Hardverska implementacija LVN DS

Digitalni filtri

U narednim vežbama ćemo se baviti hardverskom implementacijom LVN diskretnih sistema. Demonstraciju postupka implementacije LVN DS-a ćemo izvršiti na nekoliko primera i to na:

- Implementaciji digitalnih filtara,
- Implementaciji 2D-binDCT-a.

Hardverska implementacija digitalnih filtara

Postupak implementacije digitalnih filtara ćemo demonstrirati na primeru FIR filtra. Kako bismo se lakše uveli u problematiku prvo ćemo implementirati fiksan FIR u smislu arhitekture. Kada kažemo fiksan mislimo na to da će širina ulaznih podataka i red filtra biti fiksirani. Zatim ćemo na osnovu ovog primera modelovati parametrizovani FIR filtar koji će biti lako prilagoditi potrebama različitih sistema. Parametrizovaćemo red filtra i širinu podataka. Kao što je poznato, parametrizacija nam omogućava jednostavno prilagođavanje projektovanog modula različitim potrebama bez ponovne implementacije modula. Na ovaj način će jednom implementirani FIR biti moguće koristiti u raznim aplikacijama. Na kraju vežbe je data i motivacija za hardversku implementaciju algoritama, to jest, izvršeno je upoređivanje performansi hardverske i softverske implementacije filtra. Ovaj dodatak svakako nije potreban za polaganje ispita, a ovde je dat kako bi zainteresovani studenti stekli širu sliku o hardverskoj akceleraciji algoritama.

Implementacija fiksne arhitekture FIR filtra

Funkcionalna specifikacija sistema koji je potrebno projektovati

Potrebno je projektovati digitalni filtar (u nastavku će ovaj sistem biti nazivan FIR filtar) koji treba da ima sledeću funkcionalnost:

- Filtar treba da bude četvrtog reda
- Potrebno je da širina podataka bude 24 bita (1-celobrojni deo i 23-razlomljeni)
- Koristiti transponovanu direktnu formu

Ovaj format podataka predstavlja standard kvalitetnih AD konvertora koji se koriste u audio tehnici. Gornji tekst se može smatrati funkcionalnom specifikacijom sistema koji je

potrebno projektovati.

Dizajn specifikacija sistema

Na osnovu funkcionalne specifikacije sistema, kreira se dizajn specifikacija. Dizajn specifikacija treba jasno da definiše način na koji će se implementirati željena funkcionalnost.

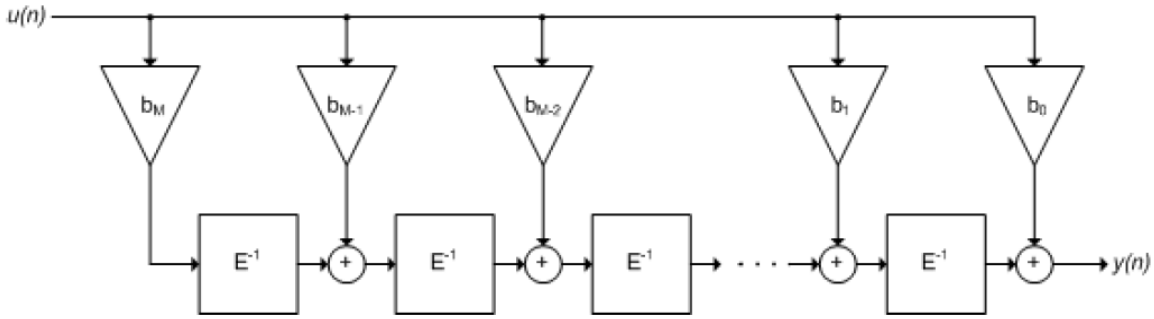
Prvi korak je definisanje interfejsa sistema koji se projektuje. U našem slučaju FIR filtar ima sledeći interfejs:

- *clk_i* – ulazni port preko kojeg se u sistem dovodi globalni sinhronizacioni signal.
- *u_i* – 24-bitni ulazni port za podatke (odbirke).
- *y_o* – 24-bitni izlazni port za podatke (filtrirani podaci).

Interfejs FIR filtra je prikazani na slici 1.

Slika 1: Interfejs FIR filtra.

Sledeći korak u razvoju dizajn specifikacije je modularizacija sistema i definisanje njegove hijerarhijske strukture. Pre nego izvršimo modularizaciju, podsetimo se dijagrama transponovane direktne forme FIR sistema iz vežbe 5 u kojoj smo se bavili sintezom LVN DS-ova. Na slici 2 je, podsećanja radi, prikazana ova šema.



Slika 2: Transponovana direktna forma FIR sistema.

Ovakav prikaz FIR sistema nam omogućava brzo transformisanje u sistem koji je moguće realizovati u hardveru jer svaki element sa slike 2 ima svog para u realnim digitalnim sistemima. Sabirač i množač se direktno preslikavaju u sabirač i množač u hardveru. Blok koji ima ulogu da zakasni signal, $E^{-1,}$ se realizuje pomoću PIPO registra. Podsetimo se sada realizacije ova tri gradivna elementa u VHDL-u.

Sabirač:

library IEEE;

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity sabirac is
   Port (a : in STD_LOGIC_VECTOR (47 downto 0);
         b : in STD_LOGIC_VECTOR (47 downto 0);
         c : out STD_LOGIC_VECTOR (48 downto 0));
end sabirac;

architecture Behavioral of sabirac is
   signal a_ext, b_ext: signed(48 downto 0);
begin
   a_ext <= resize(signed(a),49);
   b_ext <= resize(signed(b),49);
   c <= std_logic_vector(a_ext + b_ext);
end Behavioral;
```

Prethodni VHDL kod će uspešno proći sintezu na većini alata za ovu namenu (u našem slučaju *Vivado* kompanije *Xilinx*). To je znak da alat „zna" kako da realizuje operator + pomoću odgovarajuće kombinacione logike. Kao što ste ćete se upoznati na predavanjima, postoji više tipova sabirača. Opravdano se možemo pitati koji od ovih tipova sintetiše naš alat i da li mi imamo kontrolu nad time. Da bismo očuvali jednostavnost i čitljivost koda mi ćemo pustiti alat da odabere sabirač. Ipak, ukoliko to aplikacija traži od nas, moguće je modelovati odgovarajući sabirač i tako ispuniti zahteve. Na primer, ukoliko nam treba kompaktan sabirač, a brzina nam nije od velikog interesa, verovatno ćemo koristiti *ripple-carry* sabirač, ali ako nam treba brži sabirač verovatno će odluka biti da koristimo *carry-lookahead* ili neki drugi. Ista pravila važe i za množače. U nastavku je prikazan VHDL kod koji predstavlja množač, a potom i PIPO registar.

Množač:

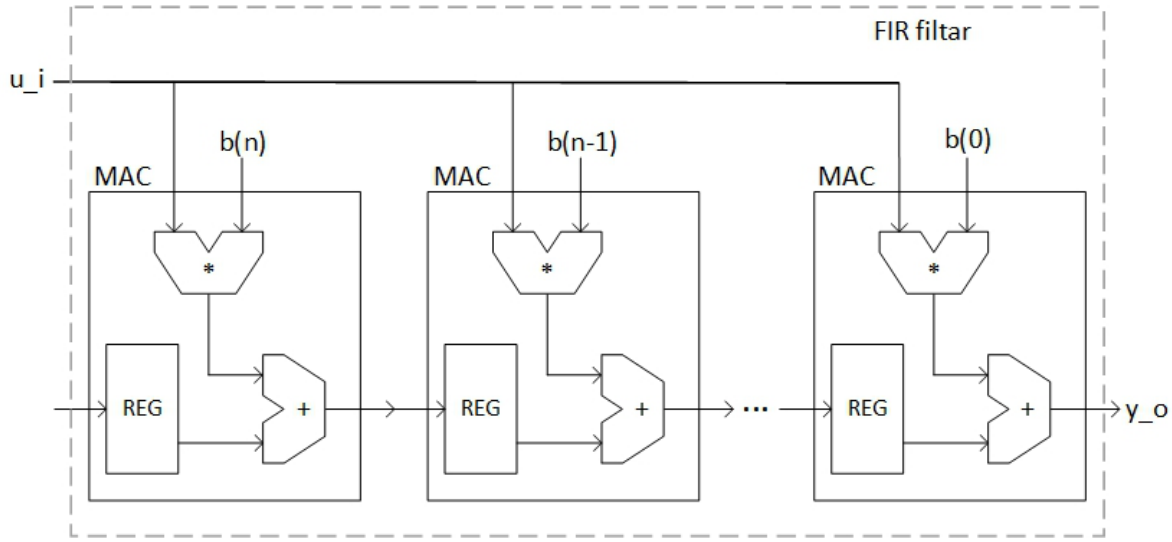
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity mnozac is
   Port (a : in STD_LOGIC_VECTOR (23 downto 0);
         b : in STD_LOGIC_VECTOR (23 downto 0);
         c : out STD_LOGIC_VECTOR (47 downto 0));
end mnozac;

architecture Behavioral of mnozac is
begin
   c <= std_logic_vector(signed(a) * signed(b));
end Behavioral;
```

PIPO registar:

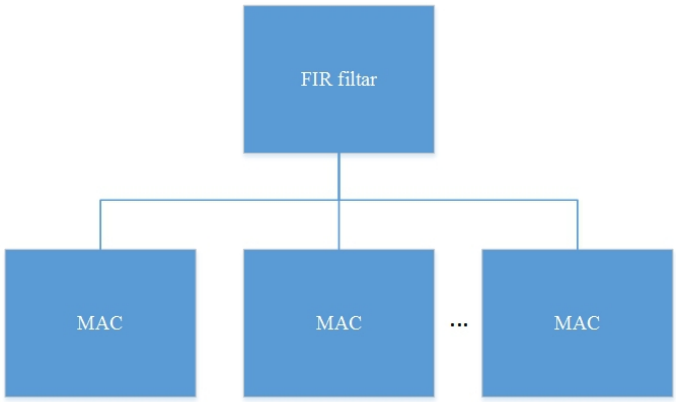
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

```
entity PIPO is
   Port (clk_i : in STD_LOGIC;
         d : in STD_LOGIC_VECTOR (47 downto 0);
         q : out STD_LOGIC_VECTOR (47 downto 0));
end PIPO;

architecture Behavioral of PIPO is
begin
   process(clk_i)
   begin
      if clk_i'event and clk_i = '1' then
         q <= d;
      end if;
   end process;
end Behavioral;
```

Primetimo da je rezultat sabiranja dva broja širine *n* bita širok *n+1* bita dok je rezultat množenja dva broja širine *n* bita širok *2 x n* bita. Pošto smo se podsetili implementacije gradivnih elemenata FIR sistema u VHDL-u, vratićemo se na blok šemu sa slike 2 i prevešćemo je u šemu koju ćemo realizovati u VHDL-u. Na slici 3 je prikazana šema koja će biti dalje korišćena, koja ujedno predstavlja i modularizaciju FIR filtra dok je na slici 4 prikazana hijerarhijska struktura.



Slika 3: Modularizacija FIR filtra.



Slika 4: Hijerarhijska struktura FIR filtra.

Kao što se može videti na slici 3 modularizaciju smo izvršili tako što smo prepoznali gradivne blokove transponovane direktne forme koji se ponavljaju te ih enkapsulirali u okviru podmodula koji ćemo dalje zvati *MAC* (Multiply Accumulate). Ova struktura predstavlja osnovu mnogih DSP algoritama te je to još jedan razlog zašto je ovakva hijerarhijska podela napravljena. U okviru *MAC* modula se nalaze množač, sabirač i registar, to jest, tri osnovna elementa LVN DS-ova. Uloga registra je identična ulozi pomerača u vremenu koji smo koristili u okviru prethodnih vežbi u *SIMULINK*-u.

Na osnovu slike 3 možemo reći da kompletan filtar čini nekoliko *MAC* modula sa registrima povezanih na odgovarajući način u okviru „*top level*" fajla. Takođe, sa slike 3 se vrlo lako uočava funkcija *MAC* modula te je ovde nećemo detaljnije analizirati.

Verifikacioni plan

U slučaju ovako koncipiranog FIR filtra verifikacioni plan je vrlo jednostavan. Formiraćemo jedan testbenč pomoću kojeg ćemo vizualno utvrditi korektnost rada FIR filtra. Da bismo to uradili potrebno je napisati kraći *MATLAB* program u kome ćemo generisati koeficijente RUI-a (b_n) i formirati ulazne test vektore. Kada formiramo koeficijente i test vektore, učitaćemo ih u okviru testbenča i jedan po jedan test vektor dovesti na ulaz FIR filtra.

VHDL model *MAC* modula

Interfejs *MAC* modula čine sinhronizacioni *clk_i* signal, ulazni podatak (odbirak) *u_i*, ulazni port koeficijenta *b_i*, *mac_i* ulazni port prethodnog modula i *mac_o* izlazni port podataka. Kao što je poznato množenje dva broja širine *n* bita daje rezultat širine *2 x n* stoga je međurezultat mnoćenja *u_i* i *b_i* širine 48 bita. Takođe, poznato je da sabiranje dva broja širine *n* bita daje rezultat širine *n+1* bit. Vodeći se ovim pravilom trebali bismo rezultat sabiranja registra i množenja da povećamo za 1 bit sa 48 na 49 bita. Ipak ovo nećemo raditi jer su prekoraćenja u ovom smislu kod FIR filtara retka te ćemo sačuvati jednostavnost modula. Na osnovu analize brojeva sa pokretnim zarezom (sa prethodnih vežbi) možemo zaključiti da je operacije sabiranja i množenja moguće izvršiti na identičan način kao i u slučaju celih brojeva.

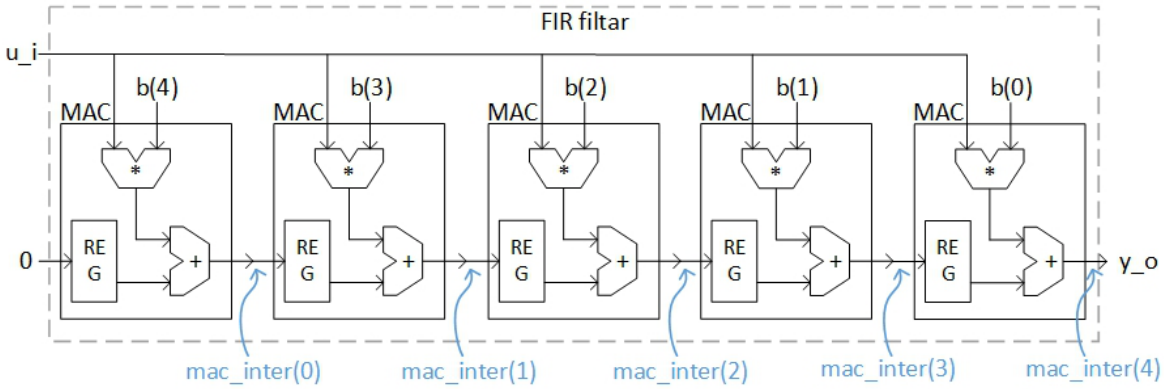
U nastavku je prikazan VHDL model *MAC* modula.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity mac is
   Port ( clk_i : in STD_LOGIC;
          u_i : in STD_LOGIC_VECTOR (23 downto 0);
          b_i : in STD_LOGIC_VECTOR (23 downto 0);
          mac_i : in STD_LOGIC_VECTOR (47 downto 0);
          mac_o : out STD_LOGIC_VECTOR (47 downto 0));
end mac;

architecture Behavioral of mac is
   signal reg_s : STD_LOGIC_VECTOR(47 downto 0) := (others=>'0');
begin
   process(clk_i)
   begin
```

```
    if (clk_i'event and clk_i = '1') then    --proces koji modeluje PIPO registar
       reg_s <= mac_i;
    end if;
  end process;
  --kombinaciona logika koja implementira funkcionalnost mnozaca i sabiraca
  mac_o <= std_logic_vector(signed(reg_s) + (signed(u_i) * signed(b_i)));

end Behavioral;
```

VHDL model FIR filtra

Da bismo formirali FIR prema željenoj specifikaciji u okviru „*top level*" fajla ćemo instancirati 5 *MAC* modula i povezati ih kao na slici 5.



Slika 5: Detaljna blok šema projektovanog FIR filtra.

Na ulazu *mac_i* prvog modula je dovedena 0. Prilikom sinteze, alat će primetiti da je vrednost registra u okviru prvog *MAC* modula konstantno 0 te će izbaciti ovaj registar i na izlaz *mac_o* će proslediti rezultat množenja *b(0)* i *u_i*. U okviru ovog fajla je formiran i niz koeficijenata veličine 5 u kome su sačuvani podaci u vezi sa koeficijentima. Postupak kojim smo došli do ovih koeficijenata će biti prikazan u okviru verifikovanja ispravnosti rada filtra.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity fixed_fir is
   Port ( clk_i : in STD_LOGIC;
          u_i : in STD_LOGIC_VECTOR (23 downto 0);
          y_o : out STD_LOGIC_VECTOR (23 downto 0));
end fixed_fir;

architecture Behavioral of fixed_fir is
   type std_2d is array (4 downto 0) of std_logic_vector(47 downto 0);
   signal mac_inter : std_2d := (others=>(others=>'0'));

   type coef_t is array (0 to 4) of std_logic_vector(23 downto 0);
   signal b : coef_t := (x"18bfcb",
                         x"1a05d9",
                         x"1a74b5",
                         x"1a05d9",
```

```
                        x"18bfcb");
begin

    prvi_MAC:
    entity work.mac(behavioral)
    port map(clk_i=>clk_i,
                u_i=>u_i,
                b_i=>b(4),
                mac_i=>(others=>'0'),
                mac_o=>mac_inter(0));

    drugi_MAC:
    entity work.mac(behavioral)
    port map(clk_i=>clk_i,
                u_i=>u_i,
                b_i=>b(3),
                mac_i=>mac_inter(0),
                mac_o=>mac_inter(1));

    treci_MAC:
    entity work.mac(behavioral)
    port map(clk_i=>clk_i,
                u_i=>u_i,
                b_i=>b(2),
                mac_i=>mac_inter(1),
                mac_o=>mac_inter(2));

    cetvrti_MAC:
    entity work.mac(behavioral)
    port map(clk_i=>clk_i,
                u_i=>u_i,
                b_i=>b(1),
                mac_i=>mac_inter(2),
                mac_o=>mac_inter(3));

    peti_MAC:
    entity work.mac(behavioral)
    port map(clk_i=>clk_i,
                u_i=>u_i,
                b_i=>b(0),
                mac_i=>mac_inter(3),
                mac_o=>mac_inter(4));

    y_o <= mac_inter(4)(46 downto 23);

end Behavioral;
```

NAPOMENA: Izlaz *MAC* modula ne možemo direktno voditi na izlaz filtra jer je po dizajn specifikaciji rečeno da izlaz treba da bude 24-bitni. Kao što se vidi u poslednjoj liniji VHDL fajla skraćivanje je urađeno odsecanjem. Množenjem dva 24-bitna broja dobili smo 48-bitni. Oba broja su imala po jedan bit levo od decimalne tačke i još 23 desno. Množenjem dobijamo 2 *x* 1 bit sa leve strane (od svakog člana proizvoda po jedan) i 2 *x* 23 sa desne

strane koji predstavljaju razlomljeni deo broja. Ukoliko želimo da dobijemo izlazni podatak u formatu 1.23 potrebno je uzeti bite izlaznog vektora od 46 do 23 (1 celobrojni bit i 23 MSB bita razlomljenog dela).

Verifikaciono okruženje

Ispravnost projektovanog filtra ćemo proveriti tako što ćemo prvo formirati ekvivalentan filter u okviru *MATLAB*-a, a potom ćemo ga uporediti sa VHDL implementacijom. U okviru projektovanja fiksnog FIR filtra izvršićemo isključivo vizualnu proveru dok ćemo za parametrizovani filtar formirati verifikaciono okruženje koje će porediti rezultate sa rezultatima *MATLAB*-ove funkcije *filter*. Sa radom u *MATLAB*-u smo se do sada već dobro upoznali te nećemo detaljnije analizirati postupak formiranja koeficijenata filtra. Test filtar je NF tipa koji ne propušta frekvencije iznad $w = 0.1$. Test signal je oblika:

$$u = 0.85 \cdot \cos\left(2 \cdot \pi \cdot \frac{400\,Hz}{22050\text{Hz}} \cdot n\right) + 0.2 \cdot \cos\left(2 \cdot \pi \cdot \frac{4000\,Hz}{22050\text{Hz}} \cdot n\right).$$

Pošto je u pitanju NF filtar očekujemo da komponenta na 4000Hz bude potisnuta, to jest, da FIR propusti isključivo 400Hz. Primetimo i da je prekvencija odabiranja 22050Hz što znači da se propusni opseg završava negde oko 1100Hz. Dužina test signala je 150 odbiraka.

U nastavku je program koji formira koeficijente željenog filtra, test signal i 2 tekstualna fajla u kojima će biti sačuvani koeficijenti odnosno test vektori.

```
%broj bita odbirka (format je 1.23)
word_length = 24;
fraction_length = 23;
fs = 22050;
f1 = 400;
f2 = 4000;

%specifikacija NF filtra
fir_ord = 4;
Wn=[0.1];
%odbirci prozorske funkcije koja se koristi
pravougaoni = rectwin(fir_ord+1);
%projektovanje FIR filtara koriscenjem funkcije fir1
b = fir1 (fir_ord, Wn, pravougaoni);
a = 1;
%diskretno vreme
n = 0:149;
%definisanje ulaznog signala u trajanju od 150 odbiraka
u = 0.85*cos(2*pi*f1/fs*n) + 0.2*cos(2*pi*f2/fs*n);
%filtriranje signala pomocu formiranog filtra
y_real = filter(b,a,u);
%crtanje ulaznog i izlaznog signala
set(gcf, 'color', 'w');
subplot(2,1,1), stem(n,u), title('Ulazni signal u trajanju od 150 odbiraka');
subplot(2,1,2), stem(n,y_real), title('Izlazni signal u trajanju od 150 odbiraka racunat pomocu funkcije filter');

struct.mode = 'fixed';
strct.roundmode = 'floor';
struct.overflowmode = 'saturate';
struct.format = [word_length fraction_length];
q = quantizer(struct);
```

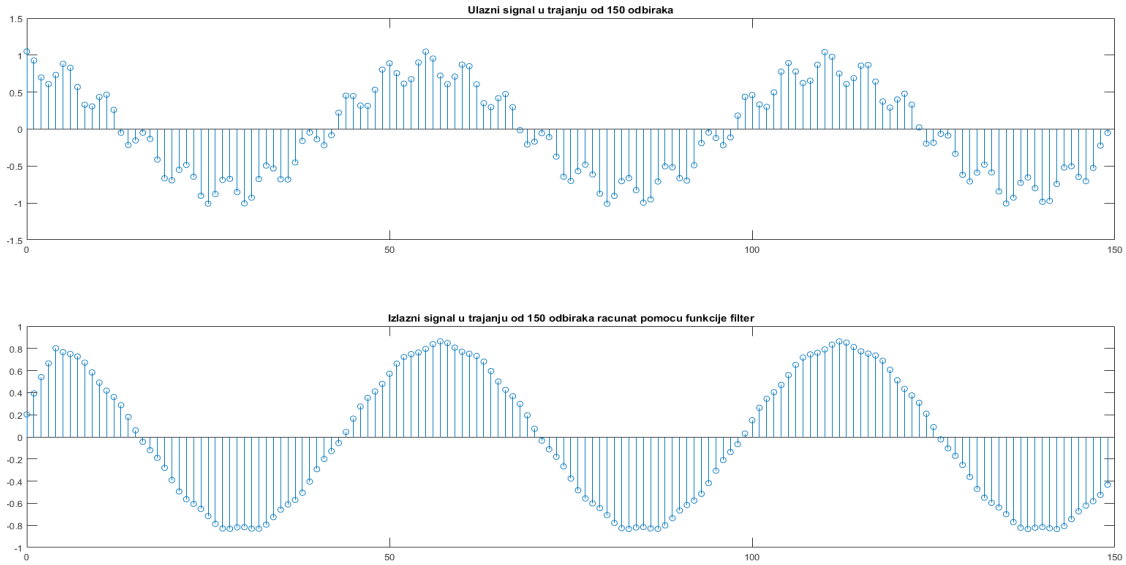
```
%digitalizacija diskretnog signala
u_digital = quantize(q,u);

%koeficijenti filtra
fileIDh = fopen('coef_hex.txt','w');
for i=1:fir_ord+1
   fprintf(fileIDh,'x"');
   fprintf(fileIDh,num2hex(q,b(i)));
   fprintf(fileIDh,'",\n');
end
fclose(fileIDh);

fileIDb = fopen('input.txt','w');
for i=1:length(u_digital)
   fprintf(fileIDb,num2bin(q,u(i)));
   fprintf(fileIDb,'\n');
end
fclose(fileIDb);
```

Za vizualnu proveru bi bilo dobro da u okviru *MATLAB* programa grafički prikažemo ulazni i izlazni signal FIR filtra. To je i urađeno, a na slici 6 možemo videti odziv na pobudu *u*. Na slici 6 se jasno vidi da je komponenta ulaznog signala na 4000Hz potisnuta projektovanim FIR filtrom. Da bismo prikazali kompletan proces projektovanja diskretnih sistema, posle diskretizacije sproveli smo i postupak digitalizacije signala. Digitalizovan signal smo propustili kroz *filter* funkciju i prikazali ovako dobijeni rezultat. Ovaj rezultat, ustvari, predstavlja očekivani rad hardverski realizovanog DS-a. Na kraju je prikazana greška prouzrokovana digitalizacijom signala. Ovime je završeno projektovanje željenog filtra u *MATLAB*-u i u nastavku prelazimo na verifikaciju modelovanog digitalnog filtra u VHDL-u.

Novina u prethodnom *MATLAB* programu u odnosu na dosadašnje vežbe jeste upis koeficijenata filtra i test vektora u tekstualni fajl. Upis u 2 tekstualna fajla smo uradili pomoću poslednje dve *for* petlje. Pre početka upisa je potrebno otvoriti željeni fajl u *write* modu. To smo uradili pozivom funkcije *fopen* sa parametrima: *ime_fajla.txt* i modom otvaranja *'w'* koji označava upis. Ukoliko fajl ne postoji *MATLAB* će ga kreirati. Za upis u otvoreni fajl ćemo koristiti funkciju *fprintf* kojoj prosleđujemo informaciju u koji fajl da upiše, *fileID,* i podatak koji treba da upiše. U slučaju upisa koeficijenata iskoristili smo funkciju *num2hex*. Ova funkcija ima mogućnost da radi kvantizaciju (parametar *q*) i pretvaranje u heksadecimalni oblik. Isti princip rada ima i funkcija *num2bin* koju smo iskoristili za formiranje ulaznih odbiraka. U nastavku su prikazani formirani fajlovi, prvo *coef_hex.txt*, a potom i deo *input.txt* fajla.

```
x"18bfcb",
x"1a05d9",
x"1a74b5",
x"1a05d9",
x"18bfcb",
```

```
011111111111111111111111
011111111111111111111111
011011000000011001001001
010111111111100010011100
011011110000010100010011
011111111111111111111111
011110001111011011101100
...
```



Slika 6: Odziv FIR LVN DS sistema dobijenog korišćenjem funkcije filter.

Sledi testbenč korišćen prilikom provere korektnosti rada modelovanog filtra.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use std.textio.all;
use work.txt_util.all;

entity fix_fir_tb is
--  Port ( );
end fix_fir_tb;

architecture Behavioral of fix_fir_tb is
    signal clk_s : std_logic;
    signal uut_input_s : std_logic_vector(23 downto 0);
    signal uut_output_s : std_logic_vector(23 downto 0);

    constant per_c : time := 20ns;
    --putanju do željenog fajla je potrebno prilagoditi strukturi
    --direktorijuma na računaru na kome se vrši provera rada.
    file input_test_vector : text open read_mode is
          "D:\predavanja\DS\fir_fixed\matlab\input.txt";

begin
    fir_under_test:
    entity work.fixed_fir(behavioral)
    port map(clk_i=>clk_s,
                u_i=>uut_input_s,
                y_o=>uut_output_s);

    clk_process:
    process
    begin
       clk_s <= '0';
       wait for per_c/2;
```

```
    clk_s <= '1';
    wait for per_c/2;
  end process;

  stim_process:
  process
    variable tv : line;
  begin
    uut_input_s <= (others=>'0');
    wait until falling_edge(clk_s);
    --ulaz za filtriranje
    while not endfile(input_test_vector) loop
      readline(input_test_vector,tv);
      uut_input_s <= to_std_logic_vector(string(tv));
      wait until falling_edge(clk_s);
    end loop;
    report "verification done!" severity failure;
  end process;

end Behavioral;
```

Kao što možemo videti testbenč je vrlo jednostavan. Nešto složeniji deo okruženja predstavlja *stim_process*. Pre nego se upustimo u analizu ovog procesa pomenimo da testbenč ima još i proces koji generiše takt signal periode *per_c* nazvan *clk_process*. Pored takt procesa u okviru arhitekture su definisani i signali kojima testbenč interaguje sa modelovanim filtrom (*clk_s, uut_input_s, uut_output_s*). Da bismo omogućili učitavanje test vektora morali smo otvoriti tekstualni fajl u *read_mode*-u sledećom linijom koda:

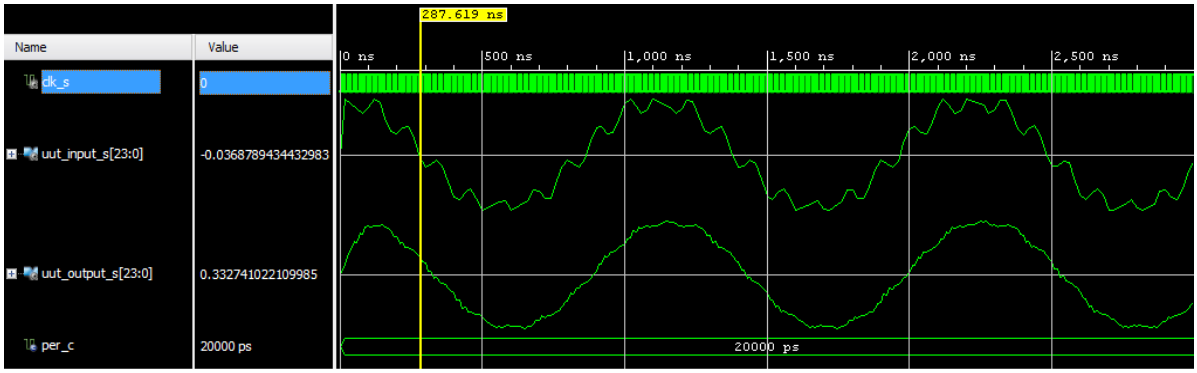
```
file input_test_vector : text open read_mode is
        "D:\predavanja\DS\fir_fixed\matlab\input.txt";
```

Otvorenom fajlu ćemo u okviru testbenča pristupati preko imena *input_test_vector*. Na početku simulacije pored *clk_process*-a počinje da se izvršava i *stim_process*. Na početku *stim_process*-a dodeljujemo početnu vrednost ulaznom signalu filtra (*uut_input_s <=(others=>'0');*) i čekamo opadajuću ivicu takt signala. Potom ulazimo u *while* petlju u kojoj ostajemo dok ne pročitamo sve test vektore. Za ostanak u petlji smo iskoristili funkciju *endfile* kojoj je dovoljno da prosledimo ime fajla, a kao povratnu vrednost ćemo dobiti *true* u slučaju da smo došli do kraja fajla. Po ulasku u petlju pozivamo funkciju *readline* kojoj prosleđujemo ime fajla i promenljivu tipa *line* (u našem slučaju *tv*) u koju će nam pozvana funkcija sačuvati sledeću liniju tekstualnog fajla. Pozivom *readline* funkcije simulator automatski prelazi u sledeći red tako da će prilikom sledećeg poziva pročitati sledeću liniju bez potrebe za našom intervencijom. U nastavku je potrebno promenljivu tipa *line* kastovati, prvo u *string*, a zatim i u *std_logic_vector*. Tako formiran vektor se dovodi na ulaz filtra. Ovaj postupak se ponavlja na svaku opadajuću ivicu takt signala (kako bi se izbegli mogući problemi trke). Po apliciranju svih test vektora verifikacija se prekida sa linijom:

```
        report "verification done!" severity failure;
```

NAPOMENA: Da bismo radili sa funkcijama kao što su readline, otvaranje fajla i ostale potrebno je uključiti *std.textio* paket. Za funkciju *to_std_logic_vector* koja konvertuje *string* u *std_logic_vector* potrebno je uključiti i nestandardni paket *txt_util* koji se može pronaći u okviru dizajn fajlova na sajtu predmeta.

Prilikom simulacije smo dobili rezultate prikazane na slici 7. Kao što možemo videti ulazni signal *uut_input_s* ima dve komponente: „sporiju" sa većom amplitudom i „bržu" sa manjom amplitudom. Izlazni signal *uut_output_s* čini isključivo „spora" komponenta ulaznog signala kako je i očekivano. **Posle ove grube analize možemo reći da projektovani filtar radi korektno. Ipak, za kvalitetno verifikovanje vizuanlna provera nije dovoljna. Ovaj nedostatak ćemo upotpuniti u nastavku kada budemo verifikovali rad parametrizovanog filtra.**



Slika 7: Odziv modelovanog filtra pomoću VHDL-a.

NAPOMENA: Da bi se dobio ovakav prikaz u simulatoru u okviru *Vivado* alata potrebno je kliknuti desnim na željeni signal, odabrati ***Waveform Style*** i opciju ***Analog***. Potom, isto desnim klikom na signal se otvara padajući meni, ali sada biramo ***Radix*** i opciju ***Real Setting***. Kada se otvori ovaj prozor potrebno je odabrati *Fixed Point* i postaviti decimalnu tačku na poziciju 23.

Ovime smo završili modelovanje FIR filtra fiksne arhitekture. U nastavku ćemo sličan postupak primeniti na projektovanje parametrizovanog FIR filtra.

Zadaci za vežbu:

Zadatak 1:

Uneti neophodne izmene u okviru *MATLAB* programa tako da se projektuje VF filtar četvrtog reda sa graničnom učestanošću *w=0.3*. Potom je potrebno zadati odgovarajući test signal kako bi se efekti filtriranja jasno videli. Na kraju, napravite odgovarajuće izmene u okviru VHDL fajla projektovanog filtra i utvrdite ispravnost rada VF filtra na simulatoru simulacije VHDL modela.

Zadatak 2: Unesite neophodne izmene u modelovani filtar (u okviru VHDL fajlova) tako da napravite filtar šestog reda. U skladu sa navedenim prilagodite i *MATLAB* program i izvršite vizualnu proveru rada modelovanog filtra.

Implementacija parametrizovanog FIR filtra

Funkcionalna specifikacija sistema koji je potrebno projektovati

Potrebno je projektovati digitalni FIR filtar koji treba da ima sledeću funkcionalnost:

- Korisnik je u mogućnosti da menja koeficijente preko posebnog porta
- Koristiti transponovanu direktnu formu
- Parametrizovati širinu ulaznog i izlaznog signala
- Parametrizovati red filtra

Dizajn specifikacija sistema

Parametrizovani FIR filtar ima sledeći interfejs:

- *clk_i* – ulazni port preko kojeg se u sistem dovodi globalni sinhronizacioni signal.
- *data_i* – ulazni port za podatke (odbirke).
- *data_o* – izlazni port za podatke (filtrirani podaci).
- *coef_addr_i* – ulazni port adrese koeficijenta.
- *coef_i* – ulazni port koji predstavlja vrednost koeficijenta.
- *we_i* – signal dozvole upisa odgovarajućeg koeficijenta.

Pored portova FIR filtar ima i 3 parametra:

- *fir_ord* – parametar koji predstavlja red filtra.
- *input_data_width* – parametar koji predstavlja širinu ulaznih podataka.
- *output_data_width* – parametar koji predstavlja širinu izlaznih podataka.

Interfejs parametrizovanog FIR filtra je prikazani na slici 8.

Slika 8: Interfejs FIR filtra.

Dizajn specifikacija je u velikoj meri identična specifikaciji fiksnog FIR filtra. Prevashodno mislimo gradivne elemente (*MAC*-ove) i povezivanje. Bitna razlika se ogleda u tome što je filtar ovoga puta parametrizovan i što je potrebno dodati memoriju u kojoj ćemo čuvati koeficijente. U nastavku ćemo komentarisati ove izmene uz priložene VHDL kodove.

Verifikacioni plan

Verifikaciono okruženje će ovoga puta biti nešto složenije. U *MATLAB* programu ćemo učitati *speech_dft.wav* fajl (koji predstavlja kratak govor) i semplove ovog fajla upisati u *input.txt* fajl. Potom ćemo formirati koeficijente filtra (promenljivog reda) i njih, takođe,

upisati u fajl *coef.txt*. Na kraju ćemo učitani govor propustiti kroz *MATLAB*-ovu funkciju *filter* i dobijene rezultate (očekivane odbirke) upisati u fajl *expected.txt*. Po završetku generisanja svih fajlova napisaćemo VHDL testbenč koji će na ulaz FIR-a dovoditi neobrađeni govor dok će izlaz porediti sa govorom filtriranim pomoću *MATLAB* programa.

VHDL model parametrizovanog *MAC* modula

Kao što možemo primetiti u kodu koji sledi *MAC* modul parametrizovanog FIR filtra se ne razlikuje puno u odnosu na *MAC* modul filtra fiksne arhitekture. Interfejs je identičan sa tom razlikom što postoji dodatni parametar *input_data_width* koji predstavlja parametar širine ulaznih i izlaznih portova. Pomoću ovog parametra određujemo širinu *u_i* i *b_i* portova dok *sec_i* i *sec_o* imaju širinu jednaku *2 x input_data_width* jer predstavljaju rezultat množenja i sabiranja.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use IEEE.NUMERIC_STD.ALL;

entity mac is
   generic (input_data_width : natural :=24);
   Port ( clk_i : in std_logic;
          u_i : in STD_LOGIC_VECTOR (input_data_width-1 downto 0);
          b_i : in STD_LOGIC_VECTOR (input_data_width-1 downto 0);
          sec_i : in STD_LOGIC_VECTOR (2*input_data_width-1 downto 0);
          sec_o : out STD_LOGIC_VECTOR (2*input_data_width-1 downto 0));
end mac;

architecture Behavioral of mac is
   signal reg_s : STD_LOGIC_VECTOR (2*input_data_width-1 downto 0):=(others=>'0');
begin
   process(clk_i)
   begin
      if (clk_i'event and clk_i = '1')then
         reg_s <= sec_i;
      end if;
   end process;

   sec_o <= std_logic_vector(signed(reg_s) + (signed(u_i) * signed(b_i)));

end Behavioral;
```

VHDL model parametrizovanog FIR filtra

Najveći deo parametrizacije širine podataka se krije u *MAC* modulu. U okviru „*top level*" fajla je potrebno parametrizovati portove i širine signala koji povezuju *MAC* module. Ovaj fajl ima tri parametra i to *fir_ord* (red filtra), *input_data_width* (širina ulaznih podataka) i *output_data_width* (širina izlaznih podataka). Parametrizacija širine portova je slična parametrizaciji sa kojom smo se do sada sretali osim u slučaju *coef_addr_i* signala. Ovaj signal treba da bude dovoljne širine da može da adresira *fir_ord* memorijskih lokacija u kojima čuvamo vrednosti koeficijenata filtra. U kodu imamo primer u kome je red filtra 20. Ako izračunamo logaritam sa osnovom 2 od reda filtra + 1 $\left(\log_2\left(fir_{ord}+1\right)\right)$ dobićemo

potrebnu širinu adresne magistrale. Dodatak (+1) na red filtra je zbog činjenice da filtar, na primer, dvadesetog reda ima 21 koeficijent. Funkcija log2c je implementirana u okviru paketa *util_pkg*. U nastavku je prikazan VHDL kod parametrizovanog FIR filtra, a potom sledi analiza istog.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.util_pkg.all;

entity fir_param is
   generic(fir_ord : natural := 20;
             input_data_width : natural := 24;
             output_data_width : natural := 24);
   Port (clk_i : in STD_LOGIC;
         we_i : in STD_LOGIC;
         coef_addr_i : std_logic_vector(log2c(fir_ord+1)-1 downto 0);
         coef_i : in STD_LOGIC_VECTOR (input_data_width-1 downto 0);
         data_i : in STD_LOGIC_VECTOR (input_data_width-1 downto 0);
         data_o : out STD_LOGIC_VECTOR (output_data_width-1 downto 0));
end fir_param;

architecture Behavioral of fir_param is
   type std_2d is array (fir_ord downto 0) of
        std_logic_vector(2*input_data_width-1 downto 0);
   signal mac_inter : std_2d:=(others=>(others=>'0'));

   type coef_t is array (fir_ord downto 0) of
        std_logic_vector(input_data_width-1 downto 0);
   signal b_s : coef_t := (others=>(others=>'0'));

begin
   --proces koji modeluje sinkroni upis u memoriju b_s
   process(clk_i)
   begin
      if(clk_i'event and clk_i = '1')then
         if we_i = '1' then
            b_s(to_integer(unsigned(coef_addr_i))) <= coef_i;
         end if;
      end if;
   end process;
   --instanca prvog MAC-a čiji je ulaz sec_i jednak 0
   first_section:
   entity work.mac(behavioral)
   generic map(input_data_width=>input_data_width)
   port map(clk_i=>clk_i,
               u_i=>data_i,
               b_i=>b_s(fir_ord),
               sec_i=>(others=>'0'),
               sec_o=>mac_inter(0));
   --instanciranje ostalih MAC modula filtra
   other_sections:
```

```
   for i in 1 to fir_ord generate
      fir_section:
      entity work.mac(behavioral)
      generic map(input_data_width=>input_data_width)
      port map(clk_i=>clk_i,
                  u_i=>data_i,
                  b_i=>b_s(fir_ord-i),
                  sec_i=>mac_inter(i-1),    --sec_o signal prethodnog MAC modula
                  sec_o=>mac_inter(i));
   end generate;
   --registrovanje izlaznog signala
   process(clk_i)
   begin
      if(clk_i'event and clk_i='1')then
         data_o <= mac_inter(fir_ord)
                  (2*input_data_width-2 downto 2*input_data_width-output_data_width-1);
      end if;
   end process;
end Behavioral;
```

Na samom početku je potrebno da definišemo dvodimenzionalni tip (*coef_t*) za signal koji će predstavljati memoriju u kojoj ćemo čuvati koeficijente filtra. Ovaj tip ima *fir_ord+1* lokacija širine *input_data_width* bita. Potom smo deklarisali signal *b_s* koji je *coef_t* tipa i inicijalizovali sve lokacije na 0. Pored ovog tipa/signala definisali smo, takođe, dvodimenzionalni tip *std_2d* i deklarisali signal *mac_inter* koji će imati ulogu povezivanja izlaza jednog *MAC* modula sa ulazom sledećeg *MAC*-a. Uloga ovog signala je identična ulozi *mac_inter0-4* signala fiksne arhitekture filtra. Posle deklaracije signala sledi proces koji modeluje sinhroni upis u memoriju za čuvanje koeficijenata. Ukoliko je signal *we_i* aktivan, podatak sa porta *coef_i* će biti upisan na lokaciju *coef_addr_i* u memoriju *b_s*.

U nastavku koda možemo videti instancu prvog *MAC* modula. Povezivanje portova *clk_i* i *u_i* je logično dok u slučaju *b_i*, *sec_i* i *sec_o* treba voditi računa. Pošto je ovo prvi *MAC* u filtru (nema prethodnog) na port *sec_i* smo doveli signal vrednosti 0. Na svim ostalim *MAC*-ovima *sec_i* ulaz je povezan sa *sec_o* izlazom prethodnog modula i to je razlog zbog kog smo instanciranje *MAC*-ova razdvojili na dva dela, prvi *MAC* i svi ostali. Na *b_i* smo doveli odgovarajući (prvi) koeficijent dok je izlaz modula povezan na *mac_inter(0)*. Ovaj među signal će u nastavku biti povezan na ulaz sledećeg *MAC*-a u okviru *for generate* petlje. U ovoj petlji smo instancirali ostale *MAC* module i izvršili njihovo povezivanje. Povezivanje *clk_i, data_i* i *b_s(i)* je vrlo intuitivno dok smo *sec_i* i *sec_o* povezali pomoću *mac_inter* signala kao što je to opisano (izlaz prethodnog predstavlja ulaz trenutnog).

Na kraju izlaz filtra predstavlja registrovani signal *mac_inter(fir_ord)*. Registrovanje izlaznih portova je dobra praksa, ali nije obavezno i zavisi od specifikacije modula. Dodatno je moguće registrovati i ulazne portove (osim naravno *clk_i*), međutim, to ovde nije urađeno. Kao i u slučaju filtra sa fiksnom arhitekturom i ovde je skraćivanje izlaznog signala urađeno odsecanjem zbog jednostavnosti.

Verifikaciono okruženje

Kao i u slučaju fiksne arhitekture, filtar ćemo testirati pomoću ulaznih podataka generisanih u *MATLAB* programu. Za testiranje smo odabrali konfiguraciju filtra 20. reda i na isti način kao u slučaju fiksne arhitekture generisali koeficijente. Potom smo pomoću funkcije

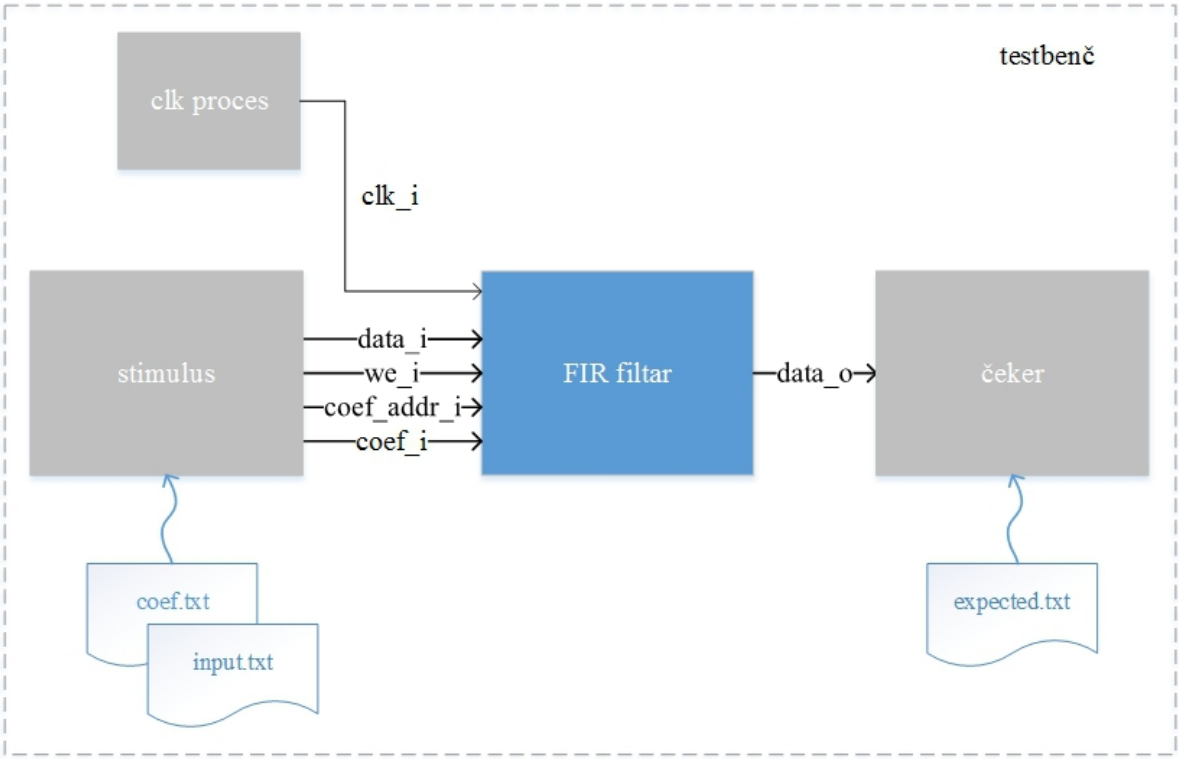
audioread učitali *speech_dft.wav* fajl. Povratne vrednosti ove funkcije su smeštene u promenljivu *u* (odbirci .wav fajla) i *Fs* (frekvenciju semplovanja) koja nam nije od značaja, ali iznosi 22050Hz. Vrednosti odbiraka *u* smo sačuvali u fajl *input.txt*, a koeficijente *b* u *fajl coef.txt*. Posle učitavanja izvorni fajl je filtriran funkcijom *filter*, a dobijeni rezultati su sačuvani u fajlu *expected.txt*. Ovi rezultati će nam poslužiti za proveru da li implementirani filtar radi korektno. Primetimo da je za proveru rada odabran NF filtra. U nastavku sledi MATLAB program koji vrši navedenu funkciju.

```
%broj bita odbirka (format je 1.23)
word_length = 24;
fraction_length = 23;

%specifikacija NF filtra
fir_ord = 20;
Wn=[0.1];
%odbirci prozorske funkcije koja se koristi
pravougaoni = rectwin(fir_ord+1);
%projektovanje FIR filtara koriscenjem funkcije fir1
b = fir1 (fir_ord, Wn, pravougaoni);
a = 1;
%ucitavanje speech_dft.wav fajla
[u,Fs] = audioread('speech_dft.wav');
%filtriranje zvuka pomocu formiranog filtra
y = filter(b,a,u);

struct.mode = 'fixed';
strct.roundmode = 'floor';
struct.overflowmode = 'saturate';
struct.format = [word_length fraction_length];
q = quantizer(struct);

%cuvanje koeficijenata filtra u fajl coef.txt
fileIDb = fopen('coef.txt','w');
for i=1:fir_ord+1
   fprintf(fileIDb,num2bin(q,b(i)));
   fprintf(fileIDb,'\n');
end
fclose(fileIDb);
%cuvanje test vektora filtra u fajl input.txt
fileIDb = fopen('input.txt','w');
for i=1:length(y)
   fprintf(fileIDb,num2bin(q,u(i)));
   fprintf(fileIDb,'\n');
end
fclose(fileIDb);
%cuvanje ocekivanih vrednosti dobijenih pomocu funkcije filter u fajl expected.txt
fileIDb = fopen('expected.txt','w');
for i=1:length(y)
   fprintf(fileIDb,num2bin(q,y(i)));
   fprintf(fileIDb,'\n');
end
fclose(fileIDb);
```

Napomena: Digitalizaciju signala nije potrebno raditi jer je učitani .wav fajl već digitalizovan.

Pošto smo formirali fajlove sa test vektorima pristupamo realizaciji testbenča u VHDL-u.

Izgled verifikacionog okruženja možemo videti na slici 9.



Slika 9: Verifikaciono okruženje parametrizovanog FIR filtra.

Sa slike 9 se jasno vidi ideja testbenča zasnovanog na zlatnim vektorima. Kreirani fajlovi koeficijenata, ulaznih i izlaznih odbiraka se učitavaju i stimulus počinje da pobuđuje ulaze filtra dok čeker proverava rezultate filtra sa očekivanim vrednostima. Testbenč kao i „*top level*" fajl dizajna ima dva parametra i to red filtra i parametar širine ulaznih i izlaznih podataka. U okviru arhitekture su definisani signali koji će služiti za pobudu filtra, odnosno za nadgledanje izlaznih podataka. Takođe, otvorena su pomenuta tri tekstualna fajla kao i u slučaju testbenča fiksne arhitekture. Obratite pažnju da prilikom pokretanja simulacije na svojim računarima podesite pravilno putanje do ovih fajlova. Signal *start_check* ima ulogu sinhronizacije stimulusa i čekera. Naime, kada stimulus počne da pobuđuje ulaz filtra, postavlja ovaj signal na jedinicu što je znak čekeru da će se na izlazu u narednom taktu pojaviti validni podaci. U vezi sa ovom problematikom će biti više reči u napomeni koja sledi. U nastavku je prikazan VHDL kod korišćenog testbenča.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use IEEE.NUMERIC_STD.ALL;
use std.textio.all;
use work.txt_util.all;
use work.util_pkg.all;

entity tb is
   generic(in_out_data_width : natural := 24;
           fir_ord : natural := 20);
--  Port ( );
end tb;
```

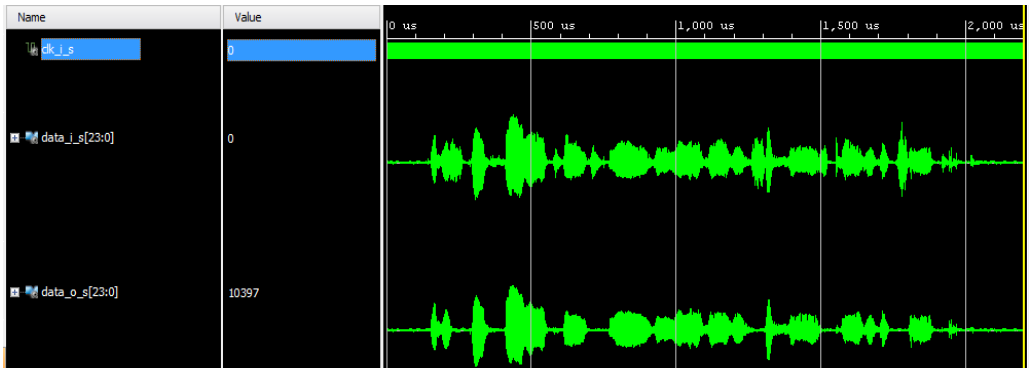
```
architecture Behavioral of tb is
   constant period : time := 20 ns;
   signal clk_i_s : std_logic;
   file input_test_vector : text open read_mode is
        "D:\predavanja\DS\fir_param\input.txt";
   file output_check_vector : text open read_mode is
        "D:\predavanja\DS\fir_param\expected.txt";
   file input_coef : text open read_mode is
        "D:\predavanja\DS\fir_param\coef.txt";
   signal data_i_s : std_logic_vector(in_out_data_width-1 downto 0);
   signal data_o_s : std_logic_vector(in_out_data_width-1 downto 0);
   signal coef_addr_i_s : std_logic_vector(log2c(fir_ord)-1 downto 0);
   signal coef_i_s : std_logic_vector(in_out_data_width-1 downto 0);
   signal we_i_s : std_logic;

   signal start_check : std_logic := '0';

begin
   --instanca filtra koji testiramo
   uut_fir_filter:
   entity work.fir_param(behavioral)
   generic map(fir_ord=>fir_ord,
                  input_data_width=>in_out_data_width,
                  output_data_width=>in_out_data_width)
   port map(clk_i=>clk_i_s,
               we_i=>we_i_s,
               coef_i=>coef_i_s,
               coef_addr_i=>coef_addr_i_s,
               data_i=>data_i_s,
               data_o=>data_o_s);

   clk_process:
   process
   begin
      clk_i_s <= '0';
      wait for period/2;
      clk_i_s <= '1';
      wait for period/2;
   end process;

   stim_process:
   process
      variable tv : line;
   begin
      --upis koeficijenata u memoriju b_s
      data_i_s <= (others=>'0');
      wait until falling_edge(clk_i_s);
      for i in 0 to fir_ord loop
         we_i_s <= '1';
         coef_addr_i_s <= std_logic_vector(to_unsigned(i,log2c(fir_ord)));
         readline(input_coef,tv);
         coef_i_s <= to_std_logic_vector(string(tv));
```

```
            wait until falling_edge(clk_i_s);
        end loop;
        we_i_s <= '0';
        --petlja koja pobuđuje data_i ulaz filtra
        while not endfile(input_test_vector) loop
            readline(input_test_vector,tv);
            data_i_s <= to_std_logic_vector(string(tv));
            wait until falling_edge(clk_i_s);
            start_check <= '1';
        end loop;
        start_check <= '0';
        report "verification done!" severity failure;
    end process;

    check_process:
    process
        variable check_v : line;
        variable tmp : std_logic_vector(in_out_data_width-1 downto 0);
    begin
        wait until start_check = '1';
        while (true) loop
            wait until rising_edge(clk_i_s);
            readline(output_check_vector,check_v);
            tmp := to_std_logic_vector(string(check_v));
            if(abs(signed(tmp) - signed(data_o_s)) > "00000000000000000000111")then
                report "result mismatch!" severity failure;
            end if;
        end loop;
    end process;
end Behavioral;
```

Iz koda se može uočiti da stimulus proces prvo upisuje koeficijente u memoriju filtra, *b_s*, a potom na ulaz *data_i* dovodi u svakom taktu novi test vektor (sempl). Stimulisanje ulaza *data_i* se dešava u *while* petlji dokle god se ne dođe do kraja fajla. Primetimo da se signal *start_check* postavlja na '1' posle apliciranja prvog sempla i ostaje aktivan do kraja verifikacije. Proces koji modeluje čeker čeka da *start_check* signal postane aktivan i potom ulazi u petlju u kojoj u svakom taktu učitava novu očekivanu vrednost pomoću funkcije *readline*. Potom proverava da li se očekivana vrednost (varijabla *tmp*) razlikuje od vrednosti koju na svom izlazu daje modelovani filtar. Ukoliko je ova razlika veća od određene konstante ($>2^{-21}+2^{-22}+2^{-23}$) proces će prijaviti grešku i prekinuti verifikaciju (*severity failure* komanda). Ukoliko se verifikacija uspešno završi dobićemo filtrirani signal oblika kao na slici 10.

NAPOMENA: Ovaj modul predstavlja nešto složeniji sistem u odnosu na module koje smo imali prilike da vidimo na nižim godinama i u skladu sa tim zahteva i ozbiljnije testiranje. Pošto je verifikacija dosta složena oblast i prevazilazi okvire ovoga kursa, ovde smo primenili metodologiju koja je najjednostavnija, testiranje pomoću zlatnih vektora. Na višoj godini postoji kurs (*Funkcionalna verifikacija hardvera*) koji se bavi isključivo verifikacijom digitalnih sistema na kojima će se na sistematičan način pristupati procesu verifikacije. Tu ćete učiti zašto nije uvek najbolje da, na primer, stimulus i čeker komuniciraju (*start_check* signal) i slične stvari. Ipak mi smo se zarad jednostavnosti verifikacionog okruženja poslužili takvim potezima. Napomenimo i to da se rezultati

projektovanog filtra razlikuju jer smo skraćivanje izlaza izvršili odsecanjem dok je u MATLAB programu korišćeno zaokruživanje. Ipak, dobijeni rezultati su dovoljno dobri u našem test slučaju u kome filtriramo govor.



Slika 10: Primer obrađenog govora pomoću projektovanog FIR filtra.

Zadaci:

Zadatak 1:

Izmeniti projektovani filtar tako da se skraćivanje izlaznog podatka radi zaokruživanjem umesto odsecanjem.

Ideja: Sa krajnjim rešenjem sabrati broj koji ima vrednost 1 na mestu najvišeg bita koji odbacujemo.

Zadatak 2:

Učitani govor *speech_dft.wav* je moguće reprodukovati u okviru *MATLAB*-a pomoću funkcije *sound*. Potražite detalje u vezi sa ovom funkcijom i reprodukujte izvorni fajl. Potom, u okviru testbenča otvorite dodatni fajl u *write* modu u koji ćete upisati filtrirane semplove sa izlaza FIR filtra. Za ove potrebe će vam trebati funkcija *writeline*. Potražite opis ove funkcije. Kada u *.txt* fajl upišete dobijene odbirke pomoću filtra, učitajte ih u *MATLAB* programu i reprodukujte filtrirani zvuk. Trebalo bi da čujete zvuk sa potisnutim visokim frekvencijama.

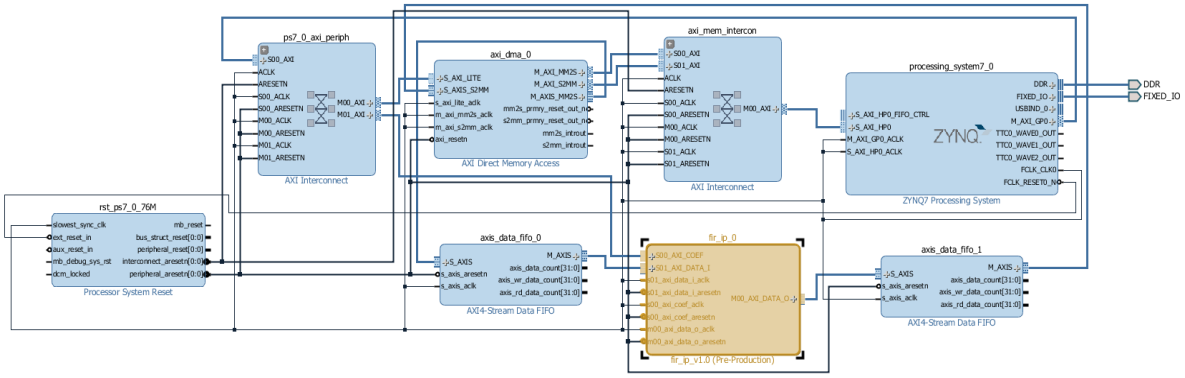
Zadatak 3:

Prilikom integracije filtra u digitalni sistem postoji mogućnost da FIR ima veću propusnu moć nego što je brzina pristizanja odbiraka. Kako bismo imali mogućnost zaustavljanja rada filtra potrebno je postojećim registarskim elementima dodati CE (*Clock enable*) ulaz. Dodati ovaj ulaz interfejsu filtra i propagirati CE do registarskih elemenata. Dodati stimulisanje CE signala u testbenč i prilagoditi čeker izmenama.

Motivacija za hardversku implementaciju digitalnih filtara

Ovaj dodatak vežbi nije materijal koji je potreban za polaganje ispita i ovde je dodat kako bi čitalac stekao bolju sliku o benefitima hardverske implementacije filtara to jest generalno algoritama. Dakle, zainteresovani studenti se ohrabruju da nastave čitanje materijala i ako postoji interesovanje za proces implementacije na realnom čipu obrade asistentu za pomoć.

Benefiti hardverske implementacije algoritama u odnosu na njihovu softversku implementaciju se najviše ogledaju u poboljšanju performansi u smislu povećanja propusne moći sistema. Da bismo uporedili performanse, projektovani FIR filtar je zapakovan u IP jezgro i okružen AXI protokolom. AXI protokol je najrasprostranjeniji u ARM baziranim sistemima. O pakovanju IP-jeva, AXI protokolu i uopšte integraciji složenih digitalnih sistema ćete učiti na predmetu Projektovanje složenih digitalnih sistema. Ovako zapakovano jezgro je integrisano u SoC, napisana je C aplikacija kako bi se na realnom hardveru proverila ispravnost rada filtra. Filtar je implementiran na *Xilinx Zynq7020* čipu. Ovaj čip ne predstavlja platformu vrhunskih performansi, ali je odličan za grubu procenu performansi hardverske implementacije sistema. Parametrizovani FIR je uspešno implementiran na frekvenciji od 70MHz. Pošto poznajemo arhitekturu filtra, jasno je da u svakom taktu može da prihvati novi sempl za obradu što znači da može da obradi 70 miliona semplova u sekundi (70Msps). Kao dodatnu informaciju možemo reći da je implementacija filtra 20. reda sa širinom ulaznih i izlaznih podataka od 32 bita, jednog DMA kontrolera (omogućava brz transfer podataka bez aktivnosti procesora) i dva FIFO bafera dubune 1000 (širine 32 bita) zauzela oko 11% hardverskih resursa FPGA čipa u okviru *Zynq7020* platforme. Blok dizajn formiran u *Vivado* alatu možemo videti na slici 11.



Slika 11: Test sistem parametrizovanog FIR filtra.

Po merenju performansi hardverski realizovanog filtra pristupamo softverskoj implementaciji. Merenje performansi je izvršeno na ARM Cortex-A9 jezgru (32-bitni CPU) koje se postoji u okviru *Zynq Processing Systema* (identična *Zynq7020* platforma). Jezgro radi na frekvenciji od 667MHz. U okviru 32-bitne ARM familije procesora pripada nekoj sredini u pogledu performansi. Široko je rasprostranjeno u domenu *low power* aplikacija, a ugrađivao se i u *smartphone* uređaje (*Samsung Galaxy S3*...). Test program je napisan u programskom jeziku C, a kod možemo videti u nastavku.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "test_vect.h"

int main()
```

```
{
   init_platform();
   int i,j;
   int output;
   long long int reg[FIR_ORD+1];
   //inicijalizacija registara
   for (i=0;i<=FIR_ORD;i++)
         reg[i] = 0;

   print("Test start!\n\r");
   for(i = 0; i<SAMPLE_CNT; i++){
      output = (int)((reg[FIR_ORD] + (((long long)b_s[FIR_ORD]) * input[i]))>>39)
         & 0x00ffffff;
      for(j = FIR_ORD; j>0;j--){
         reg[j] = reg[j-1] + (((long long)(b_s[j-1])) * input[i]);
      }
   }
   print("Test finished!\n\r");

   cleanup_platform();
   return 0;
}
```

Da bi poređenje bilo fer obe implementacije rade sa ulaznim podacima širine 32 bita, dok su međurezultati 64-bitni. Za čuvanje podataka registara smo koristili *long long int* tip. Posle inicijalizacije međurezultata (*reg* niz) pristupamo filtriranju ulaznog niza *input.* Ulazni niz i koeficijenti se nalaze u statičkim nizovima u drugom fajlu kako bi se jasno video deo koji implementira algoritam. Iz priloženog koda se jasno vidi da je softverska implementacija značajno jednostavnije. Jezgro algoritma predstavlja formiranje *output* vrednosti i *for* petlja koja potom sledi. Dakle, 3 linije koda. Za merenje performansi smo iskoristili PMU (*Performance Monitor Unit*) jedinicu koja postoji u okviru ARM jezgra. Kod pomocu kog smo merili performanse je prikazan u nastavku.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "test_vect.h"

static inline void init_perfcounters (int32_t do_reset, int32_t enable_divider)
{
   // in general enable all counters (including cycle counter)
   int32_t value = 1;

   // peform reset:
   if (do_reset)
   {
      value |= 2;     // reset all counters to zero.
      value |= 4;     // reset cycle counter to zero.
   }

   if (enable_divider)
      value |= 8;     // enable "by 64" divider for CCNT.
```

```
    value |= 16;

    // program the performance-counter control-register:
    asm volatile ("MCR p15, 0, %0, c9, c12, 0\t\n" :: "r"(value));

    // enable all counters:
    asm volatile ("MCR p15, 0, %0, c9, c12, 1\t\n" :: "r"(0x8000000f));

    // clear overflows:
    asm volatile ("MCR p15, 0, %0, c9, c12, 3\t\n" :: "r"(0x8000000f));
}

static inline unsigned int get_cyclecount (void)
{
  unsigned int value;
  // Read CCNT Register
  asm volatile ("MRC p15, 0, %0, c9, c13, 0\t\n": "=r"(value));
  return value;
}

int main()
{
  init_platform();
  int i,j;
  int output;
  long long int reg[FIR_ORD+1];
  //reg init
  for (i=0;i<=FIR_ORD;i++)
    reg[i] = 0;

  print("Test start!\n");

  /* enable user-mode access to the performance counter*/
  asm ("MCR p15, 0, %0, C9, C14, 0\n\t" :: "r"(1));
  /* disable counter overflow interrupts (just in case)*/
  asm ("MCR p15, 0, %0, C9, C14, 2\n\t" :: "r"(0x8000000f));
  // init counters:
  init_perfcounters (1, 1);
  unsigned int t = get_cyclecount();

  for(i = 0; i<SAMPLE_CNT; i++){
    output =(int) ((reg[FIR_ORD] + (((long long)b_s[FIR_ORD]) * input[i]))>>39)
      & 0x00ffffff;
    for(j = FIR_ORD; j>0;j--){
      reg[j] = reg[j-1] + (((long long)(b_s[j-1])) * input[i]);
    }
  }

  t = get_cyclecount() - t;
  printf("Clock cycle cnt/64 = %d\n",t);
  print("Test finished!\n");
```

```
    cleanup_platform();
    return 0;
}
```

Ideja merenja performansi je zasnovana na brojanju taktova potrebnih da se filtrira 1000 semplova. PMU jedinica ima ugrađene brojače koji broje rastuće ivice takt signala. Pre početka filtriranja pozvali smo funkciju *init_perfcounters* u kojoj smo resetovali vrednosti ovih brojača, dozvolili rad brojačima i podesili preskaliranje na 64. Ovakvim podešavanjem preskalera, brojač će povećavati svoju vrednost za 1 na svakih 64 takta. Pored ove funkcije napisana je i *get_cyclecount* funkcija koja kao povratnu vrednost vraća vrednost brojača. Kao što možemo videti, jezgro algoritma je ostalo nepromenjeno. Pre početka semplovanja smo sačuvali trenutnu vrednost registra brojača taktova, a po završetku obrade 1000 semplova oduzeli od trenutne vrednosti onu koju smo sačuvali pre ulaska u *for* petlju. Ova razlika predstavlja broj proteklih taktova (podeljen sa 64) za koji je CPU obradio 1000 semplova.

Po završetku rada programa ova razlika je iznosila 16810. Pošto smo postavili preskaler na 64 jasno je da je ukupno proteklo 16810 x 64 = 1075840 taktova. Pošto znamo da procesor radi na frekvenciji od 667MHz (667000000 taktova) možemo izračunati koliko ovakvih paketa od 1000 semplova može da obrati u toku 1 sekunde, 667000000/1075840≈620. Svaki paket je veličine 1000 semplova što znači da je propusna moć oko 620ksps. Ako ovo uporedimo sa hardverskim rešenjem dobićemo ubrzanje kao 70Msps/620ksps≈110 puta.

NAPOMENA: Softverska implementacija ovakvih algoritama se obično radi na procesorima specifične namene, DSP-ovima na kojima bismo najverovatnije dobili nešto bolje rezultate. Ipak ovo su specijalizovani procesori baš za ovu namenu te možemo reći da predstavljaju nešto između čistog softverskog i hardverskog rešenja problema. Takođe, za izuzetno precizno poređenje bi trebalo prvo proveriti koja forma filtra daje najbolje rezultate ako se implementira u softveru. Pored navedenog, moguće je izvršiti filtriranje i na nekom moćnijem procesoru gde bismo dobili značajnije ubrzanje. Sa druge strane ovakva hardverska implementacija nam takođe ostavlja mesta za poboljšanje kako arhitekture filtra tako i odabira FPGA čipova vrhunskih performansi. Sve u svemu iako je ovo procena urađena na jednom sistemu, jasno je da će hardverska realizacija dati ubrzanje u odnosu na softversku. Prednosti hardverske realizacije će biti još izraženije ako se red filtra dodatno poveća. Propusna moć hardverskog rešenja će ostati gotovo identična dok će softversko rešenje biti sve sporije i sporije sa porastom reda filtra.