SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 297

Digitalni generator sinusnog signala s amplitudnom, faznom i frekvencijskom modulacijom izveden DSP procesorom

Tomislav Grbin

Zagreb, lipanj 2011.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

SADRŽAJ

POPIS SLIKA

POPIS TABLICA

1. Uvod

Tipičan sustav digitalne obrade signala sastoji se od pretvorbe ulaznog signala u digitalni signal, obrade takvog signala te povrata u analogni oblik koji se šalje na izlaz. Ovakav pristup možda se čini nepotrebno kompliciranim te je do prije nekoliko desetaka godina stvarno i bio. No, početak masovne proizvodnje integriranih krugova doveo je do vrlo niske cijene komponenata potrebnih za njihov razvoj. Danas se mogu naći jednostavni mikrokontroleri tek nešto skuplji od obične pasivne komponente kao što je kondenzator, što je prije bilo nezamislivo. Malo pomalo ovakvi sustavi postali su adekvatni za široku primjenu i donijeli mnogobrojne prednosti nad njihovim analognim ekvivalentima te ih danas nalazimo gotovo svugdje.

Digitalna obrada signala mlado je područje računarstva te predmet neprestanog istraživanja i novih otkrića, a razvoj DSP procesora ide ukorak s njenim razvojem. Specifična arhitektura ovih procesora usko je povezana s operacijama potrebnim u algoritmima digitalne obrade signala, što stvara poteškoće kod pisanja programa. Naime, instrukcije nisu kratke i jednostavne kao kod RISC[1] arhitektura, već više podsjećaju na VLIW[2] instrukcije. Prevoditelj zato vrlo teško generira strojni kod koji u zadovoljavajućoj mjeri iskorištava mogućnosti ovih procesora. Ovaj problem dovodi do stvaranja biblioteka koje su pisali i optimizirali ljudi direktno u strojnom kodu, a koje su usko namijenjene izvršavanju pojedinih algoritama.

Cilj ovog rada je napraviti jednu takvu biblioteku za generiranje signala raznih oblika, s naglaskom na visoku preciznost od 32 bita. U poglavljima koja slijede bit će opisan cjelokupni razvoj generatora signala, od matematičke pozadine aproksimacije amplitude sinusa i vrsta modulacije, preko specifičnosti razvojnog okruženja i sklopovlja na kojem je razvijen generator signala, pa sve do konačno postignutih rezultata.

[1] Reduced Instruction Set Computer

[2] Very Long Instruction Word

2. Izračun sinus funkcije

Najvažniji dio generatora signala precizna je aproksimacija sinusoide. Rješenje korišteno za ovaj dio uvelike određuje konačne performanse cjelokupnog sustava: brzinu izvođenja, maksimalnu pogrešku, iskorišteni memorijski prostor, pa čak i potrošnju energije.

U ovom poglavlju teoretski ćemo opisati metodu korištenu pri aproksimaciji amplitude sinusa te kako je ona primijenjena na stvarni sustav. Za dublju analizu metode pogledati [8] i [9].

2.1. Kubična B-spline interpolacija

Za zadanu fazu $p \in [0, 2\pi)$ potrebno je izračunati što precizniju aproksimaciju odgovarajuće amplitude sinusa, odnosno veličine $\sin(p)$.

Interval $[0, 2\pi)$ podijelit ćemo na proizvoljan broj segmenata koji označavamo s N, pri čemu su segmenti jednake širine, $\frac{2\pi}{N}$. Osnovna ideja ove metode je kroz svaki segment provući polinom trećeg stupnja koji najbolje odgovara sinusoidi na tom intervalu.

Koliko dobro polinom odgovara sinusoidi procijenjujemo koristeći maksimalnu pogrešku na cijeloj periodi koju nastojimo minimizirati. Također, prijelazi s jednog segmenta na drugi moraju biti maksimalno glatki.

Argument polinoma označavat ćemo s Δx. Lijevom rubu promatranog segmenta pridijelit ćemo vrijednost $\Delta x = -1$, a desnom $\Delta x = 1$.

Polinomi kojima aproksimiramo amplitudu sinusa tada su 2.1 i 2.2.

$$f(\Delta x) = a(\Delta x)^3 + b(\Delta x)^2 + c\Delta x + d \tag{2.1}$$

$$f(\Delta x) = ((a\Delta x + b) \cdot \Delta x + c) \cdot \Delta x + d \tag{2.2}$$

Oblik 2.2 pogodniji je za računanje jer omogućava implementaciju koristeći manje operacija nego 2.1. Primijetimo da se točka koja odgovara vrijednosti $\Delta x = 0$ nalazi

na sredini promatranog segmenta te da amplituda sinusa u toj točki odgovara upravo vrijednosti koeficijenta d.

2.2. Korištenje MATLAB skripte za izračun koeficijenata

Za računanje koeficijenata koristit ćemo MATLAB skriptu napisanu u sklopu [8]. U tablicama 2.1 i 2.2 može se naći kratak pregled ulaznih te izlaznih veličina skripte.

Tablica 2.1: Ulazne veličine MATLAB skripte

Parametar	Opis
b	broj segmenata, $N = 2^b$
poly_type	tip polinomijalne funkcije; moguće vrijednosti: **1:** *LMS*[1] metoda **2:** *MAE*[2] metoda **3:** optimalno *SFDR*[3] rješenje **4:** kubična B-spline interpolacija
b_out	željena izlazna rezolucija
coef_guard	broj guard bitova za svaki od četiri koeficijenta
quan_one	odabir saturacije
disp_coef	ispis svih koeficijenata

Koeficijenti korišteni u ovom radu dobiveni su pozivom

$$gen\_cub(10, 4, 29, [1, 2, 2, 2], 1, 0).$$

Drugim riječima, broj segmenata je $N = 1024$, korištena je B-spline interpolacija, a željena izlazna rezolucija je 1.31 (29 i još 2 guard bita) u frakcionalnoj notaciji (vidi [4] i [6]). Ostali parametri uzeti su po uzoru na [9].

Za razumijevanje izlaznih parametara potrebno je naglasiti nekoliko stvari. Koeficijenti nisu prikazani u jednakim rezolucijama. Za prikaz koeficijenta uz Δx^3 dovoljno je nekoliko bitova, dok nam za slobodni koeficijent treba puna izlazna rezolucija. Ovo je intuitivno jasno, jer bitovi jako male težine u koeficijentu a neće uopće utjecati na rezultat. Detaljnije objašnjenje može se naći u [9]. Kvantizacija koeficijenata provodi se zaokruživanjem na predviđeni broj bitova za taj koeficijent.

Tablica 2.2: Izlazne veličine MATLAB skripte

Parametar	Opis
cubB	koeficijenti prikazani s pomičnim zarezom
po	odnos stvarne vrijednosti koeficijenata i vrijednosti prikazane predviđenim brojem bitova
cubBn	normalizirani koeficijenti, $cubB$ podjeljen sa 2^{po}
MAE	maksimalna apsolutna pogreška
MSE	srednja kvadratna pogreška
MSEk	srednja kvadratna pogreška za svaki segment pojedinačno
rez_coef	potrebne rezolucije koeficijenata
cubBqn	koeficijenti normalizirani i kvantizirani na potreban broj bitova
max_coef_error	maksimalna pogreška nastala zbog kvantizacije; pogreška je izražena u LSB[4]-u izlazne amplitude
std_coef_error	standardna pogreška nastala zbog kvantizacije, izražena na isti način kao i prije
sign_coef_fix	fiksni predznak koji treba dodijeliti koeficijentima da bi u prvom kvadrantu svi bili pozitivni
QcubB, QcubBn, QcubBqn	iste vrijednosti kao i odgovarajući izlazi, ali svedeni na prvi kvadrant u skladu sa $sign\_coef\_fix$

Vrijednost sinusa preko cijele periode moguće je izračunati koristeći samo koeficijente iz prvog kvadranta. Ovo slijedi iz svojstava simetrije sinusoide i lako se izvodi relacija.

Nama su za efikasnu implementaciju potrebni koeficijenti u cjelobrojnom 32-bitnom obliku koji će onda biti promatrani kao frakcionalni 1.31 brojevi. Koeficijente u željenom obliku dobivamo množenjem $cubBqn$ sa 2^{rez_coef}.

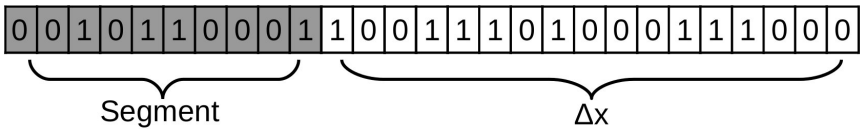
Potrebne rezolucije za pohranu koeficijenata (rez_coef) za korišteni poziv skripte iznose redom 4, 15, 24 te 32.

2.3. Prikaz u računalu

Pogledajmo kakav oblik imaju faze koje koristimo u računanju. U [8] je pokazano da za 32-bitni izlaz nema potrebe imati rezoluciju faze veću od 28, jer time ne dobivamo

na preciznosti. U ovom radu, bit će napravljeni generatori koji koriste faze od 26 i 27 bitova, zbog arhitekture ciljnog sustava.

Neka je faza 27-bitni cijeli broj bez predznaka, $p \in [0, 2^{27} - 1]$. Ova faza odgovara realnom broju $p \cdot \frac{2\pi}{2^{27}} \in [0, 2\pi\rangle$.



Slika 2.1: 27-bitni prikaz faze

Slika 2.1 prikazuje strukturu faze. Prvih 10 bitova označava segment u kojem se nalazi tražena faza i direktno nam govori koji od izračunatih koeficijenata nam trebaju.

Iz ostalih bitova nalazimo koji argument uvrštavamo u odgovarajući polinom. Želimo da se niz nula preslika u vrijednost -1, odnosno lijevi rub segmenta, a niz jedinica u 1, odnosno desni rub segmenta. Ovo lako postižemo komplementiranjem prvog bita te promatranjem takvog broja u 1.16 frakcionalnom formatu. Ovaj broj sad uvrštavamo u 2.2 te konačno dobivamo aproksimaciju amplitude za promatranu fazu.

Pogledajmo za početak jednostavan C-kod koji, koristeći izračunate koeficijente u pomičnom zarezu, dobiva željenu aproksimaciju amplitude sinusa. Neka su koeficijenti pohranjeni u formatu pomičnog zareza u nizu $coef[N][4]$. Potrebna nam je i funkcija koja iz faze računa stvarnu vrijednost argumenta polinoma. Program pretpostavlja 27-bitnu fazu.

Računanje sinusa u formatu pomičnog zareza

```
// racuna argument polinoma
double arg( int x ) {
  double w = 0.5;
  double ret = (x&(1<<16))? 0: -1;
  for( int i = 15; i >= 0; --i ) {
    if( x&(1<<i) ) ret += w;
    w /= 2.;
  }
  return ret;
}

double f( int x ) {      // aproksimacija sinusa
  int seg = x>>17;       // prvih 10 bitova daje nam segment
  double dx = arg( x ); // argument polinoma

  return  ((coef[seg][0]*dx
          + coef[seg][1])*dx
          + coef[seg][2])*dx
          + coef[seg][3];
}
```

Ova funkcija korištena je samo za inicijalnu provjeru izračunatih koeficijenata, dok je za stvarnu primjenu neprikladna zbog rada s brojevima u pomičnom zarezu. Naime, svi brojevi s kojima radimo bit će iz intervala $[-1, 1]$, pa nam za zbrajanje i množenje ne treba prikaz u pomičnom zarezu. Koristeći frakcionalni prikaz potrebne operacije se svode na rad s cijelim brojevima. Da bi dobili točan rezultat, stalno treba imati na umu koliku težinu imaju pojedini bitovi te koje su stvarne vrijednosti brojeva s kojima radimo.

U nastavku je kod koji koristi koeficijente pohranjene u 32-bitnom cjelobrojnom obliku i frakcionalnu aritmetiku za izračun. Koeficijenti su ponovno u nizu *coef*.

Računanje sinusa u frakcionalnoj aritmetici

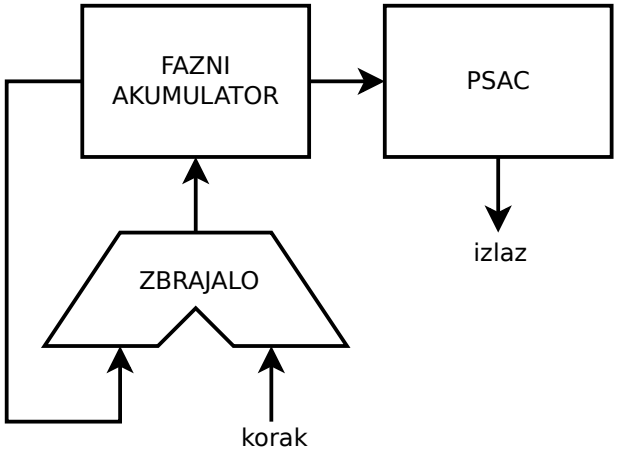
```
int f( int x ) {
  int seg = x>>17;           // prvih 10 bitova daje nam segment
  int dx = (x&((1<<17)-1)); // zadnjih 17 bitova
  dx -= (1<<16);             // komplementiranje prvog bita,
                             // te prosirenje na 32 bita

  long long sol = (coef[seg][0]*dx>>16) + coef[seg][1];
  sol = (sol*dx>>16) + coef[seg][2];
  sol = (sol*dx>>16);

  int isol = sol;

  if( isol+coef[adr][3]<0 && coef[adr][3]>0 && isol>0 ) isol = 0x7fffffff;
  else if( isol+coef[adr][3]>0 && coef[adr][3]<0 && isol<0 ) isol = 0x80000000;
  else isol += coef[adr][3];

  return isol;
}
```

Nakon svakog množenja rezultat je potrebno logičkim pomakom poravnati na frakcionalni format 1.31 u kojem su pohranjeni koeficijenti. Zadnjih nekoliko linija koda vrši saturaciju rezultata kod zadnjeg zbrajanja što će u stvarnoj izvedbi biti rješeno sklopovski. Kod je testiran uspoređivanjem sa standardnom sinus funkcijom u jeziku C te s funkcijom koja radi s brojevima u formatu pomičnog zareza. Dobivene pogreške približno su jednake onima u [9].

U stvarnoj implementaciji prva dva koeficijenta zauzimat će po 16 bitova, a preostala dva po 32 bita, što daje ukupno 12 byte-ova po segmentu. Ukupno zauzeće memorije iznosi 12 Kb.

2.4. Model generatora signala

Na slici 2.2 vidi se shematski prikaz generatora signala. Model generatora sastoji se od dva dijela. Glavni dio, u kojem se odvija samo računanje amplitude, zove se PSAC (engl. *Phase to Sine Amplitude Conversion*). PSAC na ulazu dobiva fazu, a na izlazu daje odgovarajuću amplitudu sinusa. Osim PSAC-a, generator signala ima i fazni akumulator, koji čuva trenutnu vrijednost faze. U našem slučaju fazni akumulator je samo varijabla koju mijenjamo ovisno o željenom obliku signala.



Slika 2.2: Model generatora signala

Za dobivanje sinusoide fazi u jednakim vremenskim razmacima dodajemo konstantni *korak*. Frekvencija dobivenog signala ovisi o koraku koji dodajemo te o brzini kojom smo u mogućnosti računati pojedine uzorke. Ako označimo korak sa k, a broj uzoraka koji možemo izračunati u jednoj sekundi sa s, tada je dobivena frekvencija dana izrazom 2.3.

$$f = \frac{s \cdot k}{2^b} \tag{2.3}$$

S b smo označili broj bitova faze, koji je u našem slučaju 26 ili 27. Da bismo dobili bar nekakvu sinusoidu, broj generiranih uzoraka po periodu trebao bi biti barem četiri, odnosno trebalo bi vrijediti $k \geq \frac{2^b}{4}$.

3. Modulacija

Prema [7], modulacija je proces mijenjanja jednog ili više svojstava signala s obzirom na neki drugi signal. Obično je prvi signal znatno više frekvencije od drugog. Svojstva koja nas najčešće zanimaju su amplituda, faza te frekvencija signala. Signal u koji moduliramo drugi signal zovemo i nosač (engl. *carrier*), dok signal koji moduliramo zovemo modulirajući. Modulacijom je moguć prijenos podataka sadržanih u modulirajućem signalu koristeći val nosač visoke frekvencije koji je pogodan za fizički prijenos u mediju, a koristi se i za multipleksiranje više različitih signala u jedan. Svi telekomunikacijski sustavi moraju koristiti neku vrstu modulacije da bi prenijeli željenu informaciju. Na primjer, kada razgovaramo mobilnim telefonom analogni signal iz mikrofona čija je frekvencija reda veličine par kiloherca modulira se u signal nosač od par gigaherca zato da bi bio moguć prijenos elektromagnetskim valovima na velike udaljenosti.

Razlikujemo digitalnu i analognu modulaciju.

Digitalna modulacija koristi se za prijenos nizova bitova preko vala nosača, a najraširenije vrste su FSK[1], ASK[2] te PSK[3]. Osnovni princip rada je preslikavanje niza od N bitova u diskretan skup od 2^N vrijednosti svojstva korištenog za modulaciju. Drugim riječima, svaki niz od N bitova kodira se s jednom od 2^N vrijednosti moduliranog svojstva signala. Može se reći i da je signal nosač kvantiziran po promatranom svojstvu. Primjerice, za FSK to svojstvo je frekvencija te val nosač ovisno o modulirajućem signalu poprima jednu od konačno mnogo unaprijed određenih frekvencija.

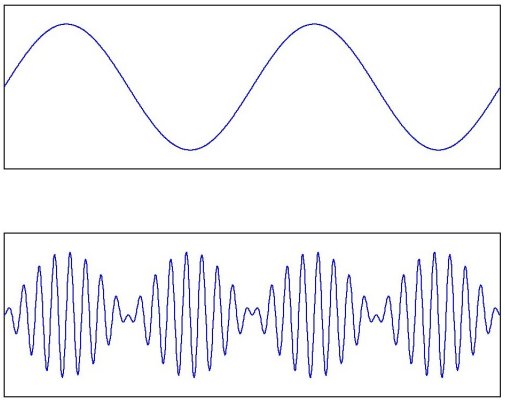
Kod analogne su modulacije signali s kojima radimo kontinuirani, odnosno nema diskretiziranog skupa iz kojeg oni poprimaju vrijednosti. Naravno, implementacijom na digitalnom računalu ovo je nemoguće u potpunosti postići, ali u analizi svejedno promatramo signale kao kontinuirane.

U nastavku ćemo pobliže opisati tri najraširenije vrste analogne modulacije: ampli-

[1] Frequency-Shift Keying
[2] Amplitude-Shift Keying
[3] Phase-Shift Keying

tudnu, frekvencijsku te faznu modulaciju.

3.1. Amplitudna modulacija



Slika 3.1: Amplitudna modulacija

Amplitudna modulacija najjednostavniji je oblik modulacije. Prvi radio prijenosi s početka prošlog stoljeća bili su modulirani upravo na ovaj način, a amplitudna modulacija ostaje dominatna u radio prijenosima sve do osamdesetih godina.

Kod ove vrste modulacije, amplituda vala nosača mijenja se ovisno o amplitudi modulirajućeg signala. Neka je val nosač sinusoida stalne frekvencije $f_c = \omega/2\pi$, a $m(t) \in [-1, 1]$ bilo kakav signal koji želimo modulirati. Tada amplitudno modulirani signal $y(t)$ dobivamo koristeći 3.2.

$$c(t) = \sin(\omega t), \tag{3.1}$$

$$y(t) = [A + m(t)] \cdot c(t), \tag{3.2}$$

Veličina A označava amplitudu vala nosača koju prenosimo u izlazni signal te će nam kasnije poslužiti za određivanje modulacijskog indeksa.

Spektar izlaznog signala sastoji se od tri dijela. Naime, prisutna je osnovna frekvencija vala nosača te još dvije komponente simetrične oko ove frekvencije koje nastaju zbog modulirajućeg signala. Ovakva modulacija nije energetski učinkovita jer je dosta energije sadržano u osnovnoj frekvenciji vala nosača koja ne donosi nikakvu informaciju. Takva modulacija zove se *Double-Sideband Full-Carrier*.

Uz odabir $A = 0$ dobivamo *Double-Sideband Supressed Carrier* modulaciju, kod koje u izlazu nije prisutan spektar vala nosača. Ovaj oblik koristit ćemo za generator

signala zbog toga jer nam jamči da se izlazni signal kreće unutar intervala $[-1, 1]$, što je bitno za implementaciju.

U diskretnom slučaju, amplitude signala se računaju i šalju na izlaz u pravilnim vremenskim razmacima.

Amplitudna modulacija

```
for(;;) {
  faza_c = faza_c + korak_c;
  faza_m = faza_m + korak_m;
  izlaz = PSAC( faza_c ) * PSAC( faza_m );
}
```

Možemo vidjeti da su prisutna dva fazna akumulatora, $faza_c$ i $faza_m$, te njihovi koraci. Konačni rezultat dobivamo kao umnožak amplituda za ove dvije faze. Frekvenciju oba signala možemo dobiti koristeći formulu 2.3.

Kod je moguće prilagoditi za generiranje i drugačijih oblika amplitudne modulacije uvođenjem konstanti koje se dodaju ili množe s amplitudama signala, ali je pri tome potrebno voditi računa o frakcionalnoj aritmetici u kojoj se izračun odvija.

3.1.1. Modulacijski indeks

Kod svih vrsta modulacije, modulacijski indeks važan je parametar modulacije. On nam govori u kolikoj će mjeri val nosač mijenjati svoja svojstva, odnosno koliko se daleko od osnovne vrijednosti smije pomaknuti određeno svojstvo. Ova veličina zove se još i dubina modulacije te se najčešće izražava postotkom. U našem slučaju, kada je modulirajući signal ograničen na interval $[-1, 1]$, modulacijski indeks dan je s 3.3.

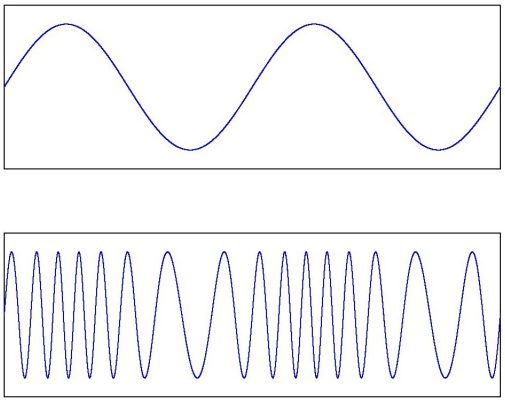
$$h = \frac{1}{A} \tag{3.3}$$

Modulacijski indeks od 50% značio bi da se amplituda vala nosača kreće u intervalu $\left[-\frac{3A}{2}, \frac{3A}{2}\right]$.

U našem slučaju je $A = 0$, pa nema govora o modulacijskom indeksu u ovom obliku.

3.2. Frekvencijska modulacija

Frekvencijska modulacija znatno je otpornija na šum i interferenciju od amplitudne te je koristi većina današnjih radio-stanica. FM prijenos patentiran je 1933. godine, a prva radio stanica osnovana je 1939. Dvadesetak godina kasnije u potpunosti se prelazi sa amplitudne na ovu vrstu modulacije.



Slika 3.2: Frekvencijska modulacija

Kod frekvencijske modulacije amplituda je stalna, a frekvencija vala nosača mijenja se ovisno o amplitudi modulirajućeg signala.

$$c(t) = \sin\left[2\pi \int_0^t (f_c + f_\Delta m(t))\right] \tag{3.4}$$

Veličina f_Δ se zove još i *devijacija*. Ona predstavlja maksimalno moguće odstupanje od osnovne frekvencije nosača i u direktnoj je vezi s modulacijskim indeksom frekvencijske modulacije.

Ako s f_m označimo maksimalnu frekvencijsku komponentu prisutnu u $m(t)$, tada je modulacijski indeks dan formulom 3.5.

$$h = \frac{f_\Delta}{f_m} \tag{3.5}$$

Ako je $h \ll 1$, tada govorimo o uskopojasnoj frekvencijskoj modulaciji za koju širina pojasa iznosi otprilike $2f_m$. Ako je $h \gg 1$, govorimo o širokopojasnoj modulaciji koja zauzima znatno više, ali je mnogo otpornija na šum.

U diskretnom slučaju, integral u 3.4 svodi se na sumiranje u pravilnim vremenskim razmacima.

Frekvencijska modulacija

```
for(;;) {
  faza_m = faza_m + korak_m;
  faza_c = faza_c + korak_c + d*PSAC( faza_m );
  izlaz = PSAC( faza_c );
}
```

Veličina d direktno utječe na devijaciju modulacije te na modulacijski indeks. Ovu vezu možemo izračunati tako da nađemo maksimalnu frekvenciju koju možemo dobiti i od toga oduzmemo osnovnu frekvenciju vala nosača. Frekvencije dobijemo iz koraka

koristeći 2.3.

$$\begin{aligned} f_\Delta &= \frac{s \cdot (k+d)}{2^b} - \frac{s \cdot k}{2^b} \\ &= \frac{s \cdot d}{2^b} \end{aligned} \tag{3.6}$$

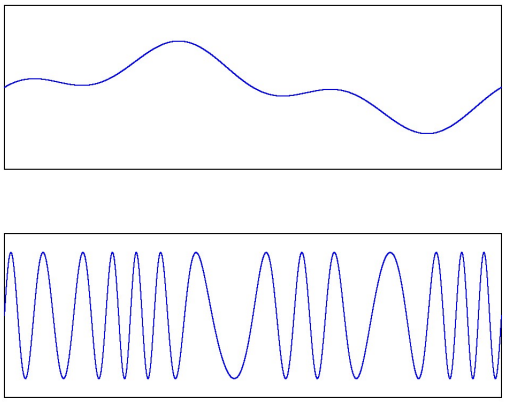
Obrnutu vezu daje nam 3.7.

$$d = \frac{f_\Delta \cdot 2^b}{s} \tag{3.7}$$

Ovdje opet koristimo veličine s i b koje nam kažu broj uzoraka u sekundi te rezoluciju faze u bitovima.

3.3. Fazna modulacija

Kod fazne modulacije korisna informacija modulira se mijenjajući fazu vala nosača. Rijetko se koristi u radio komunikaciji zbog složenijeg sklopovlja potrebnog za modulaciju i demodulaciju, dok primjenu nalazi u digitalnim glazbenim sintesajzerima gdje se uspostavilo da je njena implementacija jednostavnija od implementacije frekvencijske modulacije.



Slika 3.3: Fazna modulacija

Izlazni signal dan je formulom 3.8.

$$y(t) = \sin(\omega t + m(t)) \tag{3.8}$$

Faznu modulaciju, kao poseban slučaj frekvencijske, dobijemo uvrštavanjem derivacije modulirajućeg signala u formulu 3.4.

Diskretni slučaj također je sličan onome kod frekvencijske modulacije. Razlika je što ovdje modulirajući signal ne dodajemo na fazu nosača u svakom koraku, već kod samog računanja.

Fazna modulacija

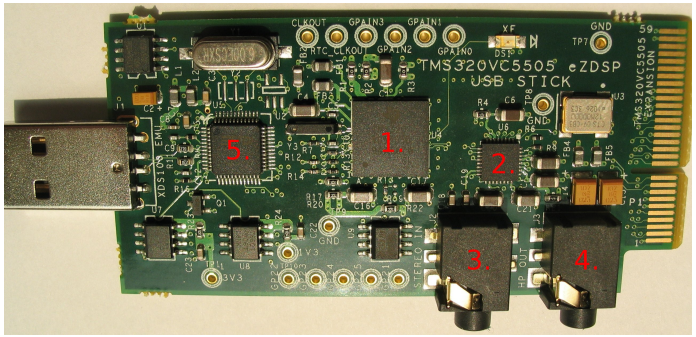
```
for(;;) {
  faza_m = faza_m + korak_m;
  faza_c = faza_c + korak_c;
  izlaz = PSAC( faza_c + d*PSAC( faza_m ) );
}
```

Modulacijski indeks i devijacija, slično kao u prethodnim primjerima, opisuju koliki pomak u fazi može uzrokovati modulirajući signal, međutim nemaju praktičnu manifestaciju kao kod ostalih spomenutih modulacija.

4. Razvojna pločica

U ovom poglavlju osvrnut ćemo se na okolinu u kojoj je razvijen generator signala. Korišten je Texas Instruments razvojni sustav oznake *TMS320VC5505 eZDSP*.

Riječ je o pločici koja sadrži DSP procesor te sve potrebno za prototipni razvoj programske podrške, a moguće je i spajanje dodatnog sklopovlja. Pločica sadrži sklopovlje potrebno za komunikaciju s računalom preko USB-a te JTAG sučelje za učinkovito pronalaženje grešaka u programu.



Slika 4.1: Razvojna pločica

Pločica sadrži i sklop za kodiranje i dekodiranje zvuka te standardne ulazne i izlazne stereo priključke. Ovo omogućava jednostavnu međusobnu konverziju između analognih i digitalnih signala, ali i jednostavno spajanje te će biti od važnosti kod testiranja generatora signala.

Na slici 4.1 vidljiv je izgled pločice te raspored spomenutih komponenti:

1. DSP procesor
2. audio kodek
3. stereo ulaz
4. stereo izlaz
5. USB sučelje

Programska podrška sastoji se od alata *Code Composer Studio* s kojim stižu i potrebni driver-i. Jednostavno i intuitivno sučelje omogućava brz razvoj programske

podrške te pokretanje i kontrolu rada na samom DSP-u. Detalji korištenja alata mogu se naći u [1].

4.1. DSP procesor

DSP procesor koji se nalazi na pločici je *TMS320VC5505*. To je cjelobrojni procesor (engl. *Fixed-Point processor*) konstruiran posebno za sustave u kojima je bitna mala potrošnja. Dolje su navedene njegove najbitnije karakteristike.

- takt do 100Mhz
- 320Kb memorije
- dvije 17x17 *pomnoži-i-zbroji* (engl. *Multiply-Accumulate*) jedinice
- 40-bitna ALU jedinica
- dodatni 16-bitni ALU
- nezavisni 40-bitni Barrel Shifter
- standardne periferije: I^2C, UART, AD konverter, ...
- velike mogućnosti za paralelizaciju instrukcija
- 12 nezavisnih sabirnica koje omogućavaju istovremeni rad sa više memorijskih lokacija
- sklopovski upravljane petlje

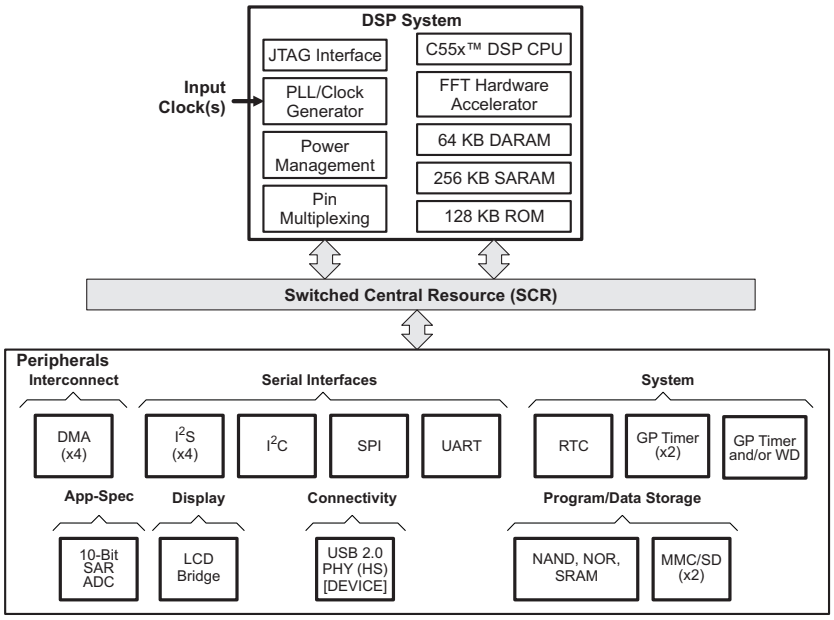
Tipične primjene ovog procesora uključuju bežične komunikacijske sustave, prijenosne media-playere, osobne medicinske uređaje, digitalne kamere te razne druge sustave koji zahtijevaju učinkovitu obradu signala s obzirom na potrošnju i brzinu.

Na slici 4.2 može se vidjeti osnovna struktura procesora sa svim periferijama koje su nam na raspolaganju. Periferije su približno iste kao i kod drugih procesora, no zanimljivo je primijetiti sklopovlje za izračun brze Fourierove transformacije (engl. *FFT*).

Mi ćemo se fokusirati na jezgru procesora (CPU) u kojoj se izvršavaju instrukcije i koja upravlja radom ostalih podsustava. Slika 4.3 prikazuje njenu osnovnu strukturu. Vidljive su četiri glavne jedinice koje će u nastavku biti ukratko opisane. Detaljniji uvid u arhitekturu jezgre može se pronaći u [5].

4.1.1. Jedinica za dohvat instrukcija (I jedinica)

Ova jedinica (engl. *Instruction Buffer Unit*) brine se za dohvat instrukcija iz memorije. Instrukcije su varijabilne duljine, od jednog pa do maksimalno šest byte-ova. U sva-



Slika 4.2: Arhitektura DSP-a

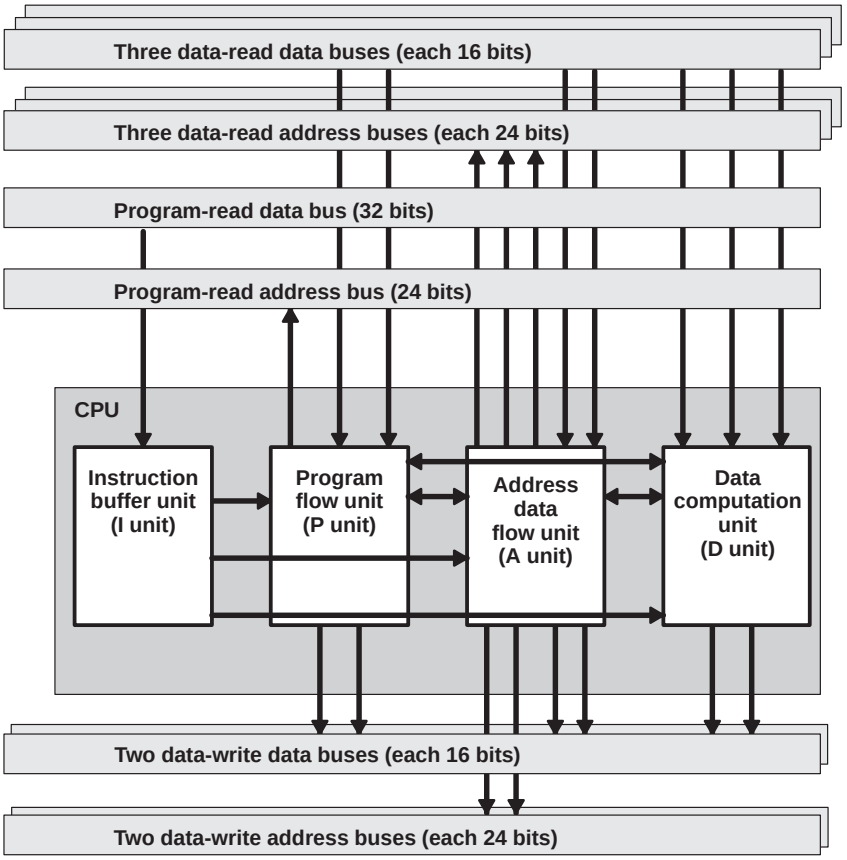
kom ciklusu jedinica može učitati četiri byte-a iz programske memorije i spremiti ih u međuspremnik (engl. *buffer*) kapaciteta 64 byte-a. Podaci iz međuspremnika se dekodiraju te se šalju upravljački signali ostalim jedinicama. Ovakav pristup omogućava neprekidan protok instrukcija prema samoj jezgri.

4.1.2. Jedinica za kontrolu programskog toka (P jedinica)

Jedinica za kontrolu programskog toka (engl. *Program Flow Unit*) generira adrese za dohvat instrukcija, kontrolira grananja, sklopovski izvedene programske petlje, pozive funkcija, povrate iz funkcija te posluživanje prekida. Njena izvedba zaslužna je za dobru popunjenost cjevovoda.

4.1.3. Jedinica za kontrolu toka podataka (A jedinica)

Ova jedinica (engl. *Address Data Flow Unit*) generira adrese potrebnih podataka u memoriji. Sadrži zasebni 16-bitni ALU koji omogućava izvođenje jednostavnijih operacija paralelno sa glavnim ALU-om. Ovaj ALU koristi se, na primjer, za generiranje adrese dodavanjem konstante na početnu adresu. Kada ALU nije upogonjen generiranjem adrese, instrukcija ga može iskoristiti za zbrajanje, oduzimanje, ili bilo kakvu logičku operaciju nad 16-bitnim operandima. Osim ovog ALU-a, jedinica sadrži i tri dodatna adresna generatora koji provode jednostavnije operacije pomicanja pokazi-



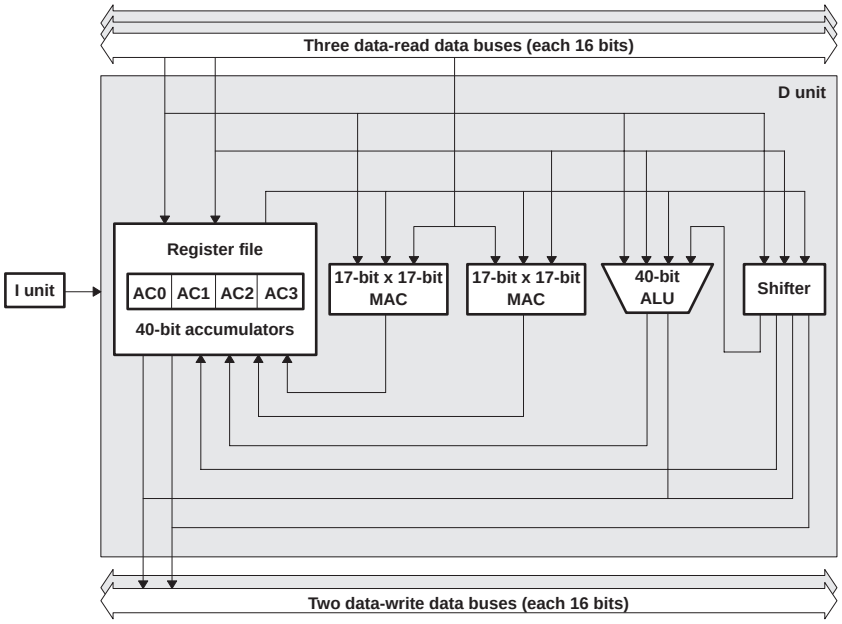
Slika 4.3: Arhitektura jezgre DSP-a

vača za jedno mjesto u memoriji. Dodatni adresni generatori omogućavaju generiranje adrese za tri 16-bitna podatka u memoriji, čije se dohvaćanje onda odvija preko nezavisnih sabirnica. Ovo će biti detaljnije objašnjeno kad bude govora o načinima adresiranja, u poglavlju 5.

4.1.4. Računska jedinica (D jedinica)

U računskoj jedinici (engl. *Data Computation Unit*) obavljaju se sve složene operacije množenja, zbrajanja i logičkog pomaka. Jedinica sadrži dva Multiply-Accumulate (MAC) sklopa, ALU te barrel shifter (slika 4.4). Ova jedinica, zajedno sa sustavom sabirnica, zaslužna je za velike mogućnosti paralelizacije koje podržava ovaj procesor.

Kao primjer, moguće je u jednom ciklusu dohvatiti iz memorije tri 16-bitna broja, dodati umnožak prva dva na vrijednost akumulatora, dodati umnožak druga dva na vrijednost nekog drugog akumulatora te logički pomaknuti vrijednost trećeg akumulatora za zadanu vrijednost. Više o iskorištavanju paralelizma bit će rečeno u poglavlju 5.



Slika 4.4: Arhitektura D jedinice

4.2. Audio kodek

Sklop koji kodira i dekodira zvučne signale sa stereo priključaka na pločici ima oznaku *AIC3204*. Konfigurira se upisom odgovarajućih vrijednosti u registre sklopa koji su dostupni preko I^2C sabirnice. Sam prijenos zvučnih signala odvija se preko odvojene sabirnice koja se naziva I^2S (engl. *Inter-IC Sound*), a procesor sadrži sklopovlje koje upravlja i prenosi podatke ovom sabirnicom. Moguće je snimanje i reprodukcija 16-bitnog stereo zvuka pri brzini od 192kHz.

Korištenje I^2S periferije isto je kao i kod drugih periferija koje služe za komunikaciju kao što su UART, SPI i slično. Kada šaljemo podatke, čekamo da periferija postane spremna za njih te ih onda jednostavno upisujemo u odgovarajući registar. Čekanje spremnosti može se obavljati u petlji (engl. *busy-wait*) ili koristeći prekidni sustav procesora.

Razvijeni digitalni generator signala može puno brže generirati uzorke nego što ih ovaj sklop može pretvarati u analogni oblik, ali moguće je potvrditi očekivani oblik signala, što je nama dovoljno. Također, generirani signali imaju preciznost od 32-bita, a konverter je 16-bitni pa se ni ovdje ne iskorištava puni potencijal generatora.

5. Programiranje

Možda najvažnija karakteristika bilo kojeg procesora njegov je instrukcijski skup. Svi DSP-ovi, pa tako i ovaj, imaju instrukcijski skup posebno namijenjen učinkovitom izvršavanju algoritama iz područja digitalne obrade signala. U to spadaju FIR[1], IIR[2] i razni drugi filtri, FFT transformacije, kompresije slike i zvuka te mnogi drugi.

U ovom poglavlju prikazat će se osnove pisanja asemblerskih instrukcija za ovaj procesor. Pokazat ćemo i osnovne načine adresiranja, o čemu treba voditi računa pri paralelizaciji instrukcija te kako povezati brze i optimizirane asemblerske dijelove koda sa kodom u višem programskom jeziku C.

Bit će objašnjeni dijelovi korišteni u ovom radu, a više se može naći u [2], [3] i [4].

5.1. Načini adresiranja

Procesor podržava tri načina adresiranja:

apsolutno adresiranje: podatak je adresiran sa konstantom zadanom unutar instrukcijskog koda

direktno adresiranje: podatak je adresiran sa pomakom u odnosu na *DP, SP, ili PDP* registar[3]

indirektno adresiranje: podatak je adresiran koristeći referencu, odnosno njegova adresa smještena je u neki od adresnih registara procesora

Nama je najzanimljivije indirektno adresiranje, koje ćemo koristiti za dohvat koeficijenata kubičnog polinoma. Adresni registri procesora koji su nam raspolaganju za čuvanje adresa su *AR0*, *AR1*, ..., *AR7*.

Pregled osnovnih indirektnih načina adresiranja dan je u tablici 5.1.

[1] Finite Impulse Response

[2] Infinite Impulse Response

[3] redom Data Page, Stack Page, Peripheral Data Page

Tablica 5.1: Indirektni načini adresiranja

**ARn*	adresa podatka je u ARn
**ARn+*	registar ARn uvećan je za 1 nakon dohvata podatka
**ARn-*	registar ARn smanjen je za 1 nakon dohvata podatka
**+ARn*	registar je prvo uvećan za 1, a onda se dohvaća podatak
**-ARn*	registar je prvo smanjen za 1, a onda se dohvaća podatak
**(ARn+T0)*	dohvaća se podatak sa adrese ARn+T0, nakon čega se pomak dodaje na registar
**ARn(T0)*	registar se ne mijenja, a podatak je na adresi ARn+T0

Ovakav način rada s pokazivačima omogućava da adrese operanada budu spremne iz instrukcije u instrukciju. Na primjer, moguće je sa samo jednom MAC (engl. *Multiply-and-Accumulate*) instrukcijom koju ponavljamo u petlji napraviti FIR filtar jer će instrukcija sama, koristeći gore spomenute načine adresiranja, pomicati pokazivač na sljedeće podatke koji nam trebaju. Spomenimo također i da ovaj DSP, kao i većina ostalih, podržava sklopovsko izvođenje petlji pa nema dodatnih instrukcija za oduzimanje brojača i provjeru uvjeta za zaustavljanje.

5.2. Instrukcijski skup

U ovom radu korišten je mnemonički instrukcijski skup, a postoji još i algebarski instrukcijski skup čiji kod podsjeća na pisanje jednadžbi. Tako ćemo, umjesto korištenja instrukcije *add* u mnemoničkom kodu, u algebarskom napisati nešto poput *AC0 = AC0 + AC1*.

U tablici 5.2 može se vidjeti pregled registara koji su nam na raspolaganju.

Tablica 5.2: Registri procesora

Ime	Veličina (bit)	Kratak opis
AC0..AC3	40	akumulatori
T0..T3	16	pomoćni registri
AR0..AR7	16	adresni registri

U akumulatorima procesora čuvaju se brojevi veće preciznosti s kojima trenutno algoritam radi. Svi rezultati množenja idu u akumulatore jer množila daju $17x17 = 34$ bitni rezultat. MAC instrukcije također koriste akumulatore za pohranu međurezultata.

U pomoćnim registrima čuvaju se 16-bitni podaci koji se mogu iskoristiti kao operandi u instrukcijama pomaka, zbrajanja, množenja i još nekih operacija.

U adresnim registrima čuvaju se adrese operanada u memoriji, kao što je opisano u prijašnjem odjeljku.

Pogledajmo primjer načina pisanja konkretne instrukcije. Slika 5.1 izvadak je iz [3], a prikazuje načine pisanja *MPY* instrukcije koja provodi množenje.

No.	Syntax	Parallel Enable Bit	Size	Cycles	Pipeline
[1]	**MPY**[R] [ACx,] ACy	Yes	2	1	X
[2]	**MPY**[R] Tx, [ACx,] ACy	Yes	2	1	X
[3]	**MPYK**[R] K8, [ACx,] ACy	Yes	3	1	X
[4]	**MPYK**[R] K16, [ACx,] ACy	No	4	1	X
[5]	**MPYM**[R] [T3 =]Smem, Cmem, ACx	No	3	1	X
[6]	**MPYM**[R] [T3 =]Smem, [ACx,] ACy	No	3	1	X
[7]	**MPYMK**[R] [T3 =]Smem, K8, ACx	No	4	1	X
[8]	**MPYM**[R][40] [T3 =][uns(]Xmem[)], [uns(]Ymem[)], ACx	No	4	1	X
[9]	**MPYM**[R][U] [T3 =]Smem, Tx, ACx	No	3	1	X

Slika 5.1: MPY instrukcija

Kratka pojašnjenja pojmova sa slike mogu se vidjeti u tablici 5.3.

Tablica 5.3: Pojašnjenja pisanja instrukcije

[]	dio unutra ovih zagrada smije se koristiti kod pisanja instrukcije, ali ne mora
uns()	operandi se proširuju do 17 bita dodavanjem nule
40	svih 40 bita akumulatora se koristi
R	uključi zaokruživanje
Smem,Cmem	podatak iz memorije
Xmem,Ymem	istovremeni dohvat dva podatka
K8,K16	8 odnosno 16-bitna konstanta

Kod pisanja programa prvo u [3] pronađemo instrukciju koja bi nam mogla pomoći te pogledamo u kakvim se sve oblicima ona može pisati. Ovo je važno za pripremanje operanada instrukcije u odgovarajuće registre ili memorijske lokacije. Potrebno je paziti na potrebne širine rezultata i operanada, a važno je i da se oblik uklapa u

ostale instrukcije koje koristimo u programu zato da izbjegnemo dodavanje dodatnih instrukcija koje pripremaju operande i rezultate od instrukcije do instrukcije.

Vrlo je važno složiti instrukcije tako da maksimalno iskoristimo mogućnosti paralelizacije koje nudi ovaj procesor, a o čemu će biti govora u nastavku.

5.2.1. Paralelizacija

Postoje dva osnovna načina paralelizacije instrukcija.

Prvi je način već ugrađeni paralelizam (engl. *built-in parallelism*), kod kojeg jedna instrukcija sama koristi više resursa i izvodi dvije operacije istovremeno. Ovakva instrukcija ima format u kojem su pojedinačne operacije odvojene sa '::'. Slika 5.2 izvod je iz [3] i prikazuje jednu takvu instrukciju.

MPY::MPY *Parallel Multiplies*

Syntax Characteristics

No.	Syntax	Parallel Enable Bit	Size	Cycles	Pipeline
[1]	**MPY**[R][40] [uns()]Xmem[)], [uns()]Cmem[)], ACx **::** **MPY**[R][40] [uns()]Ymem[)], [uns()]Cmem[)], ACy	No	4	1	X

Slika 5.2: MPY::MPY instrukcija

Ova instrukcija provodi dva množenja istovremeno:

$$AC0 = Xmem * Cmem$$
$$AC1 = Ymem * Cmem$$

Za razliku od ovakve paralelizacije, postoji i korisnički određena paralelizacija (engl. *user-defined parallelism*). U ovom slučaju korisnik sam odabire dvije različite instrukcije za koje želi da se izvedu istovremeno, pri čemu je važno da se resursi koje one zauzimaju ne preklapaju. Točnije, dvije instrukcije smiju se paralelizirati ako i samo ako:

- suma duljina njihovih operacijskih kodova nije veća od 6
- barem jedna od njih sadrži bit za omogućavanje paralelizacije (engl. *Parallel enable bit*)
- nijedna instrukcija ne smije koristiti adresiranje koje koristi 16-bitnu ili veću konstantu
- ne mijenjaju isti registar ili memorijsku lokaciju

– ne koriste isti dio procesora; na primjer moguće je paralelizirati 16-bitno zbrajanje u A-jedinici te 40-bitno zbrajanje u D-jedinici, ali ne i dva zbrajanja istog tipa

– instrukcije smiju generirati maksimalno:
  - dvije podatkovne adrese
  - jednu adresu koeficijenta
  - jednu adresu na stogu

Ako želimo paralelizirati dvije instrukcije na ovaj način, u asembleru ih odvajamo sa '||'.

Dvije spomenute vrste paralelizacije moguće je i kombinirati. Naime, kod već ugrađenog paralelizma izvršavamo ustvari jednu instrukciju, koja se može paralelizirati s drugom instrukcijom ako vrijede pravila navedena gore.

Prevoditelj nam neće dopustiti da paraleliziramo dvije instrukcije koje ne zadovoljavaju navedena pravila, ali pravila je potrebno poznavati tako da dobijemo ideju u kojem smjeru razvijati rješenje.

5.3. Značajke jezika C

Kompleksnije programe mnogo je lakše pisati u višem programskom jeziku, kao što je C, nego u asembleru. Traženje grešaka i kasnije održavanje je jednostavnije jer je kod mnogo pregledniji i intuitivniji.

Međutim, treba imati na umu da je ciljna arhitektura DSP procesor, a ne nekakav operacijski sustav. Valja voditi brigu o nekim stvarima koje inače nisu toliko važne, kao što su širine pojedinog tipa podataka, u kojoj vrsti memorije je spremljena koja varijabla, kako pisati prekidne rutine i slično. Ovdje ćemo opisati najvažnije karakteristike jezika C za korišteni DSP procesor.

Tablica 5.4: Tipovi podataka u C-u

Tip	Veličina (bit)
char, int	16
long	32
long long	40
float, double, long double	32

U tablici 5.4 dane su širine najčešće korištenih tipova podataka. Primjetimo da tip *char* zauzima 16-bitova, a ne 8. Također, *long long* zauzima 40-bitova jer je upravo to veličina akumulatora procesora.

Ključne riječi koje ne nalazimo u standardnom C jeziku možemo vidjeti u tablici 5.5.

Tablica 5.5: Nestandardne ključne riječi

Ključna riječ	Kratak opis
ioport	varijabla deklarirana koristeći ovu ključnu riječ bit će pohranjena u I/O memoriji
onchip	16
restrict	jamčimo prevodiocu da pokazivač deklariran na ovaj način jedini pokazuje na određene podatke
interrupt	koristi se za deklaraciju prekidnih rutina

5.3.1. Povezivanje asemblera i C-a

Dijelove koda za koje želimo da su izrazito učinkoviti, najčešće zato jer se ponavljaju više puta, poželjno je napisati u asembleru. Čovjek svojom inteligencijom i kreativnošću može puno bolje iskoristiti sve mogućnosti procesora nego što to može prevoditelj. U nastavku ćemo pokazati kako iz C-a pozvati funkciju napisanu u asembleru.

Pri pisanju funkcije potrebno je pridržavati se nekoliko važnih pravila koji govore o prijenosu parametara, vraćanju rezultata te očuvanju konteksta.

Registri kojima funkcija mora sačuvati vrijednost su:

- pomoćni registri T2 i T3
- adresni registri AR5, AR6, AR7
- te pokazivač stoga SP

Ako funkcija uopće ne koristi spomenute registre, nema potrebe za njihovim spremanjem. No, ako koristi neke od njih, tim registrima potrebno je prije korištenja negdje zapamtiti vrijednost te im je povratiti prije izlaza iz funkcije. Vrijednosti se najčešće pamte na stogu, ali mogu se zapamtiti i u memoriji. Na pokazivač stoga ne treba posebno obraćati pažnju jer će on ostati isti ako sve što smo stavili na stog ujedno i uklonimo s njega, što se u praksi najčešće i radi.

U tablici 5.6 možemo vidjeti kako se prenose argumenti u funkciju, odnosno u kojim se registrima oni pohranjuju prije poziva. Vidimo da se preklapaju registri kod

Tablica 5.6: Prijenost parametara u funkciju

Tip podatka	Registar
16-bitni podatak	T0, T1, AR0, AR1, AR2, AR3, AR4
32-bitni podatak	AC0, AC1, AC2
Pokazivač	AR0, AR1, AR2, AR3, AR4

16-bitnih podataka te kod pokazivača. Prevoditelj slaže parametre po redu kojim su dani u deklaraciji funkcije: ako nakon jednog pokazivača dolaze tri 16-bitna podatka, treći podatak bit će spremljen u registar AR1, jer je AR0 već zauzet pokazivačem.

Funkcija vraća vrijednost u sljedećim registrima:

- 16-bitni podatak u T0
- 32-bitni podatak u AC0
- pokazivač u AR0

Ovdje nećemo ulaziti u manipulaciju strukturama, detalji o tome mogu se pronaći u [4].

Funkcije ili globalne varijable moguće je koristiti unutar asemblerske funkcije, ali je potrebno držati se nekoliko nekoliko pravila navedenih u nastavku. Ovdje pretpostavljamo da je asemblerska funkcija u zasebnoj datoteci, što je najurednije i najčešće rješenje.

- varijable i funkcije definirane u C-u u asembleru imaju prefiks '_'
- varijable i funkcije potrebno je učiniti vidljivima unutar asemblerske datoteke koristeći makro *.global*
- ime funkcije treba definirati kao globalni simbol koristeći makro *.def*
- u C-u je potrebno deklarirati prototip funkcije

Ovo sve demonstrirat ćemo na jednostavnom primjeru. Iz C-a pozivamo funkciju pisanu u asembleru koja zbroji dani argument s globalnom varijablom deklariranom u C-u. Neka se datoteke u kojima su kodovi zovu *main.c* i *func.asm*.

Datoteka func.asm

```
  .def _sum
  .global _x
_sum:
  mov #(_x), ar0
  add *ar0, t0
  ret
```

Datoteka main.c

```
int x = 2;
extern int sum( int );

void main(void) {
  sum( 3 ); // funkcija vraca x+3=5 kao rezultat
}
```

Argument funkcije je 16-bitni podatak pa se sprema u registar T0. U adresni registar AR0 pohranimo adresu globalne varijable *x*, a zatim njenu vrijednost dodamo na T0. U T0 se i vraća rezultat, pa sve ostavljamo kako je i odlazimo iz funkcije naredbom *ret*.

5.4. Chip Support Library

Ova biblioteka sadrži funkcije pisane u C-u za konfiguriranje i kontrolu periferija na procesoru te pruža puno veću razinu apstrakcije od alternativnog direktnog upisivanja bitova u pojedine registre i uvelike pomaže razvoju programske potpore. Također, kod je prenosiv na druge procesore bez većih izmjena.

Biblioteku ćemo iskoristiti za pozivanje PSAC funkcije u jednakim vremenskim intervalima koristeći brojilo ugrađeno u procesor te njegov prekidni sustav.

5.4.1. Brojilo

Periferija u kojoj je brojilo ima naziv GPT(engl. *General Purpose Timer*). Rad započinjemo pozivom funkcije *GPT_open* kojoj predajemo oznaku brojila kojeg želimo koristiti (naš procesor ima tri brojila) te objekt koji CSL koristi za pohranu podataka o periferiji. Funkcija nam zatim vraća identifikator periferije (engl. *handle*) koji ubuduće predajemo ostalim funkcijama za rad s brojilom.

Rad sa brojilom u CSL-u

```
#include <csl_gpt.h>

void main(void) {
  CSL_GptObj gptObj;
  CSL_Handle hGpt;
  CSL_Status status = 0;
  CSL_Config hwConfig;

  hwConfig.autoLoad     = GPT_AUTO_ENABLE;
  hwConfig.ctrlTim      = GPT_TIMER_ENABLE;
  hwConfig.preScaleDiv  = GPT_PRE_SC_DIV_0;
  hwConfig.prdLow       = 0xFFFF;
```

```
  hwConfig.prdHigh      = 0x0000;

  hGpt = GPT_open( GPT_0, &gptObj, &status );
  status = GPT_config( hGpt, &hwConfig );
  status = GPT_start( hGpt );

  // trenutna vrijednost brojila
  GPT_getCnt( hGpt, &cnt );
}
```

Nakon što smo inicijalizirali brojilo konfiguriramo ga sa željenim parametrima. Ovo radimo popunjavanjem strukture *hwConfig* koju nakon toga šaljemo funkciji *GPT_config*. U primjeru gore brojilo smo konfigurirali tako da kreće ispočetka kad odbroji do kraja, da koristi takt procesora te da broji do 0xFFFF. Sad smo spremni za pokretanje brojila, što radimo pozivom *GPT_start*. U bilo kojem trenutku možemo saznati na kojoj je trenutno vrijednosti brojilo koristeći funkciju *GPT_getCnt*.

Na sličan način radi se i s drugim periferijama, samo što se prefiks imena funkcije mijenja sukladno korištenoj periferiji, na primjer *GPIO_open* ili *I2C_init*.

Sljedeće što nam treba jest da brojilo neprestano poziva neki korisni kod, za što koristimo prekidni sustav procesora.

5.4.2. Prekidne rutine

Prekidna rutina mora obavezno biti tipa *void*, ne smije primati ni jedan parametar i mora biti deklarirana sa ključnom riječi *interrupt*.

Rad s prekidnim rutinama u CSL-u

```
#include <csl_intc.h>

extern void VECSTART(void);

interrupt void myIsr() {
  // kod rutine
}

void main(void) {
  IRQ_setVecs((Uint32)(&VECSTART));
  IRQ_plug(TINT_EVENT, &myIsr);
  IRQ_enable(TINT_EVENT);
  IRQ_globalEnable();
}
```

Funkcija *IRQ_setVecs* služi za inicijalizaciju prekidnih vektora i koristi se kao što je prikazano u primjeru. Sa *IRQ_plug* vežemo izvršavanje prekidne rutine za određeni događaj. U ovom primjeru taj događaj je *TINT_EVENT*, odnosno odbrojavanje brojila do kraja svog perioda. Na kraju je potrebno uključiti izvršavanje prekida.

5.5. Board Support Library

Board Support Library (BSL) malena je biblioteka za jednostavno povezivanje DSP procesora i ostalog sklopovlja koje se nalazi na razvojnoj pločici. Ovakve biblioteke gotovo uvijek dolaze s evaluacijskim sklopovljem te uvelike olakšavaju korisniku rad sa razvojnom pločicom. Korisnik tako ne mora proučavati sheme pločice i tražiti na koje priključke procesora je što spojeno te kako su međusobno povezani ostali sklopovi na pločici, već ovi podaci dolaze ugrađeni u funkcije i definicije biblioteke.

BSL ponuđen za ovu pločicu sadrži jednostavne funkcije za pisanje i čitanje s vanjskih priključaka dostupnih na pločici, za upravljanje LED diodom, za rad sa EEPROM čipom te za komunikaciju sa AIC3204 audio kodekom. Ovo posljednje bit će nam od najvećeg značaja.

U biblioteci su dane funkcije koje, koristeći I^2C sabirnicu, čitaju i pišu kontrolne registre AIC sklopa. Dan je i primjer osnovne konfiguracije koja je korištena za testiranje generatora signala.

6. Razvijeni generator signala

Kao što je već rečeno, temelj generatora signala je PSAC, odnosno izračun aproksimacije amplitude sinusa za zadanu fazu. U sklopu rada u asembleru su napisane četiri PSAC funkcije, a u sljedećim odjeljcima bit će opisan način rada s ovim funkcijama, njihova učinkovitost te način na koji je testiran ispravan rad.

6.1. Pregled napisanih funkcija

Pregled svih napisanih funkcija dan je u tablici 6.1. Za brojanje ciklusa u ovoj i ostalim tablicama koristi se mogućnost *profiling*-a ugrađena u *Code Composer Studio*.

Sve funkcije su pisane u asembleru i pažljivo optimizirane, a njihovi prototipovi dani su u nastavku.

Prototipovi funkcija za računanje PSAC-a

```
long psac16( long, const int[][6] );
long psac17( long, const int[][6] );
void psac16_2( long, long, long*, long*, const int[][6] );
void psac17_2( long, long, long*, long*, const int[][6] );
```

Zadnji argument svake od funkcija pokazivač je na niz koeficijenata. Niz mora biti složen tako da prva dva koeficijenta zauzimaju po 16-bita, a druga dva po 32-bita, što daje ukupno 6 *int*-ova odnosno 12 byte-ova. Ovo je moguće zbog različitih rezolucija koeficijenata, o čemu je bilo govora u poglavlju 2.

Funkcije koje imaju dva ulaza računaju dvije amplitude odjednom i vraćaju ih preko danih pokazivača. One se koriste kod amplitudne i frekvencijske modulacije kod kojih je moguće neovisno računati dva PSAC-a. Kod fazne modulacije ovo nije slučaj jer vrijednost prvog PSAC-a ulazi kao argument u drugi PSAC. Istovremeno računanje dva argumenta može se iskoristiti i za računanje kvadraturne modulacije (engl. *QAM*) kod koje se koriste dva vala nosača pomaknuta u fazi za 90 stupnjeva. Poziv ovih funkcija učinkovitiji je od dva poziva funkcije koja računa jedan PSAC jer se bolje iskorištavaju mogućnosti paralelizacije procesora što se i vidi iz broja ciklusa.

Ovime štedimo i instrukcije potrebne za pripremu argumenata te poziv funkcije iz C-a.

Tablica 6.1: Pregled PSAC funkcija

Ime	Širina faze (*bit*)	Broj ulaza	Broj Ciklusa
psac16	26	1	27
psac17	27	1	30
psac16_2	26	2	38
psac17_2	27	2	53

Funkcije koje primaju 26-bitnu fazu nešto su učinkovitije jer je argument polinoma tada 16-bitni što čini rad s njima jednostavnijim. Gubitak preciznosti u odnosu na 27-bitnu fazu je minimalan. Sve funkcije daju 32-bitni izlaz.

Očekuje se da argument funkcije ima nule na svim mjestima osim onih od značaja za izračun. Dakle, funkcije koje primaju 26-bitnu fazu ne rade ispravno ukoliko bitovi težine veće od 26 nisu nula, a isto vrijedi i za 27-bitni argument. Ovaj uvjet mora osigurati pozivatelj funkcije.

Tablica 6.2: Maksimalna postignuta brzina

Ime	Širina faze (*bit*)	Broj Ciklusa	Brzina (*ksps*)
run16	26	30	3333
run17	27	33	3030

Pisanjem isključivo u asembleru, bez pozivanja funkcija iz C-a, moguće je malo dobiti na brzini jer ne gubimo na dodatnim instrukcijama vezanim za pozivanje funkcije (engl. *overhead*). Napisana su dva takva programa u kojima se unutar petlje pisane u asembleru mijenja faza za zadani korak. Funkcije za jedini argument primaju željeni korak koji se dodaje na fazu i vrte se beskonačno. Njihove performanse mogu se vidjeti u tablici 6.2. Ove brzine samo su teoretske, jer programi računaju PSAC, ali ne rade ništa korisno s njim. Tu su zbog ilustracije pristupa pisanja samo u asembleru.

Brzina u tablicama izražena je u tisućama uzoraka po sekundi (engl. *kilosamples per second*) i dobivena je formulom 6.1 uz pretpostavljeni takt procesora od 100mHz. Brzina pretpostavlja jednu instrukciju po ciklusu procesora, što je blizu stvarnosti za ovaj procesor.

$$brzina = \frac{f_{takt}}{brojCiklusa} \tag{6.1}$$

Najčešće ćemo u praksi PSAC funkcije pozivati iz C-a u pravilnim vremenskim razmacima koristeći brojilo i prekidnu rutinu. Na ovaj način procesor može obavljati i druge zadatke kad nije zauzet generiranjem signala. Ako nam učinkovitost nije toliko kritična, ovakvo rješenje puno je elegantnije i lakše za razvoj. Potrebno je podesiti period brojila i u prekidnu rutinu smjestiti poziv PSAC funkcije te željenu promjenu faze za ciljni oblik signala. Brojilo ne smije imati period toliko mali da se funkcija ne stigne izvršiti. Ovdje treba napomenuti da se broj ciklusa odnosi samo na kod unutar funkcije te da treba paziti na dodatne instrukcije potrebne za pozivanje i izlaz iz prekidne rutine.

Maksimalne moguće brzine ovakvog pristupa dane su u tablici 6.3.

Način računanja modulacija dan je prethodno u poglavlju 3, a ovdje možemo vidjeti konkretnu implementaciju za frekvencijsku modulaciju s 26-bitnom fazom.

```
void tap_fm16() {
  long psac1_output, psac2_output;
  psac16_2( phaseA, phaseB, &psac1_output, &psac2_output, coef );
  output = psac1_output;
  phaseA = (phaseA + deltaA + (psac2_output>>9))&0x3ffffff;
  phaseB = (phaseB + deltaB)&0x3ffffff;
}
```

Nakon svakog dodavanja koraka fazi potrebno je osigurati da su svi bitovi osim prvih 26 jednaki nula.

Tablica 6.3: Brzine po modulacijama

Širina faze	Tip modulacije	Broj Ciklusa	Brzina (*ksps*)
26	Amplitudna	85	1176
26	Frekvencijska	69	1449
26	Fazna	78	1282
27	Amplitudna	99	1010
27	Frekvencijska	84	1190
27	Fazna	88	1136

6.2. Funkcija psac16

Funkcija *psac16* prima dva argumenta: fazu za koju želimo izračunati amplitudu sinusa te pokazivač na niz koeficijenata. Prvi argument prevoditelj sprema u akumulator *AC0*, a drugi u adresni registar *AR0*. Ovo je u skladu sa pravilima navedenim u 5.3.1.

Navedimo ovdje i da funkcija *psac16_2* prva dva argumenta sprema u *AC0* i *AC1*, a tri pokazivača redom u *AR0*, *AR1* te *AR2*.

Iz 26-bitne faze koja se nalazi u akumulatoru potrebno je izdvojiti prvih 10 te zadnjih 16 bitova. Prvih 10 reći će nam kojem segmentu pripada ova faza, a zadnjih 16 argument s kojim ulazimo u polinom. Segment nam direktno govori koji koeficijenti nam trebaju i njega koristimo za izračun adrese tih koeficijenata u cijelom nizu. Prisjetimo se da je za dobivanje argumenta polinoma u frakcionalnoj notaciji potrebno komplementirati prvi bit. Instrukcije koje slijede obavljaju gore navedene operacije.

```
xor #8000h, ac0, t0 || mpyk #6, ac0, ac1
add ac1, ar0
```

Argument polinoma nalazi se u zadnjih 16 bitova akumulatora *AC0*. Prva instrukcija promijeni prvi bit i stavi rezultat u *T0*. Kao što je objašnjeno u [3], ovako napisana instrukcija *xor* koristi zadnjih 16-bitova akumulatora što nam odgovara. Instrukcija *mpyk* množi akumulator *AC0* sa 6 i rezultat sprema u *AC1*. Instrukcije množenja koriste bitove akumulatora $[32 - 16]$ pa nema potrebe za posebnim izdvajanjem bitova da bismo došli do oznake segmenta. Instrukcija množenja paralelno se izvodi sa *xor* instrukcijom što je moguće jer množenje koristi množilo iz D-jedinice, a *xor* 16-bitni ALU iz A-jedinice. Nakon množenja sa 6 dodajemo pokazivač na niz koeficijenata te tako dolazimo do adrese koeficijenata za željeni segment.

Sad možemo nastaviti sa računanjem; množimo argument polinoma s prvim koeficijentom te poravnavamo i dodajemo drugi koeficijent. Tada opet množimo s argumentom, i dodajemo treći koeficijent.

```
mpym *ar0+, t0, ac0
add *ar0+ << #16, ac0

mpy t0, ac0
sfts ac0, #-16
add dbl(*ar0+), ac0
```

Adresiranje **ar0+* dohvaća koeficijent na toj adresi i pomakne adresni registar na sljedeći koeficijent. Napomenimo da, ako se adresiranje koristi sa konstruktom *dbl* koji označava 32-bitni dohvat, tad se adresa uvećava za 2, a ne za 1.

Kao što je i prije spomenuto, ako je operand množenja akumulator koristi se njegovih 17 bitova iz intervala $[32 - 16]$. Na ovo treba pripaziti, odnosno bitovi za koje želimo da uđu u množenje moraju prije samog množenja biti na dobroj poziciji što osiguravamo odgovarajućim pomacima. Napomenimo da u ova dva bloka nije bilo moguće iskoristiti "zbroji-i-pomnoži" (engl. *MAC*) instrukciju zbog mjesta gdje se nalaze operandi te zbog potrebnih poravnavanja između množenja i zbrajanja.

Za sada imamo broj $(a\Delta x + b) \cdot \Delta x + c$ spremljen u akumulator *ac0*. Dosad su sva množenja mogla biti izvršena koristeći 17x17 množilo procesora, no trenutni nam je rezultat širi i morat ćemo pristupiti drugačije. Množimo 32-bitni broj sa 16-bitnim argumentom polinoma koji se nalazi u *T0*. Ovo radimo tako da podijelimo veliki broj na dva dijela od po 16 bitova, napravimo dva množenja te na kraju zbrojimo dobivena dva međurezultata. Da bismo dobili točan rezultat, kod zbrajanja treba prvi međurezultat pomaknuti za 16 bitova ulijevo.

```
and #65535, ac0, ac1
sftl ac1, #16

mpy t0, ac0 || mpy t0, ac1

add ac0 << #16, ac1
sfts ac1, #-16
add dbl(*ar0+), ac1, ac0
```

Dva množenja mogu se izvršavati istovremeno jer procesor sadrži dva neovisna množila. Konačan rezultat nalazi se u *ac0*.

Za točno izvođenje ove funkcije potrebno je postaviti i neke statusne bitove. Ova postavljanja pokušavamo smjestiti paralelno sa izvođenjem drugih instrukcija, ali pazimo da budu postavljeni na vrijeme, odnosno prije nego što se pojavi instrukcija za čije izvođenje je važan određeni statusni bit. Bitovi koje je potrebno postaviti su:

frct: uključuje pomak za jedan u lijevo nakon svakog množenja

m40: nalaže procesoru da koristi svih 40 bitova akumulatora

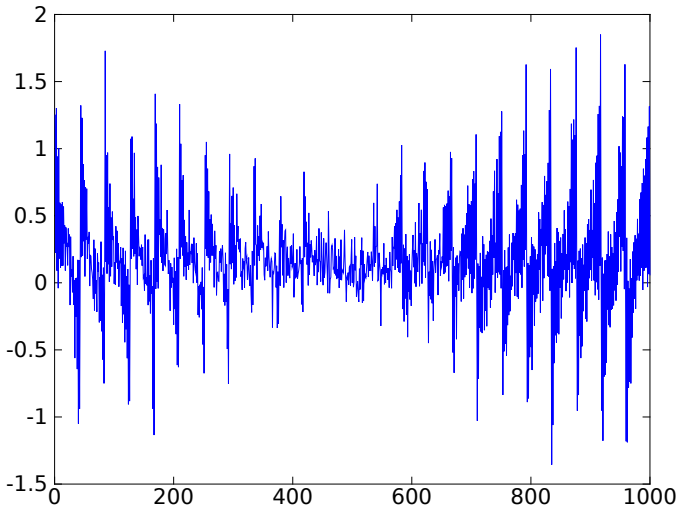
satd: uključuje saturaciju pri zbrajanju

Prije izlaska iz funkcije obrisat ćemo statusne bitove koje smo postavili tako da po izlasku iz funkcije prevoditelju ostavimo sve kako je bilo jer inače može doći do nepredviđenog ponašanja prevedeneg koda. Ako pišemo samo u asembleru, moguće ih je izostaviti.

Kompletan kod funkcije ove i drugih funkcija može se vidjeti u dodatku A.

6.3. Testiranje

Sve četiri verzije PSAC funkcija testirane su generiranjem velikog broja uzoraka te uspoređivanjem s referentnim programom u C-u. Osim ovoga, izlazna sinusoida promatrana je na osciloskopu te je izgledala očekivano. Generirana sinusoida frekvencije od nekoliko kiloherca može se čuti i na zvučnicima.



Slika 6.1: Graf LSB pogreške

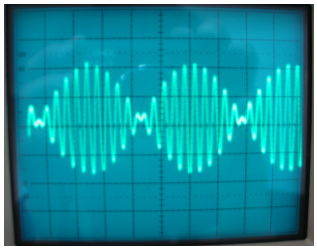
Tablica 6.4 prikazuje vrijednosti amplitude dobivene za 6 uzoraka jednoliko raspoređenih po cijeloj periodi, a slika 6.1 LSB pogrešku za 1000 uzoraka. LSB pogreška dobivena je dijeljenjem apsolutne pogreške sa 2^{-29}.

Tablica 6.4: Usporedba funkcije psac16 i sinus funkcije iz C-a

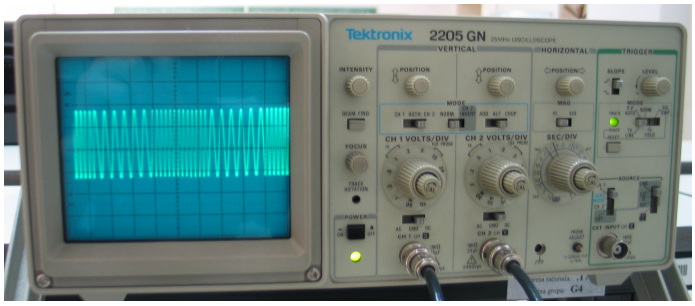
Faza ($\cdot\frac{2\pi}{6}$)	C funkcija	psac16	Pogreška ($\cdot 10^{-9}$)	LSB pogreška
0	0.00000000000	-0.00000000232	2.3283	1.2500
1	0.86602537257	0.86602537240	0.1656	0.0889
2	0.86602546620	0.86602546600	0.1993	0.1070
3	0.00000018726	0.00000018440	2.8614	1.5362
4	-0.86602527894	-0.86602527974	0.7995	0.4292
5	-0.86602555983	-0.86602556007	0.2327	0.1249

Slika 6.2 prikazuje signal moduliran amplitudnom modulacijom. Za dobivanje ove slike oba signala generirana su i modulirana na DSP-u.

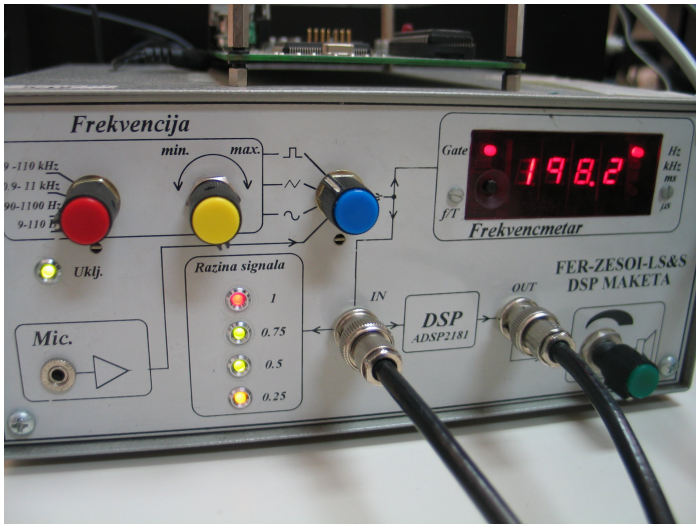
Na slici 6.3 može se vidjeti osciloskop korišten za testiranje te frekventno modulirani signal na njegovom zaslonu. Ovaj put val nosač je generiran na DSP-u, a modulirajući signal na generatoru signala koji vidimo na slici 6.4. Modulirajući signal sastojao se od pravokutnih impulsa što se može lijepo razabrati i na rezultatu modulacije.



Slika 6.2: Amplitudna modulacija vidljiva na osciloskopu



Slika 6.3: Frekvencijska modulacija vidljiva na osciloskopu



Slika 6.4: Korišteni generator analognog signala

7. Zaključak

U ovom radu pokazano je da se arhitektura DSP procesora značajno razlikuje od arhitektura ostalih procesora. Ove posebnosti u arhitekturi imaju za cijenu nešto složeniji razvoj programske potpore, ali ostavljaju dobrom programeru mjesta za zanimljive i neočekivano učinkovite dijelove koda.

Razvijeni generator signala vrlo je učinkovit ako uzmemo u obzir visoku izlaznu preciznost. U najjednostavnijoj varijanti može biti korišten kao generator općenitog sinusnog signala čiji izlaz se dalje može iskoristiti za testiranje drugih sustava. Mogućnost modulacije nudi i primjenu kod testiranja analognih ili digitalnih demodulatora.

Korištenje D/A konvertera za dobivanje analognog signala na izlazu ne iskorištava u potpunosti brzinu i preciznost koju generator može dati. Pune mogućnosti mogle bi se iskoristiti u nekim algoritmima digitalne obrade signala.

Ovim radom dan je cjelokupni uvid u razvoj rješenja koristeći DSP procesor te može biti od velike koristi studentima pa čak i inženjerima koji imaju potrebu za razvojem u sličnoj ili istoj okolini.

LITERATURA

[1] Texas Instruments. http://www.ti.com.

[2] TMS320C55x DSP Programmer's Guide. http://focus.ti.com/lit/ug/spru376a/spru376a.pdf, Kolovoz 2001.

[3] TMS320C55x DSP Mnemonic Instruction Set Reference Guide. http://focus.ti.com/lit/ug/spru374g/spru374g.pdf, Listopad 2002.

[4] TMS320C55x Optimizing C/C++ Compiler User's Guide. http://focus.ti.com/lit/ug/spru281f/spru281f.pdf, Prosinac 2003.

[5] TMS320C55x DSP CPU Reference Guide. http://www.ti.com/litv/pdf/spru371f, Veljača 2004.

[6] http://en.wikipedia.org/wiki/Fixed-point_arithmetic, Svibanj 2011.

[7] http://en.wikipedia.org/wiki/Modulation, Svibanj 2011.

[8] Davor Petrinović i Marko Brezović. Spline based high-accuracy piecewise-polynomial phase to sinusoid amplitude converters. TUFFC-03290-2009.R2, 2009.

[9] Antonela Šipić. Direktna digitalna sinteza iznimno visoke točnosti. Diplomski rad, Fakultet elektrotehnike i računarstva, 2009.

Digitalni generator sinusnog signala s amplitudnom, faznom i frekvencijskom modulacijom izveden DSP procesorom

Sažetak

DSP procesori izrazito su učinkoviti kod izvođenja računski zahtjevnih algoritama iz područja digitalne obrade signala. Ovim radom je kroz razvoj generatora digitalnog signala pokazano kako iskoristiti to što ovi procesori nude.

Signal se generira računanjem amplitude sinusa za fazu koju jednoliko mijenjamo kroz vrijeme. Računanje sinusa provodi se dijeljenjem periode na određeni broj segmenata te aproksimiranjem sinusoide kubičnim polinomom unutar svakog segmenta. Koeficijenti polinoma dobiveni su kubičnom B-spline interpolacijom. Koristeći amplitudnu, frekvencijsku ili faznu modulaciju, mogu se kombinirati dvije generirane sinusoide u jedan signal.

Računanje amplitude izvedeno je funkcijama napisanim u asembleru koje se onda pozivaju iz C-a za dobivanje željenog valnog oblika ili ostvarivanje modulacije.

Ključne riječi: PSAC, B-spline, TMS320VC5505

Digital signal generator with amplitude, phase and frequency modulation implemented on DSP

Abstract

DSP processors can perform digital signal processing algorithms very efficiently. Throughout the development of digital signal generator, this document explains how to exploit the processing power of these processors.

Signal is generated by continuously adding a constant to the phase and calculating the amplitude of the sine wave for each phase. Calculation is performed by dividing the period to smaller intervals and aproximating within each interval using third degree polynomial. Polynomial coefficients are obtained using B-spline interpolation. By using amplitude, phase or frequency modulation, two signals can be combined into one.

Assembly language is used so that extensive calculations can be done very fast. Subroutines written in assembly are then called from C in order to provide simple and intuitive way of generating desired signals.

Keywords: PSAC, B-spline, TMS320VC5505

Dodatak A
Kod PSAC funkcija

Funkcija psac16

```
xor #8000h, ac0, t0 || mpyk #6, ac0, ac1
add ac1, ar0

bset frct

bset m40 || mpym *ar0+, t0, ac0
add *ar0+ << #16, ac0

mpy t0, ac0
sfts ac0, #-16
add dbl(*ar0+), ac0

and #65535, ac0, ac1
sftl ac1, #16
mpy t0, ac0 || mpy t0, ac1
bclr frct || add ac0 << #16, ac1
bset satd || sfts ac1, #-16

bclr m40
add dbl(*ar0+), ac1, ac0
bclr satd
ret
```

Funkcija psac16_2

```
mov xar2, xar3

xor #8000h, ac0, t0 || mpyk #6, ac0, ac2
add ac2, ar2

xor #8000h, ac1, t1 || mpyk #6, ac1, ac2
add ac2, ar3

bset frct

mpym *ar2+, t0, ac0 || mpym *ar3+, t1, ac1
bset m40 || add *ar2+ << #16, ac0
```

```
add *ar3+ << #16, ac1

mpy t0, ac0 || mpy t1, ac1
sfts ac0, #-16
sfts ac1, #-16 || add dbl(*ar2+), ac0
add dbl(*ar3+), ac1

and #65535, ac0, ac2
sftl ac2, #16 || and #65535, ac1, ac3
sftl ac3, #16

mpy t0, ac0 || mpy t0, ac2

mpy t1, ac1 || add ac0 << #16, ac2
mpy t1, ac3 || sfts ac2, #-16

add ac1 << #16, ac3
bset satd || sfts ac3, #-16

bclr m40
add dbl(*ar2+), ac2, ac0
add dbl(*ar3+), ac3, ac1

bclr satd || mov ac0, dbl(*ar0)
bclr frct || mov ac1, dbl(*ar1)

ret
```

Funkcija psac17

```
bset m40 || sftl ac0, #-17, ac1
sub ac1 << #17, ac0
xor #1 << #16, ac0 || sftl ac1, #16

mpyk #6, ac1 || sftl ac0, #16
add ac1, ar0

mpym *ar0+, ac0, ac2
add *ar0+ << #16, ac2

mpy ac0, ac2
sfts ac2, #-16
add dbl(*ar0+), ac2

and #65535, ac2, ac1
sftl ac1, #16
mpy ac0, ac2 || mpy ac0, ac1
add ac2 << #16, ac1
bset satd || sfts ac1, #-16
bclr m40
add dbl(*ar0+), ac1, ac0

bclr satd
ret
```

Funkcija psac17_2

```
mov xar2, xar3

bset m40 || sftl ac0, #-17, ac2
sub ac2 << #17, ac0

xor #1 << #16, ac0 || sftl ac1, #-17, ac3

sub ac3 << #17, ac1
xor #1 << #16, ac1 || sftl ac2, #16

mpyk #6, ac2 || sftl ac0, #16
add ac2, ar2

sftl ac3, #16 || mpym *ar2+, ac0, ac2

mpyk #6, ac3 || sftl ac1, #16
add ac3, ar3

add *ar2+ << #16, ac2

mpy ac0, ac2 || mpym *ar3+, ac1, ac3
add *ar3+ << #16, ac3

sfts ac2, #-16 || mpy ac1, ac3
sfts ac3, #-16 || add dbl(*ar2+), ac2

add dbl(*ar3+), ac3

psh dbl( ac3 )
and #65535, ac2, ac3
sftl ac3, #16

mpy ac0, ac2 || mpy ac0, ac3
add ac2 << #16, ac3
bset satd || sfts ac3, #-16
bclr m40
add dbl(*ar2+), ac3, ac0
pop dbl( ac3 )

and #65535, ac3, ac2
bset m40 || sftl ac2, #16

mpy ac1, ac3 || mpy ac1, ac2
bclr satd
add ac3 << #16, ac2
bset satd || sfts ac2, #-16
bclr m40
add dbl(*ar3+), ac2, ac1

mov ac0, dbl(*ar0)
bclr satd || mov ac1, dbl(*ar1)

ret
```

Dodatak B
C kod za modulaciju

```
const int coef[][6]; \\ ovdje inace idu koeficijenti, tu nisu navedeni zbog
preglednosti

long psac16( long, const int[][6] );
long psac17( long, const int[][6] );
void psac16_2( long, long, long*, long*, const int[][6] );
void psac17_2( long, long, long*, long*, const int[][6] );
void run16( long );
void run17( long );

long phaseA = 0, phaseB = 0;
long deltaA = 5000000, deltaB = 100000;
long output = 0;

void tap_am16() {
  long psac1_output, psac2_output;
  psac16_2( phaseA, phaseB, &psac1_output, &psac2_output, coef );
  output = ((psac1_output>>16) * (psac2_output>>16))<<1;
  phaseA = (phaseA + deltaA)&0x3ffffff;
  phaseB = (phaseB + deltaB)&0x3ffffff;
}

void tap_fm16() {
  long psac1_output, psac2_output;
  psac16_2( phaseA, phaseB, &psac1_output, &psac2_output, coef );
  output = psac1_output;
  phaseA = (phaseA + deltaA + (psac2_output>>9))&0x3ffffff;
  phaseB = (phaseB + deltaB)&0x3ffffff;
}

void tap_pm16() {
  long m = psac16( phaseB, coef );
  output = psac16( phaseA + (m>>9), coef );
  phaseA = (phaseA+deltaA)&0x3ffffff;
  phaseB = (phaseB+deltaB)&0x3ffffff;
}

void tap_am17() {
  long psac1_output, psac2_output;
```

```
  psac17_2( phaseA, phaseB, &psac1_output, &psac2_output, coef );
  output = ((psac1_output>>16) * (psac2_output>>16))<<1;
  phaseA = (phaseA + deltaA)&0x7fffff;
  phaseB = (phaseB + deltaB)&0x7fffff;
}

void tap_fm17() {
  long psac1_output, psac2_output;
  psac17_2( phaseA, phaseB, &psac1_output, &psac2_output, coef );
  output = psac1_output;
  phaseA = (phaseA + deltaA + (psac2_output>>9))&0x7fffff;
  phaseB = (phaseB + deltaB)&0x7fffff;
}

void tap_pm17() {
  long m = psac17( phaseB, coef );
  output = psac17( phaseA + (m>>9), coef );
  phaseA = (phaseA+deltaA)&0x7fffff;
  phaseB = (phaseB+deltaB)&0x7fffff;
}
```