalne vrednosti *gain*):

```
for(n = 0; n < N; n++)
{
      ... // obrada

      for (j=0 ; j < M j++ )
            *pOut++ = yn * gain;
}
```

3.2.3.3 Prekoračenje opsega prilikom akumulacije brojeva u aritmetici nepokretnog zareza

Akumulatorski registri DSP procesora čuvaju međurezultate MAC instrukcija, kao i drugih aritmetičkih operacija. Ovi registri su po pravilu veći od dužine reči množača (2 x n) za m zaštitnih bita, ali pored toga može da dođe do prekoračenja maksimalne vrednosti koja može da se smesti u akumulator.

Kako bi se ovo izbeglo, moguće je uvesti bezbednosni opseg (*headroom*), odnosno, skalirati vrednosti signala na ulazu u blok odbrade (na opseg manji od maksimalne veličine tipova za podatke), a na izlazu iz bloka izvršiti inverzno skaliranje. Na primer, na ulazu u blok obrade, svi odbirci se podele sa 64 (aritmetički pomeraj u desno za 6), a nakon obrade, vrednosti izlaznog odbirka se pomnože sa 64. Na ovaj način opseg akumulatora se povećava za 6 bita, ali dolazi do gubitaka 6 bita preciznosti.

Primer:

```
... //Ucitavanje ulaznih odbiraka
for(j = 0; j < N; j++)
{
      *pIn = *pIn >> 6;
      *pIn++;
}

... // obrada

for(j = 0; j < N; j++)
{
      *pOut = *pOut << 6;
      pOut++;
}
... //Zapis izlaznih odbiraka
```

Važno je razumeti da se ovakve modifikacije ne smeju primenjivati bez analize opsega vrednosti i efekta na izlaznu tačnost. Poboljšanje tačnosti u jednom delu koda može da dovede do smanjenja tačnosti u drugom delu obrade.

3.2.4 Model 3 – Prevođenje koda kompajlerom za ciljnu arhitekturu

U ovoj fazi razvoja koda uvodi se namenski DSP kompajler, pa modifikacije podrazumevaju prlilagođenje koda specifičnostima C kompajlera za ciljni procesor.

Prvo se napravi simulatorski projekat koristeći CLIDE razvojno okruženje, pa se u njega dodaju datoteke sa izvornim kodom Modela 2. Deo koda zadužen za čitanje i pisanje vrednosti u datoteku potrebno je izmeniti tako da se koriste funkcije koje su deo standardne biblioteke za ciljnu platformu. Zatim se pristupa otkanjanju grešaka koje prijavljuje CCC kompajler. To su najčešće greške zbog razlike u podržanim C standardima između CCC i opštenamenskog prevodioca. Svim globalnim promenljivama potrebno je pridružiti kvalifikator memorijske zone.

Nakon što je kod uspešno preveden i daje ispravne rezultate kada se izvršava na DSP simulatoru, pristupa se proceni utroška resursa (profajlingu) i optimizaciji koda. Prvi korak je primena tehnika optimizacija na C kod. Nakon toga se delovi koda koji su i dalje neefikasni (troše najviše resursa), optimizuju na nivou asemblerskog koda. Dva moguća pristupa su:

1. Korišćenje ugrađenih asemblerskih instrukcija (*inline asm*) u okviru C koda.
2. Izdvajanje zahtevnih delova koda u zasebne funkcije i implementacija čitavih funkcija u asemblerskom jeziku. Pri ovom koraku potrebno je voditi računa o pozivnoj konvenciji, s obzirom da će ove funkcije biti pozivane iz C koda.

Saveti za optimizaciju i primeri optimizovanog koda dati su u poglavljima praktikuma koji sadrže opis arhitekture procesora i asemblerskih instrukcija i poglavlju koje sadrži opis CCC kompajlera.

Nakon optimizacije, kod se integriše sa sistemskim softverom za platformu CS48x.

3.3 Ispitivanje i verifikacija DSP aplikacije

Na kraju svakog modula je neophodno proveriti ispravnost rešenja, tako što se izlazni vektori modula porede sa izlaznim vektorima Modela 0. Tipičan scenario ispitivanja ispravnosti aplikacije prikazan je na slici 3.2.

Skup ulaznih vektora je veoma često deo referentnog softverskog paketa (Modela 0). Ovi vektori se formiraju tako da se ispita svaka grana obrade, pa je važno ispitivanje izvršiti nad celim skupom ulaznih vektora. Poređenje izlaznih rezultata modifikovanog koda sa referentim, može da se izvede na nekoliko načina: slušnim testovima, poređenjem datoteka na nivou bita i poređenjem spektralnih slika.

Slika 3.2 - Tipičan scenario ispitivanja

3.3.1 Slušni testovi

Slušni testovi spadaju u kategoriju subjektivnih metoda ispitivanja i formalnom smislu zahtevaju veliki broj obučenih slušalaca u kontrolisanom okruženju. Čujna razlika između originalnog signala i signala koji se ispituje, posmatra se kao oštećenje i ocenjuje se u skladu sa skalom **ITU-R BS.1284 standarda** (SDG je skraćenica od *Subjective Distortion Grade*, odnosno, stepen subjektivnog izobličenja):

Tabela 3.1 – Skala ITU-R BS.1284 standarda

Subjektivni kvalitet signala	*Ocena*	*SDG*	*Izobličenje*
Veoma dobar	5.0	0	Nečujno
Dobar	4.0	-1.0	Čujno ali ne smeta
Srednji	3.0	-2.0	Malo smeta
Slab	2.0	-3.0	Smeta
Loš	1.0	-4.0	Veoma smeta

Za potrebe izrade vežbi i ispitnih zadataka pod slušnim testovima podrazumeva se preslušavanje ispitnih testova i provera čujnih nepravilnosti - distorzija u odnosu na referenti izlaz. Ovaj pristup omogućava brzo i efikasno uočavanje grešaka prepoznatljivih ljudskom uhu.

3.3.2 Test identičnosti na nivou bita

Drugi način ispitivanja rezultata je test identičnosti na nivou bita. Ovo ispitivanje se vrši tako što se ispitna i referentna datoteka porede na nivou binarnog zapisa.

U okviru opisane metodologije razvoja aplikacija, identičnost rezultata na nivou bita očekuje se između Modela 0 i Modela 1, i između Modela 2 i Modela 3. Treba imati u vidu da promena aritmetike prilikom prelaska iz Modela 1 u Model 2 dovodi do promene preciznosti (npr. kao posledica zaokruživanja). U zavisnosti od algoritma koji se implementira, može se uvesti tolerancija greške, ukoliko ta greška ne utiče na krajnji rezultat rada sistema.

Poređenje datoteka na nivou bita može se izvršiti na više načina. Jedan od načina jeste upotrebom *Total Commander* alata, ili bilo kog uređivača binarnih datoteka (npr. *HexEditor*). Za potrebe poređenja datoteka na nivou bita, u okviru ove vežbe je dat i jednostavan alat *PCMCompare.* Ovaj alat se poziva iz komandne linije, sa sledećim parametrima:

- *PCMCompare* <Datoteka1> <Datoteka2> [dodatne opcije]

Dodatne opcije su:

- -b<16|24|32> - broj bita po odbirku (podrazumevano: 24)
- -d<broj odbiraka>
- -o<broj odbiraka>
- -i<broj odbiraka>
- -w - zanemari WAV zaglavlje

```
c:\tools>PCMCompare.exe Freq_sweep_2_0_0.wav Freq_sweep_2_0_02.wav -b16 -w
  Max difference is 26 (5 bits, -68.03dB)
  Max difference is 26 (5 bits, -68.03dB)
  8360103 samples compared
 ------------------------------------------------------------
  Dif(bits) |    Samples    |   PERCENT   |   First dif
 ------------------------------------------------------------
       1    |          1    |    0.00%    |  0x000006b4
       5    |          1    |    0.00%    |  0x000006cc
 ------------------------------------------------------------
    Error   |          2    |    0.00%
```

Slika 3.3 - Rezultat poređenja dve datoteke PCMCompare alatom

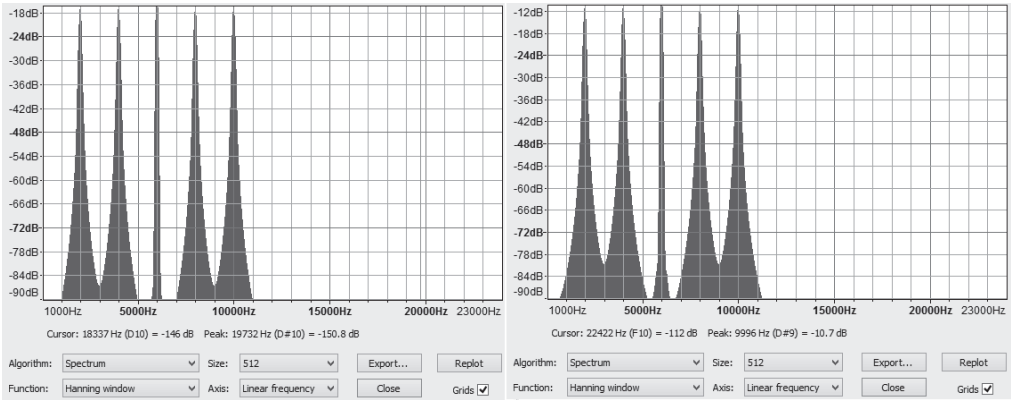
Rezultat poređenja dat je u formi tabele. Primer rezultata dat je na slici 3.3. Ukoliko je jedna datoteka kraća od druge, poređenje će se prekinuti kada se stigne do kraja kraće datoteke i o tome će biti ispisano obaveštenje. Među rezultatima poređenja data je najveća razlika između dva odbirka predstavljena kao vrednost razlike, broj bita i vrednost razlika predstavljena u decibelima, zatim broj poređenih odbiraka, i tabela koja sadrži redom: broj bita razlike, broj odbiraka koji sadrže grešku, procenat odbiraka u odnosu na sve odbirke u datotekama i adresu prve pojave greške. Na dnu tabele nalazi se ukupan broj odbiraka sa greškom. Na datom

primeru na slici 3.3, vidimo da je upoređeno ukupno 8360103 odbirka, da razlika postoji na ukupno dva, i to 1 bit razlike na adresi 0x000006b4 i 5 bita razlike na adresi 0x00006cc.

3.3.3 Spektralni testovi

Ispitivanje vektora u spektralnom domenu se primenjuje kada pretpostavljamo da razlike u izlaznim vektorima na nivou bita nisu posledica greške, već smanjene tačnosti. Spektralna analiza može se izvršiti upotrebom alata *Audacity*. Potrebno je otvoriti ispitne i referente vektore unutar *Audacity* alata i prikazati spektralnu karakteristiku signala odabirom opcije:

- *Analyze->Plot Spetrum*.



Slika 3.4 – Prikaz spektralne karakteristike referentnog i izlaznog signala koristeći alat Audacity

Za spektralnu analizu visoke preciznosti izlaznih signala sa razvojne ploče u realnom vremenu postoje razni alati i uređaji. Primer ovakvog uređaja je *Audio Precision Audio Analzyer* (http://ap.com/products/2700).

3.3.3.1 Ispitivanje izlaznih vektora na sistemima u realnom vremenu

Ispitivanje ispravnosti sistema u realnom vremenu je značajno složenije od ispitivanja rada aplikacije na simulatoru. Ako pojednostavimo, sistem u realnom vremenu radi u beskonačnoj petlji i stalno osluškuje signale na ulazu. Ako ispitni signal nije pušten, sistem će detektovati i obraditi tišinu ili neke vrednosti koje su zaostale u ulazno-izlaznoj memoriji. S druge strane, kod koji se izvršava na simulatoru podrazumeva da obrada počinje na početku prvog bloka obrade. Iz ovoga proizilaze dva problema:

- Nemogućnost poravnavanja signala na nivou bloka/odbirka, odnosno, obrada se neće vršiti nad istim podacima pa samim tim i rezultati nisu uporedivi
- Različito početno stanje sistema u realnom vremenu u odnosu na simulatorsko rešenje, što opet dovodi da različitih rezultata obrade (ukoliko rad sistema zavisi od prethodnih ulaznih odbiraka)

Sihnronizacija se obezbeđuje modifikacijom ispitnog signala, odnosno, uvođenjem sinhronizacione reči pre prvog odbirka ispitnog signala. Pored ovoga, potrebno je dodati sinhronizacioni kod u okviru softverskog rešenja za ciljnu platformu. Početno stanje aplikacije na ciljnoj platformi je čekanje na sinhronizacionu reč. Kada se detektuje sinhronizaciona reč, prvi odbirak nakon sinhronizacione reči je potrebno poravnati na početak prvog bloka obrade. Tek nakon toga se prelazi u stanje u kome se vrši obrada signala. Na ovaj način omogućavamo poređenje rezultata aplikacije na fizičkoj platofrmi sa rezulatima na simulatoru.

3.3.4 Automatizacija ispitivanja

Ispitivanje aplikacije, tako što se sekvencijalno izvršava velik broj ispitnih vektora, dugotrajan je posao. Složene audio tehnologije zahtevaju izvršavanje i više hiljada ispitnih slučajeva u procesu verifikacije rada implementiranog rešenja. Isto tako, nakon svake izmene u kodu, ispravke greške ili optimizacije, neophodno je ponoviti ispitivanje nad celim skupom ulaznih vektora. Ovaj proces je veoma spor, pa se teži ka automatizaciji procesa ispitivanja.

Najčešće se za automatizaciju ispitivanja koriste skript jezici: za pokretanje faza ispitivanja, podešavanje parametara i ulaznih vektora, kao i za zapis rezultata u odgovarajuću datoteku koja predstavlja izveštaj o uspehu ispitivanja. Primer skript jezika koji se može koristiti za automatizaciju jeste *batch* jezik. *Batch* datoteke predstavljaju tekstualne datoteke koje sadrže komande koje mogu biti interpretirane od strane Windows ili DOS komandne linije.

Većina alata koji se nalaze u sklopu CLIDE razvojnog okruženja se mogu pokrenuti i neposrednim pozivom iz komandne linije. Detalji oko pokretanja procesa prevođenja projekata iz komandne linije mogu se pronaći okviru *Clide->Help* menija. Za pokretanje izvršavanja projekta iz komandne linije potrebno je pokrenuti simulator sa odgovarajućim parametrima. Izvršna datoteka za pokretanje simulatora nalazi se na putanji

- `<sdkDir>/bin/cmdline_simulator.exe.`

Poziv simulatora vrši se pozivom:

- `cmdline_simulator.exe -project` *`filename`* [*`options`*]

gde opcije predstavljaju:

- `-max_cycles` *`###`* – simulacija će trajati maksimalno ###instrukcijskih ciklusa, nakon toga će biti prekinuta.
- `-silent` – zabrana emitovanja poruka poslatih od strane simulatora osim poruka koje sam program ispisuje na standardni konzolni izlaz.

Projektna datoteka koja se prosleđuje simulatoru kao glavni argument predstavlja XML datoteku sa konfiguracijom simulatora. Datoteka ima *.sbr* ekstenziju. Ova datoteka može se napisati ručno, ili generisati od strane CLIDE-a kada se pokreće *Slave Boot* proces nad simulatorskim (*Standalone ULD*) projektom. Struktura simulatorske konfiguracione datoteke data je sa:

```
< CL_PROJECT >
  <MEMORY_CFG>
    <ULD_FILE disk_path="..." />
    ...
  </MEMORY_CFG>
  <argv>  arg1 arg2 ... </argv>
  <FOPEN_VARS variable="varname" replacement_value="value" />
  ...
  <FILE_IO_CFG block_type="Input|Output"
        channels_per_line="n" index="i"
        sample_size="16|24|32"
sample_format="BigEndian|LittleEndian"
        justification="Left|Right" file_mode="PCM|WAVE|ASCII"
        sample_rate="Fs" >
        <FILE disk_path="..." channel="portAddress" />
    ...
  </FILE_IO_CFG>
  <SCP_CFG load_scp="clock cycles">
    <FILE disk_path="..." />
    ...
  </SCP_CFG>
   < PROFILE_CFG enable="on"|"off" />
< /CL_PROJECT>
```

`<MEMORY_CFG>` oznaka omogućava odabir jedne ili više ULD datoteka koje će biti učitane u simulator nakon pokretanja. ULD datoteka predstavlja izlaznu datoteku procesa prevođenja projekta i može se naći unutar /output/standalone direktorijuma. Ova sekcija je obavezna.

<argv> omogućava popunjavanje argumenta komandne linije koji će biti prosleđen *main* funkciji ukoliko je aplikacija pisana upotrebom programskog jezika C. Ova sekcija može da postoji ali nije obavezna.

Simulator podržava dva načina čitanja i pisanja u spoljne datoteke:

- preko *stdio.h* funkcija iz programskog jezika C,
- simulacijom hardverskog pristupa periferijskim jedinicama koristeći *inp* i *outp* operacije.

Ukoliko se koriste funkcije za standardan ulazi i izlaz, moguće je unutar *fopen* funkcije prilikom navođenja naziva datoteke koristiti promenljive u formi $(naziv_promenljive), a unutar konfiguracione datoteke simulatora tim promenljivim dodeliti vrednosti koristeći <FOPEN_VARS> oznaku. Ukoliko se koriste *inp* i *outp* za čitanje i pisanje vrednosti, adrese korišćene kod ovih instrukcija mapiraju se na odgovarajuće datoteke zadate koristeći <FILE> oznaku unutar <FILE_IO_CONIG> sekcije. Za simulator jednog jezgra polje "index" uvek treba da je 0. <FILE> sekcija sadrži putanju do datoteke i aderesu na koju se ta datoteka mapira. Primer korišćenja mapirane datoteke u kodu:

- x0 = inp[0x3800]; -> čitanje narednog odbirka iz datoteke koja je mapirana na prvi kanal u registar x0,
- outp[0x3400]=x0; -> pisanje sadržaja registra x0 u izlaznu datoteku mapiranu na adresu 0x3400.

Tabela 3.2 sadrži listu adresa preporučenih za mapiranje ulaznih i izlaznih datoteka.

Tabela 3.2 – Lista adresa za mapiranje ulaznih i izlaznih datoteka

Kanal	Adresa za ulaznu datoteku	Adresa za ilaznu datoteku
0	0x3800	0x3400
1	0x3801	0x3401
2	0x3802	0x3402
3	0x3803	0x3403
4	0x3804	0x3404
5	0x3805	0x3405

Oznaka <SCP_CFG> omogućava simulaciju slanja komandnih poruka od stane spoljašnjeg uređaja (npr. mikrokontrolera). Polje load_scp govori nakon koliko ciklusa poruka treba biti poslata. <FILE> oznaka sadrži putanju do datoteke sa komandnim porukama koje je potrebno poslati.

Oznakom <PROFILE_CFG> određeno je da li je uključena podrška za profilisanje koda tokom izvršavanja na simulatoru. Potrebno je voditi računa da su sve putanje relativne u odnosu na datoteku simulatorskog projekta.

Na slici 3.5 dat je primer *batch* datoteke u okviru koje je realizovano automatsko ispitivanje rezultata izvršenja jedne aplikacije razvijene po opisanoj metodologiji. U okviru datog primera realizovan je poziv izvršenja programskog koda modela 0, 1, 2 i 3, i snimanje odgovarajućih izlaznih datoteka. Nakon toga vršen je poziv *PCMCompare* alata za poređenje datoteka na nivou bita. Izveštaj svakog poređenja upisan je u odgovarajuću tekstualnu datoteku koja predstavlja izveštaj poređenja.

```
: Delete log files first.
del cmp_Model0_vs_Model1.txt
del cmp_Model1_vs_Model2.txt
del cmp_Model2_vs_Model3.txt

: Execute Model 0, Model 1 and Model 2
cd model_0\Debug
"filter Model0.exe"
cd ..\..

cd model_1\Debug
"filter Model1.exe"
cd ..\..

cd model_2_3\Debug
"filter Model2.exe"
cd ..\..

cd model_2_3\
c:\CirrusDSP\bin\cmdline_simulator.exe -project SimulatorConfigurationTemp.sbr -max_cycles 1000000
cd ..

: Generate new logs
tools\PCMCompare.exe out_intl_model0.pcm out_intl_model1.pcm -b16 2> cmp_Model0_vs_Model1.txt
tools\PCMCompare.exe out_intl_model1.pcm out_intl_model2.pcm -b16 2> cmp_Model1_vs_Model2.txt
tools\PCMCompare.exe out_intl_model2.pcm out_intl_model3.pcm -b16 2> cmp_Model2_vs_Model3.txt

```

Slika 3.5 - Primer Batch skripte za poređenje rezultata izvršenja

3.4 Zadaci za samostalnu izradu

3.4.1 Zadatak 1: Primer realizacije DSP aplikacije upotrebom metodologije razvoja zasnovane na C kompajleru

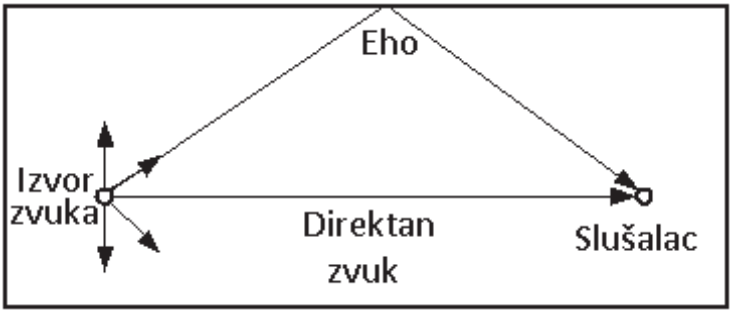
U okviru prvog zadatka na priloženom primeru upoznaje se sa procesom realizacije DSP aplikacije upotrebom metodologije razvoja zasnovane na C kompajleru. Implementirani algoritam obrade je niskopropusni filtar sa beskonačnim odzivom 10-og reda, realizovan kaskadnom vezom 5 filtera drugog reda.

3.4.1.1 Postavka zadatka:

1. Koristeći razvojno okruženje Visual Studio pokrenuti projekat lowpass_float_to_fixed\model_0\. Ovaj projekat sadrži referentni kod zadatog algoritma, odnosno Model 0.
2. Izvršiti analizu referentnog koda. Prevesti i izvršiti kod.
3. Koristeći razvojno okruženje Visual Studio pokrenuti projekat lowpass_float_to_fixed\model_1\. Ovaj projekat predstavlja Model 1 odnosno kod nakon uvođenja funkcionalnih optimizacija.
4. Analizirati kod Modela 1. Uočiti sve izmene u odnosu na Model 0. Prevesti i izvršiti kod.
5. Koristeći razvojno okruženje Visual Studio pokrenuti projekat lowpass_float_to_fixed\model_2_3. Ovaj projekat predstavlja Model 2, odnosno kod nakon prilagođenja aritmetici nepokretnog zareza.
6. Analizirati kod Modela 2. Uočiti sve izmene u odnosu na Model 1. Posebnu pažnju posvetiti analizi klasa korišćenih za emulaciju tipova u nepokretnom zarezu.
7. Prevesti i izvršiti kod.
8. Koristeći razvojno okruženje CLIDE pokrenuti projekat lowpass_float_to_fixed\model_2_3\.
9. Prevesti i izvršiti kod.
10. Izvršiti poređenje izlaznih datoteka iz Modela 0 i Modela 1, i potom Modela 1 i Modela 2 upotrebom *PCMCompare* alata.
11. Otvoriti datoteku *Test.bat* koja služi za automatizovano izvršenje različitih modela, poređenje rezultata i generisanje izveštaja. Analizirati njen sadržaj.
12. Izvršiti *Test.bat* skriptu i uporediti rezultate sa dobijenim razultatima u koraku 10.
13. Izvršiti spektralno poređenje izlaznih datoteka iz Modela 1 i Modela 2.

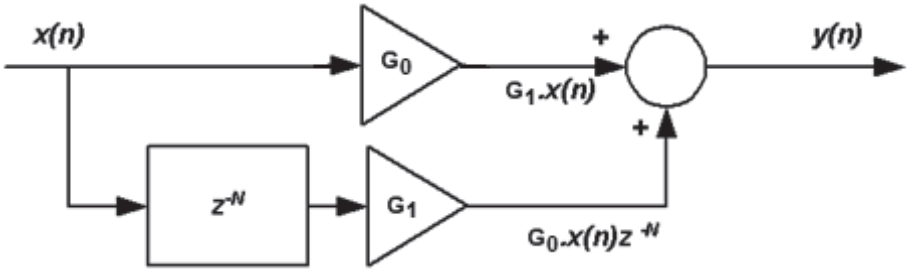
3.4.2 Zadatak 2: Primena metodologije razvoja aplikacija zasnovane na C kompajleru na realizaciju bloka za dodavanje višestrukog eho efekta audio signalu

U ovom projektnom zadatku potrebno je realizovati modul za proizvodnju eho efekta. Eho efekat nastaje kada do slušaoca, pored zvuka koji dopire neposredno od izvora zvuka, dopire i zvučni signal koji se odbio od određene prepreke.



Slika 3.6 - Eho efekat

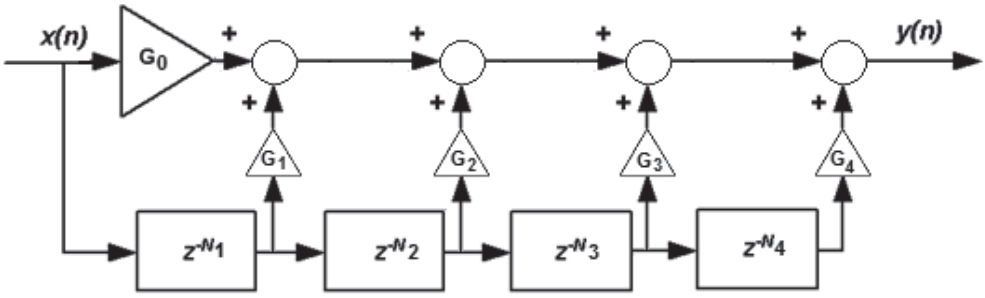
Pojednostavljeni modul za proizvodnju Eho efekta funkcioniše tako što zakasneli odbirak signala, pomnožen određenom vrednošću pojačanja eho odbirka G1, dodajemo na trenutni ulazni odbirak pomnožen sa pojačanjem G0 . Zbir ova dva odbirka predstavlja vrednost izlaznog odbirka.



Slika 3.7 – Blok dijagram sistema za dodavanje eho efekta

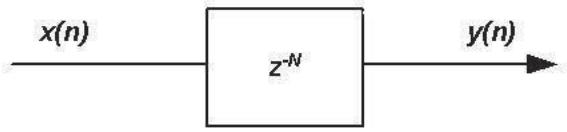
U slučaju kada se želi simulirati efekat da se zvuk odbija od više različitih prepreka, formira se sistem koji se sastoji od većeg broja jedinica za kašnjenje signala. Svaki zakasneli odbirak ima svoj koeficijent pojačanja. Da bi se očuvalo pojačanje signala, suma svih koeficijenata pojačanja bi trebalo da bude jednaka 1. Potrebno

je voditi računa da je velika verovatnoća da će doći do prekoračenja ukoliko suma koeficijenata prelazi 1.



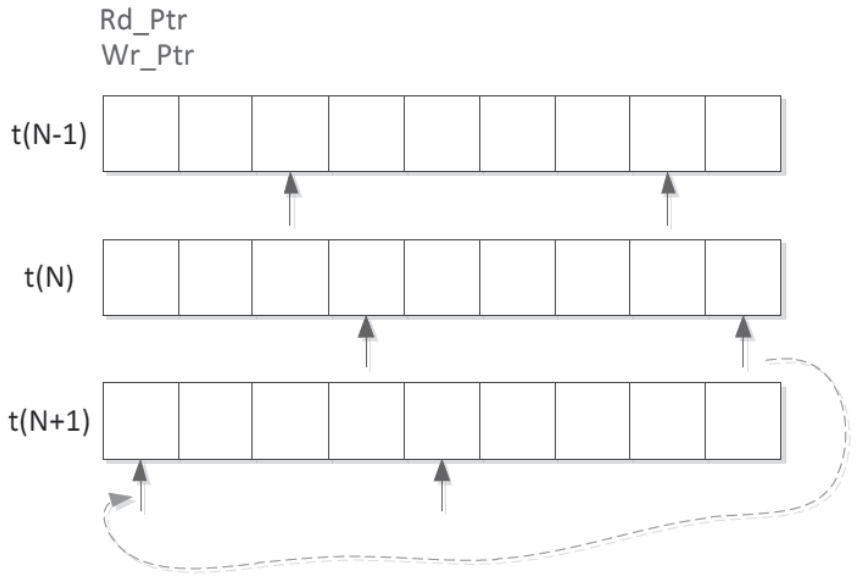
Slika 3.8 – Blok dijagram sistema za dodavanje višestrukog eho efekta

Osnovni gradivni blok eho efekta je modul za vremensko kašnjenje. Modul za vremensko kašnjenje signala (eng. *delay*) predstavlja linearan vremenski invarijantan sistem sa memorijom. Vrednost izlaznog signala u momentu n jednak je vrednosti ulaznog signala u momentu *(n-N)*.



Slika 3.9 - Modul za vremensko kašnjenje

Realizacija ovog modula podrazumeva pamćenje prethodnih N ulaznih odbiraka u memoriji sistema. Moduli za kašnjenje se efikasno implementiraju upotrebom kružnog (cirkularnog) buffer-a. Cirkularni buffer se sastoji iz memorijskog niza željene dužine i pokazivača za upis i čitanje. Najnoviji odbirci se upisuju na poziciju na koju pokazuje pokazivač za upis (na slici *Wr_Ptr*) , stari (ranije upisani) odbirci se čitaju preko pokazivača za čitanje (na slici *Rd_Ptr*). Kada jedan od pokazivača dostigne kraj memorijskog niza on se vraća na početak i odatle nastavlja svoju progresiju. Na procesorima opšte namene, i jezicima visokog nivoa (C/C++) ovo se postiže modulo operatorom (%) nakon inkrementiranja indeksa. Ovakvo rešenje kod RISC procesora je veoma neefikasno zbog toga što se ovaj operator zasniva na deljenju koje najčešće nije podržano na ovakvim procesorima. Arhitektura DSP procesora CS48x hardverski podržava adresiranje po modulu, što omogućava veoma efikasnu implementaciju cirkularnih memorijskih nizova



Slika 3.10 – Prikaz bloka za kašnjenje

3.4.2.1 Postavka zadatka:

U okviru projekta *multitapEcho_model0* dat je referentni kod sistema za generisanje višestrukog eho efekta. Sve informacije o aktuelnom stanju sistema sadržane su u okviru strukture *EchoState*. Ova struktura sadrži sledeće podatke: pokazivač na memorijski niz za skladištenje zakasnelih odbiraka, dužina pomenutog memorijskog niza, indeks za upis ulaznih odbiraka u niz, indekse za čitanje vrednosti iz niza za svaku jedinicu kašnjenja u sistemu, vrednosti kašnjenja za svaku jedinicu kašnjenja, pojačanje ulaznog odbirka (G0), pojačanja svakog od zakasnelih odbiraka (G1..Gn), i broj jedinica za kašnjenje u sistemu.

Sama obrada sastoji se iz dve funkcije:

- `void multitap_echo_init(EchoState* echoState, double* buffer, const int echoBufLen, const int delay[], const double input_gain, const double tap_gain[], const int n_tap)`
- `void multitap_echo(const double *pInbuf, double *pOutbuf, int inputLen, EchoState* echoState)`

Prva funkcija služi za inicijalizaciju sistema. Kao parametre prima instancu strukture koja predstavlja trenutno stanje sistema i koju je potrebno inicijalizovati i podatke potrebne za inicijalizaciju.

Druga funkcija predstavlja obradu, odnosno dodavanje eho efekta. Unutar ove funkcije prolazi se kroz ulazni blok podataka i svaki ulazni odbirak se smešta na

trenutnu poziciju pisanja u sistemu i vrši se uvećanje indeksa pisanja po modulu. Ulazni odbirak se množi sa ulaznim pojačanjem i smešta u izlazni niz na odgovarajuću poziciju. Zatim sledi iteracija kroz sve sisteme za kašnjenje i vrši se čitanje zakasnelog odbirka, množenje odgovarajućim koeficijentom pojačanja i dodavanje na trenutnu vrednost izlaznog odbirka. U okviru svake iteracije vrši se i uvećanje indeksa čitanja po modulu.

3.4.2.1.1 Analiza referentnog koda

1. Otvoriti projektni zadatak *multitapEcho_model0* upotrebom *Visual Studio* okruženja.
2. Izvršiti analizu referentnog koda.
3. Prevesti i izvršiti projekat.
4. Poslušati izlaznu datoteku i uporediti sa ulaznom.

3.4.2.1.2 Model 1 – Funkcionalne optimizacije

1. Napraviti novi projekat u okviru istog rešenja (*soulution*) i dodeliti mu naziv *multitapEcho_model1.* Iskopirati datoteke sa izvornim kodom iz Modela 0.
2. Izmestiti strukturu stanja tako da bude globalna.
3. Izmestiti konstante za inicijalizaciju tako da budu globalne.
4. Izbaciti sve argumente funkcije za inicijalizaciju.
5. Smanjiti broj argumenata funkcije *multitap_echo* tako da sadrži samo pokazivače na ulazni i izlazni niz.
6. Zameniti pristup elementima nizova na mestima gde se koristi indeksiranje uvođenjem pristupa preko pokazivača.
7. Prepoznati delove koda koji se nalaze u petlji, a ne zavise od same petlje ukoliko postoje i izmestiti ih van petlje.
8. Uvesti *common.h* (iz prethodnog zadata) zaglavlje koje olakšava prelazak sa Modela 1 na Model 2. Zameniti *double, int* i *short* tipove definisanim *DSPfract, DSPint* i *DSPshort* u delu koda koji sadrži obradu.
9. Nakon svakog od koraka prevesti kod, izvršiti i uporediti rešenje sa izlazom iz referentnog modela. Za potrebe poređenja prilagoditi *Test.bat* skriptu iz prethodnog zadatka promenom naziva izvršnih datoteka koje se pokreću.

3.4.2.1.3 Model 2 – Funkcionalne optimizacije

1. Napraviti novi projekat u okviru istog rešenja (*soulution*) i dodeliti mu naziv *multitapEcho_model2.* Iskopirati datoteke sa izvornim kodom iz Modela 1.
2. Dodati u projekat datoteke koje sadrže emulacione klase za tipove u nepokretnom zarezu: *fixed_point_math.h, fixed_point_math.cpp, stdfix_emu.h* i *stdfix_emu.cpp.*
3. Izvršiti zamenu tipova podataka u pokretnom zarezu emulacionim klasama.

4. S obzirom da su u pitanju klase, a ne prosti tipovi, zameniti inicijalizaciju globalnih nizova tako da umesto *memset* funkcije koriste dodelu vrednosti FRACT_NUM(0.0) svakom elementu.
5. Izvršiti pretvaranje konstanti u nepokretnom zarezu upotrebom FRACT_NUM direktive.
6. Voditi računa o tipovima podataka prilikom upisa izlaznih odbiraka u datoteku. Umesto množenja sa vrednosti maksimalne celobrojne vrednosti iskoristiti metodu *toLong()* koju poseduju emulacione klase.
7. Prevesti kod, izvršiti i uporediti rešenje sa izlazom iz referentnog modela. Dopuniti *Test.bat* skriptu iz prethodnog zadatka tako da se vrši poziv i Modela 2.w