SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1236

SUSTAV ZA DIREKTNU DIGITALNU SINTEZU SINUSA KORIŠTENJEM DSP PROCESORA

Luka Dejanović

Zagreb, lipanj 2010.

STRANICA NAMJERNO PRAZNA

UMJESTO OVE STRANICE DOLAZI TEKST ZAVRŠNOG ZADATKA

Sadržaj:

1. Uvod

Čest je slučaj kada je potrebno producirati i jednostavno kontrolirati točne valne oblike raznih frekvencija. Primjene su razne: od komunikacijskih primjena za agilne izvore s niskim šumom te izvrsnim podešavanjem frekvencije i odličnim spektralnim performansama, do industrijskih i biomedicinskih upotreba gdje su potrebni valni oblici kojima se lako podešavaju frekvencija i faza, i to bez potrebe za promjenom vanjskih komponenata.

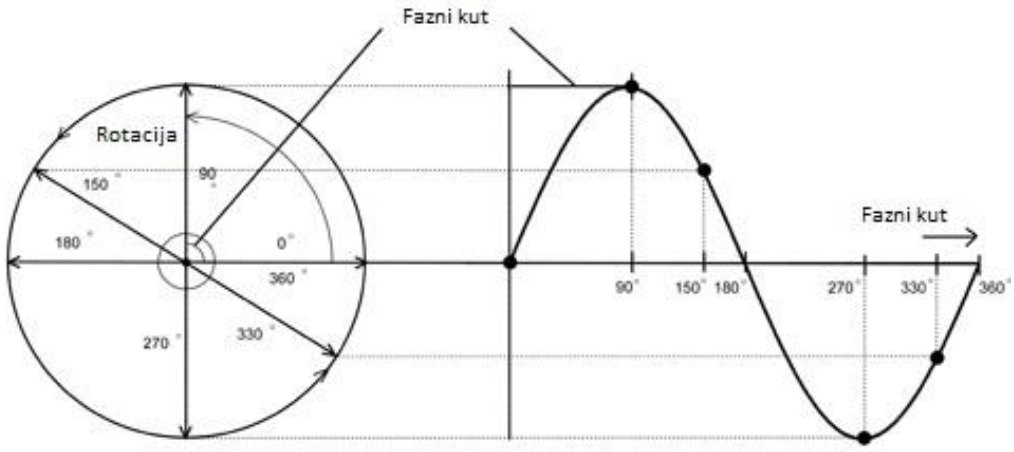
Isprobani su različiti pristupi, a najfleksibilnijim se pokazao postupak diretkne digitalne sinteze (direct digital synthesis – DDS). Ovim se postupkom producira analogni valni oblik, najčešće sinusni, generiranjem vremenski promijenjivog valnog oblika te se on digitalno-analognom pretvorbom pretvara u potreban analogni signal. DDS postupkom se jednostavno i brzo skače s jedne izlazne frekvencije na drugu, bez problema sa brzinom porasta ili vremenom ustaljivanja signala, što su česte anomalije analognog podešavanja frekvencije. Jedna od najefikasnijih izvedbi DDS-a je pomoću procesora za digitalnu obradu signala (digital signal processor – DSP) te takozvanog *on-chip* D/A pretvornika.

Ovdje se prikazuje izvedba digitalnog sintetizatora korištenjem procesora tvrtke Analog Devices, porodice Blackfin, konkretno ADSP – BF537.

2. Direktna digitalna sinteza

Sklopovi za digitalnu sintezu valnih oblika na svom izlazu generiraju najčešće sinusni valni oblik stabilne frekvencije i velike točnosti njenog podešavanja. Koriste se direktne, indirektne i hibridne metode digitalne sinteze. Diretkna metoda digitalne sinteze ima mogućnost velike brzine promjene frekvencije i visoku frekvencijsku rezoluciju, te jednostavnu realizaciju modulacijskih postupaka. Kao glavni nedostaci DDS-a ističu se spektralna nečistoća i nedovoljan opseg frekvencija izlaznog signala.

Jedan od temeljnih dijelova DDS-a je generator takta. On upravlja radom sustava te uvelike određuje kvalitetu izlaznog signala. Izvode sa i bez množača frekvencije referentnog oscilatora (ove izvedbe imaju manja izobličenja), a važne značajke su mu frekvencijska stabilnost, podrhtavanje takta i fazni šum,.
Signali koji se sintetiziraju su najčešće sinusni iako se ovom metodom mogu realizirati proizvoljni periodički signali. Sinusna funkcija se može jednostavno opisati pomoću vektora koji rotira po faznoj kružnici, gdje jedan obilazak vektora po faznoj kružnici uz stalnu brzinu odgovara jednom periodu sinusoide, zbog činjenice da svaka točka fazne kružnice odgovara jednoj točci sinusoide.



Slika 1. Fazna kružnica i sinusna funkcija

Još jedan važan dio sustava za DDS je fazni akumulator, koji sadrži informaciju o rotaciji vektora oko fazne kružnice. Hardverski se fazni akumulator izvodi pomoću registra koji obavlja jednostavnu funkciju:

$$f(i) = f(i-1) + \Delta p$$

Broj diskretnih vrijednosti na faznoj kružnici je određen brojem bita faznog akumulatora, a povezani su relacijom:

$$N_t = 2^N$$

gdje je N_t broj točaka fazne kružnice, a N broj bitova faznog akumulatora. Razne izlazne frekvencije se dobiju odabirom vrijednosti parametra prirasta faze, tzv. faznog inkrementa (u prethodnoj relaciji, Δp ima ulogu faznog inkrementa), koji je za konstantnu frekvenciju konstantan. Budući da je izlaz faznog akumulatora linearan, tj. faza raste od 0 do 2π, pa se „premata“ natrag u nulu, taj se signal ne može direktno koristiti za dobivanje valnog oblika (osim za neku vrstu pilastog valnog oblika), te se zbog toga koriste pohranjeni koeficijenti (ROM *look-up* tablica), pomoću kojih se, adekvatnom metodom, interpolira traženi signal, u ovom slučaju sinusni. Adresiranje ovih uzoraka je indirektno, pri čemu određeni broj bitova faznog akumulatora koristi za adresiranje. Budući da se koristi samo dio cijelokupnog faznog akumulatora, nekoliko bitova, javljaju se pogreške pri konverziji faze u amplitudu. Ove pogreške ovise o broju bitova faznog akumulatora, broju bitova za adresiranje te o vrijednosti faznog inkrementa.

Prije samog algoritma generiranja sinusnog signala, potrebno je reći da se u slučaju sinusnog signala mogu iskoristiti simetrije samog valnog oblika: drugi kvadrant je zrcaljenje prvog u odnosu na ordinatu, dok su treći i četvrti zrcaljenje prva dva kvadranta u odnosu na apscisu. Iz toga proizlazi da je potrebno pohraniti samo koeficijente prvog kvadranta, a ostali se dobiju jednostavnim matematičkim operacijama (drugi se dobije obrnutim redoslijedom čitanja tablice za prvi kvadrant, treći dodavanjem negativnog predznaka na koeficijente prvog kvadranta, a četvrti

istim postupkom za koeficijente drugog kvadranta). Ovime se drastično smanjuju dimenzije tablice koeficijenata.

Kako je već rečeno, koeficijenti koji se pohranjuju u ROM tablicu koriste se za interpolaciju sinusne funkcije. Postoji nekoliko metoda interpolacije sinusa, a najpoznatije polinomne metode su:

- optimalno LMS (*Least Mean Square*) rješenje
- približno optimalno MAE (*Mean Absolute Error*) rješenje
- optimalno SFDR (*Spurious-Free Dynamic Range*) rješenje bez aliasinga
- kubične *spline*

Ovim se algoritmom dobije interpolacija sinusnog signala metodom kubičnih *spline-a*. Spline su polinomne funkcije koje lokalno imaju vrlo jednostavan oblik, dok su globalno glatke i fleksibilne, pa su kubične *spline* konstruirane pomoću polinoma trećeg stupnja koji prolaze kroz nekoliko kontrolnih točaka.

Zbog karakteristika procesora na kojem se algoritam izvodi, vrlo povoljan zapis ovog polinoma je Hornerova formula:

$$f(x) = ((d \cdot x + c) \cdot x + b) \cdot x + a$$

Algoritam je u potpunosti opisan poglavlju 4.1.1., uz prvotnu realizaciju uz korištenje pomične decimalne točke, uz *floating-point* realizaciju.

3. Procesor ADSP – BF537

3.1. Procesori za digitalnu obradu signala

Potreba za digitalnom obradom signala je u današnje vrijeme vrlo raširena. Koristi se u audio sustavima, automobilskoj industriji, osobnim računalima, moblinoj telefoniji, obradi slike ili videa i u mnogim drugim područjima. U početku su DSP procesori bili namjenjeni obradi visokofrekventnih signala, radio valova i mikrovalova, ali se ta primjena proširila na ostala područja u kojima se traži veliki broj proračuna. DSP se koriste kao dodatni procesori, glavna namjena im nije korištenje velikih blokova programa poput operacijskih sustava, kao što to rade mikroprocesori, već rade sa stvarnim signalima i u realnom vremenu pa je osnovna karakteristika brzina, tj. mogućnost brzog obrađivanja podataka. Realni signali su analogni, vremenski kontinuirani, pa nisu u tom obliku pogodni za digitalnu obradu, nego se prvo moraju korištenjem kvalitetnih (najbitnije svojstvo je dovoljno velika frekvencija uzorkovanja) analogno-digitalnih pretvornika transformirati u diskretne, digitalne signale.

3.2. Karakteristike procesora ADSP – BF537

3.2.1. Arhitektura i periferija

Blackfin procesori su porodica 16-bitnih ili 32-bitnih ugradbenih procesora koju su dizajnirani kako bi zadovoljili današnje računske zahtjeve te ograničenja u potrošnji energije u komunikacijskim te audio i video primjenama. Bazirani su na arhitekturi MSA (*Micro Signal Architecture*) koja je razvijena u suradnji s Intel-om, te kombiniraju set 32-bitnih instrukcija nalik RISC-u i dualnu 16-bitnu MAC (*Multiply and ACcumulate*) funkcionalnost sa jednostavnim atributima sličnim onima kod mikrokontrolera za opću upotrebu. Ova kombinacija omogućuje Blackfin procesorima performanse jednako dobre u domeni obradbe signala i u domeni općih aplikacija i time eliminiraju potrebu za odvojenim „glavnim“ procesorom. Ove mogućnosti uvelike pojednostavljuju i hardverske i softverske zadaće dizajna.

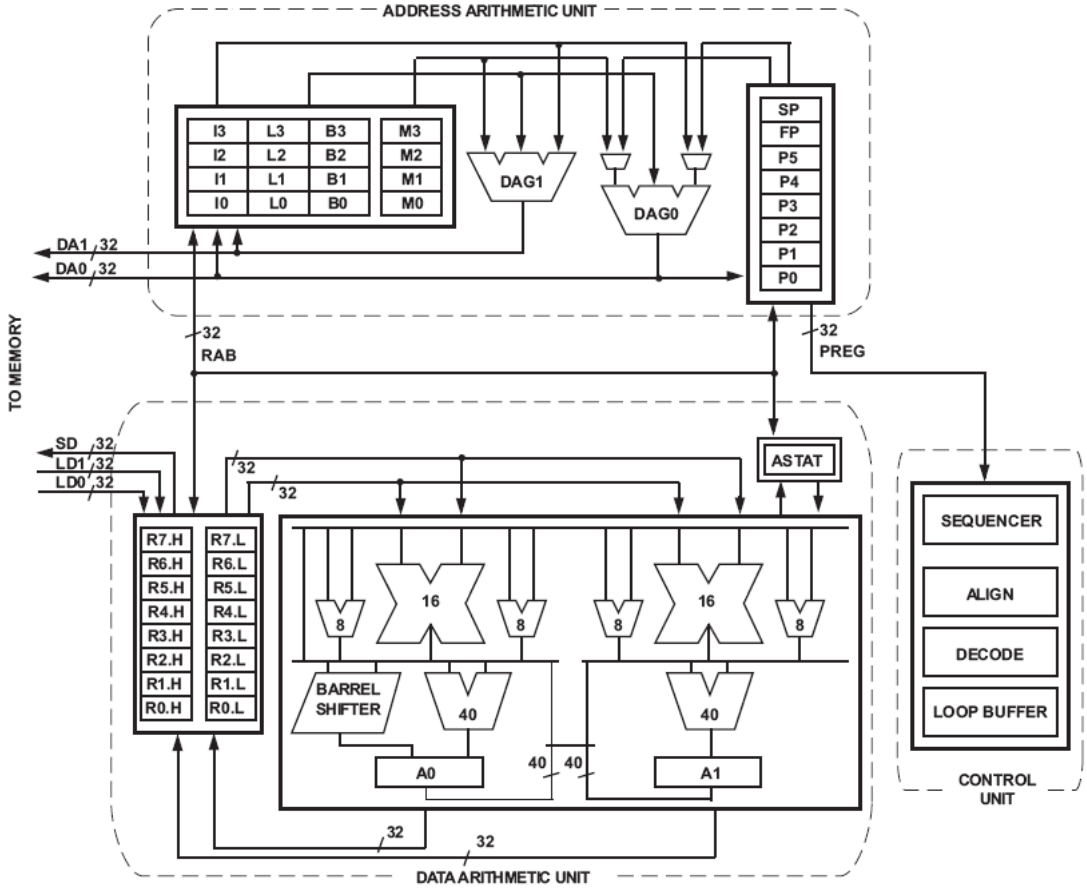
Arhitektura Blackfin procesora je bazirana na 10-stupanjskoj RISC MCU/DSP protočnoj strukturi, a 16/32-bitnim setom instrukcija dizajniranim za optimalnu gustoću kôda. Isto tako, arhitektura je potpuno SIMD kompatibilna i ima integrirane instrukcije za brzu obradu slike i video signala. Blackfin procesori imaju ugrađene višestruke, nezavisne DMA kontrolere koji omogućuju automatizirane transfere podataka uz minimalno opterećenje procesora. Ovakvi DMA transferi se mogu ostvariti između internih jedinica i bilo koje DMA-sposobne periferije ili eksternih uređaja, uključujući i SDRAM kontrolere i asinkrone kontrolere memorije. Uz nominalnu podršku za 8-bitne podatke, ovi procesori uključuju i set instrukcija napravljenih za poboljšanje performansi kod obrade video signala. Primjerice, diskretna kosinusna transformacija (DCT) je podržana sa IEEE 1180 operacijama zaokruženja, dok određene instrukcije podržavaju algoritme za estimaciju kretnji kakvi se koriste kod kompresijskih metoda kao npr. MPEG2, MPEG4, JPEG.

Također, procesori iz ove porodice imaju neke posebnosti koje se najčešće nalaze kod mikroprocesora, kao na primjer, 10/100 Ethernet MAC, UARTS, SPI, CAN kontrolere, timere sa PWM podrškom, Watchdog timer, Real-time sat i sinkroni i asinkroni kontroler memorije, čime je dizajneru omogućena velika fleksibilnost i niski završni troškovi.

3.2.2. Jedinice za računanje i kratki opis rada

ADSP-BF5xx procesori su primarno 16-bitni, cijelobrojni uređaji. Većina operacija pretpostavlja zapis brojeva u dvojnom komplementu, dok ostale pretpostavljaju brojeve bez predznaka ili jednostavne binarne slijedove. Postoje i instrukcije koje podržavaju 32-bitnu aritmetiku, ali i instrukcije koje rade 8-bitnu aritmetiku, te blokove decimalnih brojeva. Brojevi s predznakom su kod ovih procesora uvijek u dvojnom komplementu i instrukcije pretpostavljaju i podržavaju aritmetiku dvojnog komplementa, dok formati *sign-and-magnitude*, BCD (*Binary Coded Decimal*) te jedinični komplement nisu podržani. Binarni slijedovi su najjednostavniji binarni zapis, kod kojeg je 16 bitova tretirano kao niz. Operacija koje se vrše nad ovim tipom podataka su logičke: NOT, AND, OR, XOR. Ove ALU operacije ne prepoznaju bit predznaka ili poziciju binarne točke. Binarni brojevi bez predznaka se mogu tretirati kao pozitivni, i mogu imati dvostruko veću vrijednost od

brojeva s predznakom (*unsigned* – od 0 do 2^N-1, *signed* – od -2^{N-1} do 2^{N-1}-1). Procesor kod brojeva sa višestrukom preciznosti niže riječi tretira kao brojeve bez predznaka, (*unsigned*). Procesor ADSP-BF5xx je optimiziran za numeričke vrijednosti u frakcionalnom binarnom formatu opisanom u "jedan točka petnaest" notaciji, tj. 1.15 notaciji. U ovom formatu jedan bit predznaka, najteži bit (*MSB – Most Significant Bit*), i 15 frakcionalnih bitova opisuju brojeve u rasponu od -1 do 0.999969. Procesor podržava 32-bitne riječi, 16-bitne poluriječi i bajtove (8 bitova). Riječi i poluriječi mogu biti ili cijeli brojevi ili frakcije, dok su bajtovi isključivo cijeli brojevi. Cijeli brojevi mogu biti sa ili bez predznaka, a frakcije su uvijek sa predznakom. Neke instrukcije manipuliraju podatcima u registrima koristeći popunjavanje predznakom ili popunjavanje nulom, do 32 bita (tzv*. sign-extension* i *zero-extension*). Također, neke instrukcije dvije 16-bitne riječi ili četiri bajta gledaju kao 32-bitnu riječ. Kod 32-bitnih riječi, slovom H se označava gornjih, a sa L donjih 16 bitova. Kod 40-bitnog akumulatora, W označava donjih 32 bita, a X gornjih 8 bitova.



Slika 2. Arhitektura jezgre procesora ADSP – BF537

Kako je osnovna zadaća ovog i sličnih procesora računanje, on posjeduje jedinice za računanje koje provode operacije numeričke obrade kako općih algoritama kontrole, tako i DSP algoritama. Šest računskih jedinica su:

- dvije aritmetičko-logičke jedinice - ALU
- dvije jedinice za množenje i akumuliranje (zbrajanje) – MAC
- jedinica za posmak – shifter
- set video ALU jedinica

Jedan od najvažnijih dijelova arhitekture procesora je statusni registar, takozvani ASTAT registar (*Arithmetic status* registar) je 32-bitni registar ukazuje na zadnje operacije ALU jedinice, množača ili shifter-a.

Računske jedinice barataju raznim tipovima operacija. ALU jedinice vrše aritmetičke i logičke operacije. Množači obavljaju množenje te izvršavaju *množi-zbroji* i *množi-oduzmi* operacije. Jedinica za posmak provodi logičke i aritmetičke posmake, te pakiranje i ekstrakciju bitova. Video ALU jedinice provode tzv. SIMD (*Single Instruction – Multiple Data*) logičke operacije nad specifičnim 8-bitnim operandima. Računske jedinice procesora imaju tri različite grupe registara: *Data Register File*, *Pointer Register File*, i nekoliko *Data Address Generator* (DAG) registara. Registarski skup DRF se sastoji od osam 32-bitnih registara koji primaju podatke sa podatkovnih sabirnica, proslijeđuju ih računskim jedinicama i pohranjuju rezultate. Kod operacija koje rade sa 16-bitnim podatcima, registri se uparuju, čime je omogućeno korištenje 16 registara. Dva dodatna registra, A0 i A1 otvaraju mogućnost za 40-bitne akumulatorske rezultate. Ti su registri namijenjeni ALU jedinicama, i prvenstveno se koriste za *množi-i-akumuliraj* operacije. Asemblerski jezik samog procesora omogućuje pristup *Data Register File-u.* Sintaksa omogućuje programu da pomiče podatke i u isto vrijeme specificira format podataka za računanje. Jedinica za množenje, ALU, te *shifter* imaju neograničeni pristup podatkovnim registrima u *DRF-u*, dok multifunkcijske operacije mogu imati ograničenja. Tradicionali načini aritmetičkih operacija, kao na primjer cijelobrojni ili frakcionalni, su specificirani direktnu u instrukcijama. Zaokruživanje se obavlja u ovisnosti o ASTAT registru, koji pamti stanja i uvjete računskih operacija. PRF registri sadržavaju pokazivače za adresiranje operacija. DAG registri su namjenski registri koji upravljaju *zero-over-head* cirkularnim privremenim memorijama, (*buffer-ima*) za DSP operacije.

Za funkcije množenja i akumuliranja procesor nudi dvije opcije – frakcionalnu aritmetiku za frakcionalne brojeve, te cijelobrojnu aritmetiku za cijele brojeve. Kod frakcionalne aritmetike, 32-bitna izlazna riječ je modificirana na način da se proširi bitovima predznaka i posmakne za jedno mjesto ulijevo, prije nego se zbraja s akumulatorom. Tako se 31. bit produkta poravnava sa 32. bitom akumulatora (nultim bitom X-a), a nulti bit produkta se poravnava sa prvim bitom akumulatora (prvi bit W dijela riječi). Najniži bit, (*Least Significant Bit – LSB*) se puni nulom. Za cijelobrojnu aritmetiku, 32-bitni registar produkta se ne posmiče prije zbrajanja s akumulatorom. Kod bilo koje od ovih dviju aritmetika, izlazni produkt množača se ubacuje u 40-bitno zbrajalo/oduzimalo koji obavlja navedene operacije između tog produkta i trenutačnog sadržaja jednog od akumulatorskih registara kako bi se dobio krajnji 40-bitni rezultat.

Kod mnogih operacija množenja, procesor podržava zaokruživanje rezultata. Zaokruživanjem se smanjuje preciznost, prvo se dio nižih bitova odstrani, zatim se preostali bitovi modificiraju kako bi što točnije prikazali cijelokupnu vrijednost, tj. vrijednost prije zaokruživanja. Procesor nudi nekoliko načina zaokruživanja: pristrano, nepristrano i odsijecanje. Prva dva načina ovise o bitu RND_MOD registra ASTAT, tako da ako je RND_MOD jednak 1, provodi se pristrano zaokruživanje, a ukoliko je RND_MOD jednak 0, nepristrano. Kod nepristranog zaokruživanja, zaokružuje se na broj koji je najbliži originalu. U slučajevima kada je broj točno na polovici, ovom se metodom zaokružuje na najbliži paran broj (tj. onaj kod kojega je LSB = 0). Nepristrano metoda koristi mogućnost ALU jedinice da zaokruži 40-bitni rezultat na granici bitova 15 i 16, a može se izvesti kao dio tijela instrukcije. Pristranim zaokruživanjem se također zaokružuje na najbliži broj originalu, ali u slučaju kada je broj točno na polovici zaokružuje na veći od dva okolna broja (zaokruživanje na gore). Zbog toga što uvijek zaokružuje na veći broj, ova se metoda naziva pristranom. Još jedan način zaokruživanja odsijecanje - određeni broj najnižih bitova se maskira, odsiječe. Ovo je također vrlo pristrana metoda zaokruživanja, koja ne ovisi o RND_MOD bitu registra ASTAT.

Budući da su podržane cijelobrojna i frakcionalna aritmetika, slijedi nekoliko izvedbi algoritma za direktnu digitalnu sintezu, počevši sa decimalnom realizacijom, pa se kasnije prelazi na cijelobrojnu koja je puno bliža načinu rada procesora, te je time i efikasnija.

4. Realizacija algoritma na procesoru ADSP – BF537

Programsko riješenje za procesor započinjemo u programskom okruženju Visual Studio 2008, koji čemo koristiti za *floating point* realizaciju, te pripremu za cijelobrojnu realizaciju. Prava cijelobrojna realizacija (*fixed point*) se izvodi u progamskom paketu VisualDSP++ koje je namjenjeno isključivo procesorima tvrkte Analog Devices.

4.1. Floating point realizacija

Prvi korak u realizaciji algoritma direktne digitalne sinteze na procesoru za digitalnu obradu podataka je tzv. *floating point* realizacija. Ovdje se radi sa „pravim“ decimalnim brojevima, te decimalnim operacijama zbrajanja i množenja. Ovakav pristup ne pogoduje arhitekturi i načinu rada DSP procesora, pa zbog toga iako je efektivan, iznimno je neefikasan. Razlog tome je što procesor najbrže radi sa cijelobrojnim vrijednostima, dok za decimalne brojeve mora simulirati decimalne operacije, i time nepotrebno troši cikluse, i gubi vrijeme do konačnog rezultata.

4.1.1. Opis kôda i objašnjenje algoritma za floating point realizaciju

```
double g(unsigned long l)
{
  return l;
}
```

- na početku, prije tijela glavne funkcije pišemo funkciju koju koristimo za pretvaranje ulaznih koeficijenata iz unsigned long tipa podataka u double (decimalni zapis dvostruke preciznosti, 8 bajtova memorije)

-ovdje slijedi deklariracija konstantnih polja: `sign_coef_fix[]` (fiksni predznaci za svaki od 4 koeficijenta), `sr[]` (broj koraka za koje se svaki koeficijent mora pomaknuti udesno, radi poravnanja sa slijedećim koeficijentom), te

rez_coef[] (polje koje također sadrži broj koraka za pomak udesno, ali zbog kvantizacije pojedinih koeficijenata)

```
for (ja=0; ja<=3;ja++){
for (ia=0; ia<=255; ia++){
     ba=g(aa[ja][ia]);
     ca[ja][ia]=ba;
```

- pomoću dvije petlje prelazimo preko svakog elementa dvodimenzionalnog polja i prebacujemo koeficijente iz polja aa[][] preko funkcije g, koja vraća pročitani podatak u obliku double, u polje ca[][]

```
switch (ja)    {
     case 0:   ca[ja][ia]=ca[ja][ia]*pow(2, -rez_coef[ja]);
               break;
     case 1:   ca[ja][ia]=ca[ja][ia]*pow(2, -rez_coef[ja]);
               break;
     case 2:   ca[ja][ia]=ca[ja][ia]*pow(2, -rez_coef[ja]);
               break;
     case 3:   ca[ja][ia]=ca[ja][ia]*pow(2, -rez_coef[ja]);
               break;
                    }
```

- naredbom switch-case se kvantiziraju koeficijenti. Koeficijenti su tablicu poranjeni u svojim cijelobrojnim oblicima, te sa svojim stvarnim vrijednostima (koeficijenti su poravnati na desno, što znači da su najmanje značajne znamenke na istom mjestu). Budući da u prvoj realizaciji radimo sa brojevima u decimalnom zapisu, ovi se koeficijenti moraju dovesti u pogodan oblik. Ovo radimo dijeljenjem koeficijenata odgovarajućom potencijom broja 2, tj. posmakom za odgovarajući broj mjesta udesno. U prvom slučaju taj će pomak odgovarati broju bita svakog koeficijenta, tako da bi se dobio najveći mogući broj iz raspona od 0 do 1, koji će reprezentirati taj koeficijent. Ovi su podaci za svaki koeficijent pohranjeni u polju konstanti rez_coef[], i redom iznose, za koeficijent D - 4 bita, za C - 14 bita, za B - 23 bita i za A - 31 bit. No, ovakvim nejednolikim pomacima smo izgubili prvotno poravnanje na desnu stranu, te su koeficijenti nakon ovih operacija poravnati na lijevo.

Za koeficijent D (4 bita od 32-bitnog registra), posmak za 4 mjesta udesno:

Vidimo razliku poravnanja brojeva unutar registra u odnosu na decimalnu točku. Ista situacija je i s ostalim koeficijentima:

C (14 bita od 32-bitnog registra):

B (23 bita od 32-bitnog registra):

A (31 bit od 32-bitnog registra):

Ovaj gubitak poravnanja znači da ćemo pri množenju po Hornerovoj formuli u svakom koraku morati poravnavati najmanje značajne znamenke, posmakom udesno za određen broj mjesta (broj posmaka odgovara razlici prvotnih posmaka dvaju koeficijenata). Time smo na kraju dobili decimalni broj poravnat na desni rub registra, tj. posmicanjem smo došli do kraja registra. Ovi su posmaci pohranjeni u polju `sr[]`.

```
N=pow(2, b);
Nt=pow(2, 20);
step=1./Nt;
```

- pridruživanje vrijednosti varijablama `N` i `Nt` (broj segmenata za cijelu kružnicu i broj točaka), te varijabli `step` koja označava inkrement faze u jednom obilasku petlje

- ulazimo u glavnu petlju, u kojoj se za svaki od ukupno 1048576 koraka aproksimira vrijednost sinusne funkcije:

```
for (phf=0, j=0; phf<1; phf=phf+step, j++){

ph=floor(phf*pow(2, b_phac))*pow(2, -b_phac);
```

- kvantizacija `phf` na rezoluciju faznog akumulatora

```
in=floor(ph*N);
```

- prvi korak u određivanju indeksa tablice koeficijenata, pomak faze za 10 mjesta ulijevo (množenje s 2^{10})i uzimanje cijelog broja operacijom `floor`

```
inMSB=floor(in/(N/2));
```

- najviši bit indeksa (MSB – *most significant bit*), dobije se pomakom varijable `in` za 9 mjesta ulijevo, tj. dijeljenjem s 2^{b-1}, zapravo 2^9. Ovaj bit označava periodu sinusoide, vrijednost mu je 0 za prvu poluperiodu (takva će biti za prvih 524288 prolazaka kroz petlju), te 1 za drugu poluperiodu (drugih 524288 prolazaka)

```
Zdx=ph*N-in;
Zdx=2*Zdx-1;
```

- određivanje frakcionalnog dijela faze, ostatka koji smo operacijom `floor` odbacili da bismo dobili vrijednost `in`. Slijedi da je rezolucija `Zdx` zapravo `b_phac-b` (zbog množenja sa 2^b kod računanja in), dakle 18 bitova. Nadalje, ovaj se broj prebacuje iz raspona 0 do 1 u raspon -1 do 1, jednostavnom operacijom množenja i oduzimanja

```
in=(in-(int)(in/(N/2))*(N/2))+1;
```

- izbacujemo `inMSB` iz indeksa, jer će nam on služiti za određivanje poluperiode sinusoide. Ovime smo prepolovili moguće vrijednosti indeksa `in`

```
inNSB=floor((in-1)/(N/4));
```

- određujemo sljedeći najviši bit, `inNSB` (NSB – next significant bit) adrese. Za prvi i treći segment sinusoide, tj. prvu i treću četvrtinu (0 do π/2, te od π do 3π/2) inNSB poprima vrijednost 0; dok za drugi i četvrti segment poprima vrijednost 1. Ovaj nam bit također služi za iskorištavanje simetrija sinusoide, jer će se u ovisnosti o njemu mijenjati predznak pojedinih koeficijenata prije množenja po Hornerovoj formuli

```
ind=in;
if(inNSB){
          ind=((N/2-1)-(in-1))+1;
          }
```

-ukoliko je `inNSB = 1`, što bi značilo da se nalazimo u drugom ili četvrtom kvadrantu, komplementira indeks `ind`, i time se dobije adresa koeficijenata prvog kvadranta

```
ind=ind-1;
```

- ovdje se `ind` umanjuje za `1` jer, za razliku od Matlab-a gdje vrijednosti adresa polja počinju od `1` i idu do `max`, kod C-a su vrijednosti adresa u rasponu od `0` do `max-1`

```
sign_coef[0]=sign_coef_fix[0]*pow(-1, inNSB)*pow(-1, inMSB);
sign_coef[1]=sign_coef_fix[1]*pow(-1, inMSB);
sign_coef[2]=sign_coef_fix[2]*pow(-1, inNSB)*pow(-1, inMSB);
sign_coef[3]=sign_coef_fix[3]*pow(-1, inMSB);
```

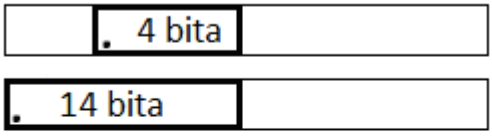
- određivanje predznaka pojedinih koeficijenata u ovisnosti o bitovima `inMSB` i `inNSB`, te o konstantnom polju `sign_coef_fix`, koje je predefinirano

- sada počinjemo množenje po Hornerovoj formuli kako bismo dobili polinom trećeg stupnja; prvo se koeficijent d množi s odgovarajućim predznakom, s obzirom na točku na sinusoidi gdje se nalazimo

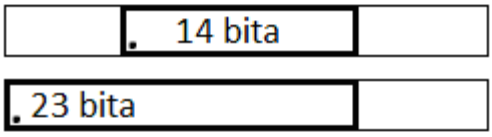
```
yiQc=sign_coef[0]*ca[0][ind];

for (i=1; i<=3; i++){
     yiQc=yiQc*Zdx*pow(2, -sr[i-1]);
     yiQc=yiQc+sign_coef[i]*ca[i][ind];
     }
```

-petlja od 3 koraka koja simulira množenje po Hornerovoj formuli: u prvom koraku je kao `yiQc` postavljen koeficijent `d`, koji se nakon toga množi sa

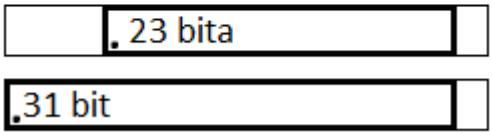
frakcijom faze, varijablom `Zdx`, pomiče udesno za broj koraka opisan poljem `sr[]`, kako bi se poravnao sa sljedećim koeficijentom, te se zbraja s njim.
Prvo se u izlaznu varijablu pohrani koeficijent D, i pomnoži se Zdx:
Poravnanje za zbrajanje sa C koeficijentom



Poravnanje za zbrajanje sa B koeficijentom



Poravnanje za zbrajanje sa A koeficijentom



Nakon što se algoritam izvrši, u istoj mapi dobijemo dvije datoteke, „yiQc.txt“ te „polje_koef_cubBqn.txt“. Prva sadrži vrijednosti izlazne varijable, u ovom slučaju „yiQc“ za svaki prolaz kroz „osnovnu“ petlju programa (dakle, sadrži 1048576 vrijednosti), dok druga sadrži sve koeficijente polinoma u decimalnom zapisu. Preostaje nam još provjera točnosti u odnosu na referentni kôd Matlab-a.

4.1.2. Provjera točnosti floating point realizacije

Točnost ovog algoritma se provjerava u nekoliko koraka, preciznije njih 9, zbog prevelikog broja izlaznih vrijednosti. Izlazno polje se podijelilo u 9 segmenata, njih 8 po 120000 elemenata i posljednji od 88576 elemenata. Zbog postavki *compilera*, nismo u mogućnosti odjednom obuhvatiti izlazne vrijednosti za sve točke (njih 1048576), već se program mora pokrenuti nekoliko puta, tako da se svaki puta pohrani novi dio izlazne sinusoide. Primjerice, u prvom „prolazu“ se pamti prvih 120000 elemenata, u drugom se pamte elementi od 120001 do 240000, i tako dalje. Na kraju, nakon 8 takvih prolaza, do kraja izlaznog polja

ostane 88576 elemenata. Varijabla x se koristi kako bi se jednostavnije formirale granice petlje pri svakom od navedenih 9 prolaza. Programsko ostvarenje ovog postupka se mora dodati na već postojeći kôd:

```
double izlaz[88576];
double x;
x=120000.*step;
```

- varijabla x služi za jednostavnije određivanje granica prilikom provjere točnosti. U prvom prolasku phf ide od 0 do x, u drugom od x do 2*x i tako redom, do devetog prolaska u kojem je phf u intervalu između 8*x i 1

- nakon što se naprave potrebne promjene u kôdu i unutar definicije petlje podesi „hod“ varijable *'phf'*, dobije se i treća datoteka, u kojoj se nalazi binarni zapis izlazne varijable *'yiQc'*. Isto se mora ponoviti nekoliko puta uz odgovarajuće izmjene:

- ime same datoteke se mora promijeniti, npr. za prvi segment je bilo *izlaz0-12.bin*, pa je za drugi *izlaz12-24-bin*, za treći *izlaz24-36.bin* i tako dalje
- hod petlje se mora prilagoditi segmentu koji želimo pohraniti, npr. za drugi segment *'phf'* ide od x do 2*x, za treći od 2*x do 3*x i tako dalje (x je varijabla u koju se pohranjuje vrijednost *'phf'* za 120000 koraka)

Nakon što smo dobili sve datoteke, sama provjera se radi unutar Matlab-a, gdje prije svega moramo pokrenuti odgovarajuću datoteku, kako bi se formirale sve potrebne varijable.

U Dodatku je ponuđen cijelokupni programski kôd, s već ugrađenim linijama za provjeru, kojima je dodan prefiks '//' kako se ne bi izvodile u prvom prolasku kroz program.

Slijedi Matlab kôd za provjeru točnosti:

```
y12=yiQc(1,1:120000);

fp=fopen('izlaz0-12.bin','rb');
[x12,count2]=fread(fp,inf,'double');
fclose(fp);
```

```
razlika1 = max(abs(y12-x12'))
```

Prikazani kôd vrijedi kod prvog segmenta izlazne varijable, dakle za prvih 120000 elemenata. U prvoj liniji u varijablu *y12* punimo prvih 120000 vrijednosti iz varijable *'yiQc'* u kojoj se nalaze vrijednosti dobivene Matlab-ovim kôdom. Zatim se iz datoteke *'izlaz0-12.bin'* upisuju vrijednosti u varijablu *x12*. Ovdje je potrebno naznačiti o kojem se tipu podataka radi, u našem slučaju od tipu *double*, kako bi Matlab znao koliko bitova treba preuzeti za pojedinu vrijednost. Sada nam preostaje oduzeti te dvije varijable. Podatke koje čita iz datoteke Matlab automatski stavlja u varijablu u obliku vektor-stupca. Zbog toga je pri oduzimanju varijabli potrebno transponirati varijablu *x12*. Razlika ovih dviju varijabli mora biti, i jest, nula, što znači da su obje varijante (Matlab kôd i C kôd) potpuno jednaki, tj, daju potpuno jednake rezultate.

4.2. Priprema za cijelobrojnu (fixed point) realizaciju

Ukoliko želimo napraviti cijelobrojnu realizaciju, potrebno je napraviti kvantizaciju frakcije faze (*'Zdx'*) i međuprodukata u Hornerovoj formuli. Bitno je naglasiti da ovo nije prava cijelobrojna realizacija, jer su varijable i dalje *double* oblika i sve operacije su također *double*, odnosno daju rezultate tipa *double*. Ovakva se „prividna“ cijelobrojna realizacija može lako dobiti uz nekoliko jednostavnih preinaka kôda za decimalnu (float) realizaciju.

- petlja u kojoj se simulira Hornerova formula se zamjenjuje se slijedećom petljom:

```
yishq=sign_coef[0]*ca[0][ind];

for (i=1; i<=3; i++){
     rez=rez_coef[i]-sr[i-1];
```

- potrebna rezolucija za kvantizaciju umnoška, ovisi o rezoluciji slijedećeg koeficijenta i potrebnom pomaku trenutnog koeficijenta (za `i=1` se radi o `rez_coef` za koeficijent `c`, te o desnom shift-u (`sr[]`) koeficijenta d)

```
b_dx=rez+b_dxq_g;
```

- na ovaj se rezultat dodaje određeni broj bitova, takozvani „guard-bitovi“, koji služe kako bi se smanjila pogreška pri kvantizaciji, u našem slučaju 2 bita

```
dxq=floor(Zdx*pow(2, b_dx))*pow(2,-b_dx);
```

- kvantiziramo `Zdx` na broj bitova određen gornjim relacijama

```
dxq=yishq*dxq;
```

- množimo izlaz prošlog prolaska petlje sa kvantiziranom fazom `dxq`

```
dxq=floor((dxq*pow(2, rez))+0.5)*pow(2,-rez);
dxq=dxq*pow(2,-sr[i-1]);
```

- kvantiziramo rezultat uz zaokruženje na bliži cijeli broj, te pomičemo udesno za određen broj koraka zbog poravnanja sa slijedećim koeficijentom (objašnjeno uz prvotnu realizaciju)

```
yishq=dxq+sign_coef[i]*ca[i][ind];
```

- zbrajamo poravnati `dxq` sa slijedećim koeficijentom

```
}
```

Ovime smo napravili podlogu za cijelobrojnu realizaciju u vidu kvantizacije međuprodukata i faze. Jednostavnom provjerom, sličnom onoj za decimalnu realizaciju se rezultat može verificirati. Oduzimanjem vrijednosti dobivenih Matlabom i vrijednosti iz binarne datoteke se dobije 0, dakle C kôd je i u ovom slučaju točan.

4.3. Realizacija s cijelobrojnim koeficijentima i decimalnom fazom

Koeficijenti za interpolaciju korištenjem kubičnih *spline-a* su pohranjeni u polju `aa[][]` u heksadekadskom zapisu, i poravnati su na desnu stranu registra, odnosno najniža znamenka svih koeficijenata, najniži bit, su na prvom mjestu u registru, na poziciji 0. Kod decimalne realizacije, prvi korak nam je bio te cijele

brojeve, koji su dani u stvarnom odnosu jednih prema drugima, prebaciti u raspon od 0 do 1, na način opisan pri izvedbi algoritma.

Budući da su ti koeficijenti u tablici već zapisani u odgovarajućem obliku i odgovarajućem poravnanju, nad njima nije potrebno raditi posmake ili dijeljenja. Jednostavno ih množimo s decimalnim brojem koji odgovara frakciji faze, iz raspona -1 do 1, i zbrajamo sa sljedećim, većim koeficijentom (sa kojim smo već poravnati). Budući da time dobijemo cijeli broj mnogo veći od 1, trebamo odgovarajućim operacijama „prevesti“ taj broj u raspon od -1 do 1.

Programski se izvodi slično kao prvotna izvedva, izvedba decimalnim brojevima. Potrebno je nekoliko modifikacija:

- više ne trebamo deklaraciju funkcije g, koja je prebacivala cijele brojeve u decimalne, te dalje radimo sa poljem `aa[][]`
- dio kôda u kojem se obavlja množenje po Hornerovoj formuli treba zamijeniti sa sljedećim kodom

```
yishq=sign_coef[0]*aa[0][ind];

for (i=1; i<=3; i++){
                dxq=Zdx*yishq;
     yishq=dxq+sign_coef[i]*aa[i][ind];
}

     y_double=yishq*pow(2, -31);
```

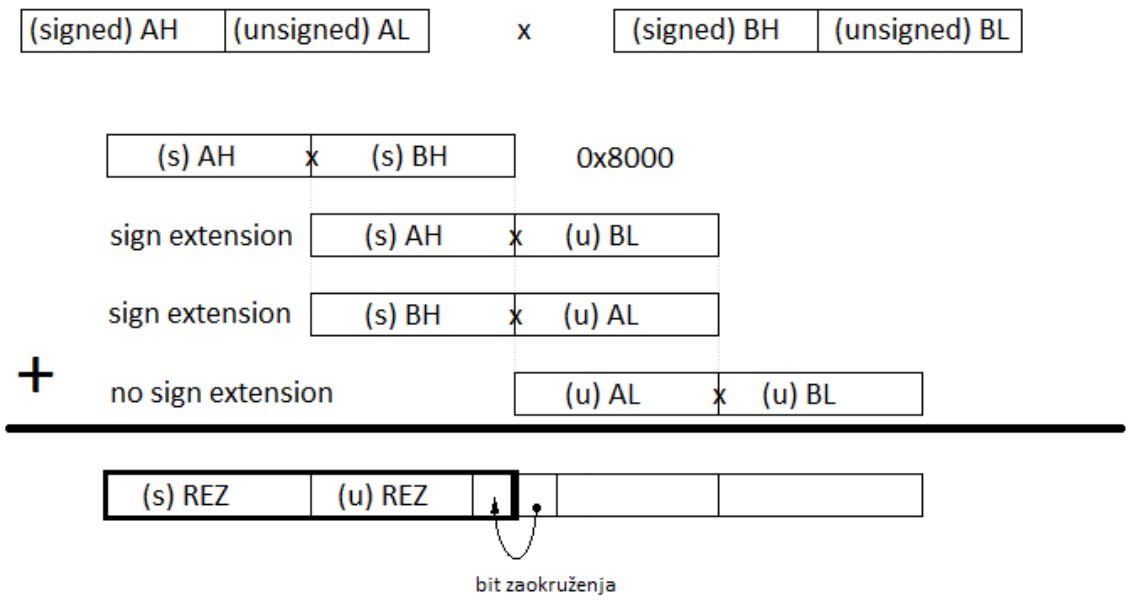
- na kraju moramo riješenje posmaknuti za 31 bit udesno kako bi bilo u rasponu od -1 do 1

U slijedećem koraku i fazu moramo prebaciti u cijeli broj, tako da bi imali cijelobrojno množenje i zbrajanje, koje je, kako smo naveli u opisu rada procesorskih jedinica, odgovarajuće cijelobrojnoj aritmetici koja je svojstvena korištenom Blackfin procesoru.

4.4. Prava cijelobrojna realizacija

Nakon što smo koristili prave cijelobrojne koeficijente, slijedeće što moramo napraviti je ostvariti potpunu cijelobrojnu realizaciju, time što ćemo i fazu `Zdx` prezentirati kao cijeli broj, čime možemo efikasnije iskoristiti ustroj i arhitekturu procesora.

Pošto je faza u rasponu -1 do 1, posmakom ulijevo za 31 bit, dobili bismo 32-bitni registar koji će sadržavati cijelobrojnu fazu sa predznakom. Ovo se jednostavno radi množenjem sa 2^{32}. No, ovdje će se javiti problemi, zbog toga što množimo dvije long varijable, tj. imamo umnožak dva 32-bitna broja, te nemamo dovoljno veliki tip podataka kuda bi pohranili rezultat. Zato moramo napraviti funkciju koja će pomnožiti takva dva broja i pohraniti samo gornja 4 bajta, tj. gornju 32-bitnu riječ. Kako bismo ovo mogli realizirati prvo moramo podijeliti obje varijable na gornju i donju poluriječ, i to imajući na umu da se gornje poluriječi moraju tretirati kao *signed*, a donje kao *unsigned*. Gornje poluriječi dobijemo ako cijelokupnu riječ od 32 bita posmaknemo udesno za 16 bitova, te rezultat spremimo u varijablu tipa `signed int`; dok donje poluriječi dobijemo maskiranjem gornjih 16 bitova, na način da napravimo bit-po-bit logičku operaciju `I` sa heksadekadskim podatkom `0x0000FFFF`, te pohranimo rezultat u varijabli tipa `unsigned int`.



Množenje se odvija u nekoliko razina unutar virtualnog 64-bitnog registra, a bitno je upamtiti da je svaki umnožak 32-bitan:

- lijevo poravnat umnožak gornjih riječi (`AH x BH`)
- središnje poravnata (16 bitova s lijeve i 16 s desne strane sredine registra) dva umnoška viših i nižih riječi (`AH x BL + BH x AL`)
- desno poravnat umnožak nižih riječi (`AL x BL`)

Prva se tri umnoška tretiraju kao `long` varijable s predznakom a preostali kao `long` bez predznaka. Slijedeće, moramo izračunati prijenos koje riječi s desne strane tog virtualnog 64-bitnog registra prenose u gornjih 32 bita, jer na kraju uzimamo samo taj dio kao rješenje. Kako bi izračunali prijenos, moramo izračunati zbroj pojedinih riječi s desne strane registra. Najniža riječ, niži dio umnoška `AL` i `BL` se zbraja s nulama, i tako ne može napraviti prijenos. Uzimamo gornju riječ posljednjeg umnoška, te donje riječi središnjih umnožaka koje spremamo u varijablu `long`, zbog desnog posmaka koji moramo raditi kako bi dobili prijenos (prijenos će biti dva bita, znači između 0 i 3) . Zbrajamo te tri long varijable i podatak `0x8000`, te sve zajedno posmičemo za 16 mjesta udesno i pohranjujemo. Sada zbrajamo preostale gornje riječi (*signed*), prijenos, i najtežu riječ, onu koju smo dobili množenjem `AH` i `BH`. Rezultat spremamo u varijablu tipa `signed long`, i radimo dalje zbrajanje po Hornerovoj formuli. U Dodatku je dan programski odsječak koji obavlja ovakvo množenje, funkcija nazvana `mult_32x32()`. Nakon što prođemo kroz 3 stupnja Hornerove formule, cijelokupni rezultat treba posmaknuti za 31 mjesto udesno, kako bi se prebacile vrijednosti u raspon između -1 i 1.

Programski se cijeli ovakav algoritam izvodi na način da se petlja za Hornerovu formulu iz prošlog primjera (4.3.) zamjeni slijedećom:

```
for (i=1; i<=3; i++){
     Zdxf=Zdx*pow(2, 31);
     dxq=floor(mult_32x32(yishq, Zdxf)*2+0.5);
     yishq=dxq+sign_coef[i]*aa[i][ind];
}
y_double=yishq*pow(2, -31);
```

Ovim postupkom dobijemo potpunu cijelobrojnu realizaciju korištenog algoritma za DDS. Ovakav će način rada biti mnogo puta efikasniji, brži od realizacije decimalnim brojevima jer procesor podržava cijelobrojnu aritmetiku, dok za decimalnu mora svaku računsku operaciju „simulirati“ kao decimalnu.

5. Zaključak

Navedeno je nekoliko pristupa realizaciji osnovnog algoritma direktne digitalne sinteze. Sve navedene izvedbe su točne, daju zadovoljavajuće rezultate, a pokazalo se kako, polazeći od prvotne decimalne izvedbe, i postepeno dodajući određena poboljšanja u kôdu, raste efikasnost kôda. Zbog značajki korištenog procesora, koji podržava cijelobrojnu aritmetiku, brže se izvršava program kod kojeg su računske operacije cijelobrojne, pa je tako posljednja navedena izvedba, izvedba kod koje su svi operandi cijeli brojevi, najefikasnija, iako jednako efektivna kao i prethodne.

6. Sažetak

Mnoge situacije i problemi s kojima se susrećemo u komunikacijama i industriji zahtjevaju stabilne i lako podesive izvore signala određenih frekvencija. Jedno od rješenja je direktna digitalna sinteza. Razvojem procesora za digitalnu obradu signala, ove metode su postale još raširenije zbog jednostavne mogućnosti programske izvedbe. Sustavi za DDS aproksimiraju potrebnu funkciju nekom od razvijenih metoda, ovisno o potrebama i svrsi samog signala. Ovdje je prikazan algoritam direktne digitalne sinteze sinusnog signala, koji se aproksimira kubičnim spline-ama, a zasniva se na faznom akumulatoru i ROM look-up tablici koeficijenata za aproksimaciju. Isti je algoritam izveden na Blackfinovom procesoru ADSP – BF537, koji podržava cijelobrojne operacije, pa je stoga algoritam optimiziran za takvu realizaciju.

Ključne riječi: direktna digitalna sinteza, fazni akumulator, kubične spline, kvantizacija, cijelobrojna aritmetika

6.1.Summary

Many situations and problems encountered in communication and industry demand stabil and easily adjustable signal sources of various frequencies. One of the solutions is a method of direct digital synthesis. With the development of digital signal processors, these methods are even more widespread, because of their simple programing implementation. DDS systems approximate the needed waveform with one of the developed methods, depending on the desired needs and application of the synthesised signal. The algorithm shown here uses cubic spline approximation, and is based on a phase accumulator and a pre-stored look-up table of approximation coefficients. The same algotihm is implemented on a Blackfin DSP, which supports integer atihmetics, therefore, the algorithm is optimised for integer realization.

Keywords: direct digital synthesis, phase accumulator, cubic spline, quantization, integer arihtmetics

7. Literatura

[1] Petrinović, Davor, Arhiva dokumenata za Blackfin porodicu, 22.04.2009. http://www.fer.hr/predmet/ppzdos/vjezbe_i_samostalni_rad

[2] Sekulić, D., Filipović, S., Praštalo, R., Osnove direktne digitalne sinteze www.etfbl.net/dokument.php/5482/1/Osnove_DDSa.doc

[3] Petrinović, Davor, Predavana iz Programske potpore za digitalnu obradu signala, 2010. http://www.fer.hr/predmet/ppzdos/predavanja

[4] Analog Devices, Blackfin processor architectural overview http://www.analog.com/en/embedded-processing-dsp/Blackfin/processors/Blackfin_architecture/fca.html

[5] Analog Devices, Blackfin processor programming reference, Revision 1.3 , September 2008

8. Dodatak

8.1. Dodatak A

U Dodatku A priložen je programski kôd za prvotnu izvedbu algoritma, decimalnu ili *floating-point* izvedbu, gdje su u kôd integrirane naredbe za provjeru, ali stavljene pod znak komentara('//'):

```
#include <stdio.h>
#include <math.h>

double g(unsigned long l)
{
  return l;
}

int main(){
      FILE *fp;
      FILE *pointer;
      //FILE *bin;
      int N;
      double inMSB, inNSB, Zdx, yiQc;
      int b=10;
      long Nt;
      double step;
      int b_phac=28;
      double phf, ph, in;
      int ind;
      int j, i;
      //double izlaz[88576];
      //double x;
      int sign_coef_fix[4]={-1,-1,1,1};
      int sign_coef[4];
      int sr[]= {10, 9, 8};
      int rez_coef[]={4, 14, 23, 31};
      int ia, ja;
      unsigned long aa[4][256]={{...},{...},{...},{...}};
      double ba;
      double ca[4][256];

      fp=fopen("polje_koef_cubBqn.txt", "w");
      for (ja=0; ja<=3;ja++){
      for (ia=0; ia<=255; ia++){
            ba=g(aa[ja][ia]);
            ca[ja][ia]=ba;

            switch (ja) {
                  case 0:     ca[ja][ia]=ca[ja][ia]*pow(2, -rez_coef[ja]);
                        break;
                  case 1:     ca[ja][ia]=ca[ja][ia]*pow(2, -rez_coef[ja]);
                        break;
                  case 2:     ca[ja][ia]=ca[ja][ia]*pow(2, -rez_coef[ja]);
                        break;
                  case 3:     ca[ja][ia]=ca[ja][ia]*pow(2, -rez_coef[ja]);
```

```
                        break;
                }
                fprintf(fp,"cubBqn [%d][%d] = %lf\n", ja, ia, ca[ja][ia]);
        }
        }

        fclose(fp);
        pointer=fopen("yiQc.txt", "w");
        fprintf(pointer,"%s","Izlazna vrijednost sinusa za floating point
realizaciju:\nVarijabla yiQc:\n");
        //bin=fopen("izlaz96-1048.bin","wb");
        N=pow(2, b);
        Nt=pow(2, 20);
        step=1./Nt;
        //x=120000.*step;

for (phf=0, j=0; phf<1; phf=phf+step, j++){
        ph=floor(phf*pow(2, b_phac))*pow(2, -b_phac);
        in=floor(ph*N);
        inMSB=floor(in/(N/2));
        Zdx=ph*N-in;
        Zdx=2*Zdx-1;
        in=(in-(int)(in/(N/2))*(N/2))+1;
        inNSB=floor((in-1)/(N/4));
        ind=in;
        if(inNSB){
                ind=((N/2-1)-(in-1))+1;
                 }
        ind=ind-1;
        sign_coef[0]=sign_coef_fix[0]*pow(-1, inNSB)*pow(-1, inMSB);
        sign_coef[1]=sign_coef_fix[1]*pow(-1, inMSB);
        sign_coef[2]=sign_coef_fix[2]*pow(-1, inNSB)*pow(-1, inMSB);
        sign_coef[3]=sign_coef_fix[3]*pow(-1, inMSB);

        yiQc=sign_coef[0]*ca[0][ind];
        for (i=1; i<=3; i++){
                yiQc=yiQc*Zdx*pow(2, -sr[i-1]);
                yiQc=yiQc+sign_coef[i]*ca[i][ind];
                }

        fprintf(pointer,"yiQc[%d]=%lf\n", j, yiQc);
        //izlaz[j]=yiQc;
        }
        //fwrite(izlaz, sizeof(double),88576,bin);
        //fclose(bin);
        fclose(pointer);
        }
```

8.2. Dodatak B

U Dodatku B donosimo petlju za izvedbu Hornerove formule uz kvantizaciju međurezultata:

```
yishq=sign_coef[0]*ca[0][ind];

      for (i=1; i<=3; i++){
            rez=rez_coef[i]-sr[i-1];
            b_dx=rez+b_dxq_g;
            dxq=floor(Zdx*pow(2, b_dx))*pow(2,-b_dx);
            dxq=yishq*dxq;
            dxq=floor((dxq*pow(2, rez))+0.5)*pow(2,-rez);
            dxq=dxq*pow(2,-sr[i-1]);
            yishq=dxq+sign_coef[i]*ca[i][ind];
            }
```

8.3. Dodatak C

U Dodatku C donosimo prvi dio koda koji upotrebljava dijelomičnu cijelobrojnu aritmetiku, na način da su koeficijenti za aproksimaciju signala cijeli brojevi, dok je frakcija faze decimalan broj:

```
yishq=sign_coef[0]*aa[0][ind];

for (i=1; i<=3; i++){
      dxq=Zdx*yishq;
      yishq=dxq+sign_coef[i]*aa[i][ind];
}

y_double=yishq*pow(2, -31);
```

8.4. Dodatak D

U Dodatku D donosimo funkcije koje obavljaju rastavljanje 32-bitnih riječi na dva dijela, te funkciju koja obavlja množenje dviju 32-bitnih riječi i pohranjuje rezultat u 32-bita. Također donosimo realizaciju Hornerove formule za cijelobrojnu izvedbu algoritma:

```
signed int h_int(signed long ulaz){
	signed int ah;
	ah=(signed int)(ulaz>>16);
	return ah;
}

unsigned int l_int (signed long ulaz){
	unsigned int al;
	al=(unsigned int)(ulaz&0x0000ffff);
	return al;
}

unsigned int h_int2(signed long ulaz){
	unsigned int ah;
	ah=(unsigned int)(ulaz>>16);
	return ah;
}
signed long mult_32x32(signed long prvi, signed long drugi){
		unsigned int prviL, drugiL, low1, low2, low3;
		unsigned long over;
		signed int high1, high2;
		signed long izlaz;
		signed int prviH, drugiH;
		signed long a1, b1, b2;
		unsigned long c1;

		prviH=h_int(prvi);
		prviL=l_int(prvi);
		drugiH=h_int(drugi);
		drugiL=l_int(drugi);

		a1=prviH*drugiH;
		b1=prviH*drugiL;
		b2=prviL*drugiH;
		c1=prviL*drugiL;

		low1=l_int(b1);
		low2=l_int(b2);
		low3=h_int2(c1);

		over=0x8000+low1+low2+low3;
		over=over>>16;

		high1=b1>>16;
		high2=b2>>16;

		izlaz=a1+high1+high2+over;

		return izlaz;
}
```

Cijelobrojna izvedba Hornerove formule:

```
yishq=sign_coef[0]*aa[0][ind];

for (i=1; i<=3; i++){
      Zdxf=Zdx*pow(2, 31);
      dxq=floor(mult_32x32(yishq, Zdxf)*2+0.5);
      yishq=dxq+sign_coef[i]*aa[i][ind];
}

y_double=yishq*pow(2, -31);
```