Digitalna Integrirana Vezja in Sistemi – Nadgradnja vaje 7 in 8: “Upscaling” slike na 800x600 in IFS

Avtor: Andrej Barachini

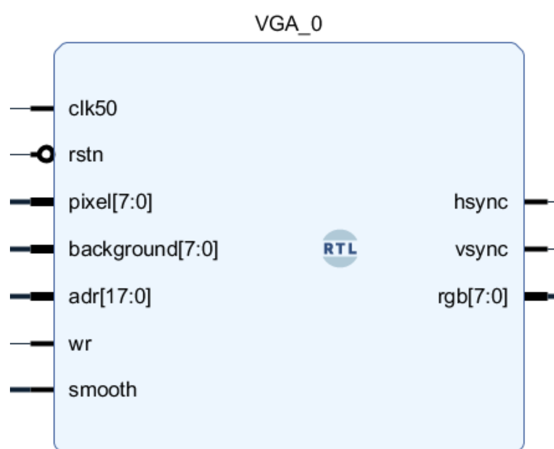
Uvod:

Motivacija za nadgradnjo se je začela pri zadnji laboratorijski vaji. Za izhod smo uporabljali SVGA 800x600 ločljivost, vendar smo izrisovali le zgornjih 256x256 točk, ostale pa pustili na barvi ozadja. Želja je bila torej narediti lep izris čez cel zaslon. Tu pa se pojavi problem z izvedbo na miniZED ploščici, saj ima Zynq7007 le 1.8 Mb Blok RAMa (50 blokov po 36 kb), za celoten zaslon pa bi potrebovali 8x800x600=3840000 bitov. Lahko bi se lotil uporabe drugih spominov (npr 512 MB DDR3L na miniZED), vendar sem se lotil drugače. Spomin bo držal le 8x400x300 podatkov (polovične dimenzije po 8 bitov na piksel), nato pa ga bo VGA modul povečal na končno velikost.

Ideje za večanje so dve in kasneje v izvedbi so uporabljene obe. Prva je zgolj podvajanje shranjene vrednosti čez 2x2 veliko območje, druga pa “interpolacija” barvnih vrednosti s povprečenjem, ki sem jo imenoval ‘smooth’, ker zgleda kot da je slika omehčana, zglajena.

Za konec sem izbral še grafiko za demonstracijo – Barnsleyevo praprot z algoritmom IFS (iterated function system). Prva izvedba jo generira programsko, druga pa strojno.

Modul VGA:



Slika 1: Modul VGA iz blokovne sheme in priključki

Zgoraj vidimo modul (vga.vhd) s priključki. Vhodi so 50 MHz ura **clk50**, signal za “active low” reset **rstn**, write enable **wr**, izbira načina delovanja **smooth**, dva vhoda 8-bitnih vrednosti pikslov **pixel** in **background**, ter 18-bitni **adr**, ki je zlepek x koordinate na MSB, ter y koordinate točke na LSB. Izhodi so sinhronizacijski signali VGA **hsync** in **vsync**, ter trenutna vrednost piksla **rgb**.

Sinhronizacija SVGA je narejena po standardu za 72Hz refresh rate: (**hsync**: front porch, sync dolžina in back porch 56, 120, 64 pikslov, za **vsync** pa 37, 6, 23 pikslov, kar je isto kot v

laboratorijski vaji), sicer trenutni piksel sestavljen iz dveh števcov **hst** in **vst**, ki gresta od 0 do 1040 in 0 do 666.

Zaradi sinhronega branja in določevanja indeksov, se prebere vrednost iz spomina glede prejšnji **hst** in **vst**, tako da bo izhod za en cikel zaostajal za pozicijo – spremenil sem tako da se sync signali (in območje izhoda) spreminjajo glede **hstold** in **vstold**, ki sta za en cikel zakasnjena signala.

Zadnji iz izhodnih signalov (**rgb**) pa je vrednost piksla, ki je znotraj vidnega območja enak **data** zunaj tega pa 0 kot rabi vga standard. (spodaj slika kode -rdeče flip flopi, oranžno procesi, modro sočasne določitve)

```
147     if rising_edge(clk50) then
148         hstold<=hst;--signala hs
149         vstold<=vst;--, tako zam
155         ---obicajni stevec za hst in vst
156         if hst < hstmax then
157             hst<=hst+1;
158         else
159             hst<=(others=>'0');
160             if vst < vstmax then
161                 vst<=vst+1;
162             else
163                 vst<=(others=>'0');
164             end if;
165         end if;
```

```
168         --sync signal
169         if hstold>=856 and hstold<976 then
170             hsync <= '1';
171         else
172             hsync <= '0';
173         end if;
174
175         if vstold>=637 and vstold<643 then
176             vsync <= '1';
177         else
178             vsync <= '0';
179         end if;
```

```
313 prgb: process(clk50, hstold, vstold, data)

321         if hstold <(xsize-1) and vstold < (ysize-1) then
322             rgb <= data;
323         else
324             rgb<=x"00";
325         end if;
```

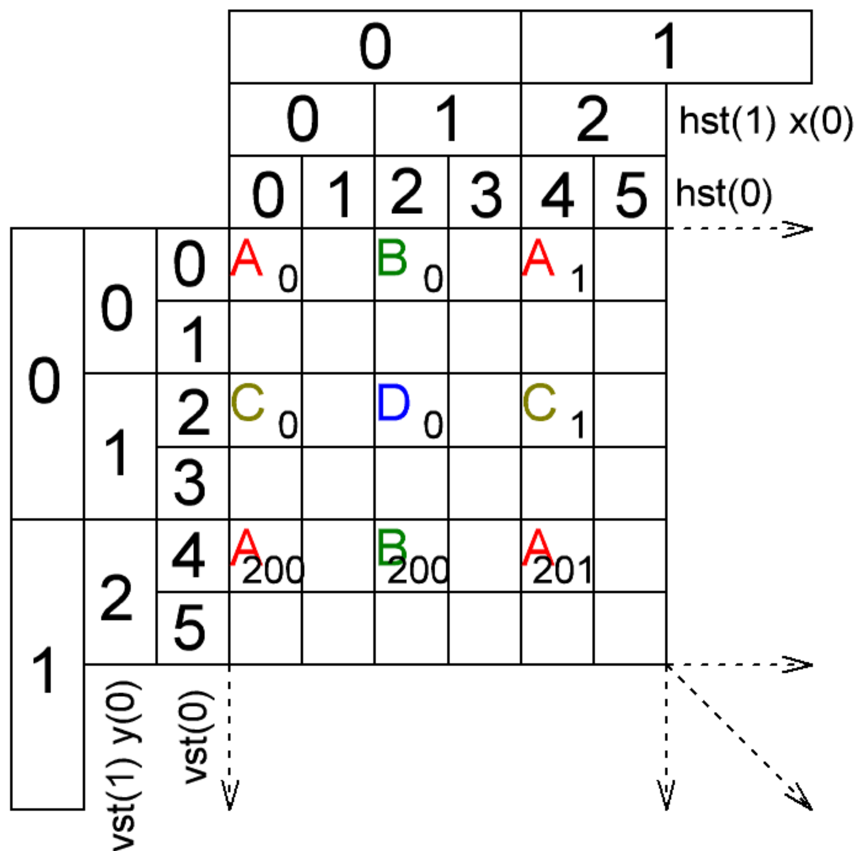
Slika 2: Koda (vhd) 1, števci, sinhronizacija in izhod

Kaj se nahaja v **data** je odvisno od načina delovanja in lokacije **hst** ter **vst**. Za boljše razumevanje branja je najprej potrebno pogledati kako se podatki zapisujejo v spomin.

Blok-RAM je spomin sestavljen (v tem primeru) iz 36 kb blokov, torej najmanjša enota, ki jo lahko vzamemo je 36 kb. Drug tip RAMa ki je dostopen v tem FPGA je porazdeljeni RAM z “look-up" tabelami. Kateri se uporabi določi sintetizator sam, glede situacijo uporabe in lastnosti branja ter pisanja. Za porazdeljeni RAM imamo na tem čipu manj prostora (8x400x300 bi zasedlo več kot 100% enot kar pomeni da ni primeren za našo uporabo.

Vendar pa ima Blok-RAM poleg tega da ima več prostora, le da je ta omejen na minimalne bloke še drug problem. Pišemo in brišemo lahko (sihnrono) znotraj enega cikla zgolj enkrat. To pomeni da ne bi morali izvajati povprečenja s sosednjimi podatki, če imamo samo en velik spomin. Zato sem razdelil celoten spomin na 4 200x150 (30000 krat po 8 bitov) velike enote. Te so razdeljene izmenično v “banke” A, B, C in D tako da se nikoli ne dotikata podatka z enako oznako.

Za lažjo predstavo sem na naslednji strani dodal sliko razdelitve slikovnega in spominskega polja.



Slika 3: Mrežna razdelitev slike in spomina

Številke na vrhu so horizontalne (vezane na **hst** oz **xin**) ter gredo do 800 v spodnji vrsti, 400 v srednji in 200 na vrhnji. Podobno na levi gremo po vertikali (**vst** oz **yin**) do 600, 300 in 150. Vidimo da če želimo na primer vpisati na sliki na x=1 in y=2 potem zapisujemo v **ramB** na lokacijo 200. Tako je enačba za naslov **adr2**=**xin**/2+ (**yin**/2)*200, pri čemer deljenja zaokrožujejo navzdol (izvedeno kar tako da odstranimo LSB bit), **xin**(0) in **yin**(0) pa izbereta v katero banko se zapiše.

Komentar na označbe, hst(0) seveda je vedno le 0 ali 1, ne gre do 800, vbistvu bi bilo prav označiti hst(10 downto 0), vrstico gor hst(10 downto 1) ali xin(9 downto 0) in še višje hst(10 downto 2) ali xhalf.

```
105   xin <= unsigned(adr(17 downto 9));
106   yin <= unsigned(adr(8 downto 0));
109   xhalf<= '0' & xin(8 downto 1);
110   yhalf<= '0' & yin(8 downto 1);
114   wrEn<= (wr and wrValid);
```

```
131   ymult <= resize((yhalf * (xsize/4)),16);
132   adr2 <= std_logic_vector(xhalf + ymult) ;
```

```
121       if( (xin<(xsize/2)) and ( yin<(ysize/2))) then
122           wrValid<='1';
123       else
124           wrValid<='0';
125       end if;
```

```
138   xyl<= xin(0) & yin(0);
139   hvl<= hstold(1) & vstold(1);
140   hv0<= hstold(0) & vstold(0);
```

```
183       if (wrA = '1') then
184          ramA(to_integer(unsigned(adr2)))<= pixelold;
185       end if;
186       if (wrB = '1') then
187          ramB(to_integer(unsigned(adr2)))<= pixelold;
188       end if;
189       if (wrC = '1') then
190          ramC(to_integer(unsigned(adr2)))<= pixelold;
191       end if;
192       if (wrD = '1') then
193          ramD(to_integer(unsigned(adr2)))<= pixelold;
194       end if;
```

```
231   selectwr<= xyl & wrEN;
232   with (selectwr) select
233     wrji <= "0001" when "001",
234             "0010" when "011",
235             "0100" when "101",
236             "1000" when "111",
237             "0000" when others;
238
239      wrA <= wrji(0);
240      wrC <= wrji(1);
241      wrB <= wrji(2);
242      wrD <= wrji(3);
```

Slika 4: Koda (vhd) 2, izbira banke, naslova in pisanje BRAMa

Zgoraj je koda zapisovanja. Iz vhoda prepiše **x** in **y**, ter zdeli s pol in izračuna naslov, iz vhoda pa se določi tudi ali je lokacija prava, tu sta **xsize** in **ysize** integer velikost zaslona za splošnost, torej 800 in 600 (če bi zapisali na x= 600 y=3 na primer bi prekrili x=200 y=4). Če je lokacija validna in imamo **wr** vhod na '1' tedaj se glede spodnji bit x in y določi kateri wrA-D je aktiven (le en na zapis).

Podobno se tudi bere v primeru da je **smooth** na '0' - takrat je izhod ves čas kar **data0**, ki je vrednost iz spomina, naslov pa izračunan podobno kot za pisanje le da je **hst** in **vst** potrebno deliti s 4 da pridemo na 200x150 mrežo. Ker sta spodnja dva bita pri tem odrezana bo vedno naslov zgoraj levo v "kvadratku" (pravi za branje) in ne potrebujemo dodatne logike, dokler prav izberemo banko:

```
246   vmult <= resize((vst(9 downto 2) * (xsize/4)),17);
247   vmultnext <= resize((vstnext(9 downto 2) * (xsize/4)),17);
250   ramindex <= hst(10 downto 2) + vmult;
```

```
218       dataA<=ramA(to_integer(unsigned(ramindexA)));
219       dataB<=ramB(to_integer(unsigned(ramindexB)));
220       dataC<=ramC(to_integer(unsigned(ramindexC)));
221       dataD<=ramD(to_integer(unsigned(ramindexD)));
```

```
139   hvl<= hstold(1) & vstold(1);
294   with (hvl) select
295     data0 <= dataA when "00",
296              dataC when "01",
297              dataB when "10",
298              dataD when others;
```

Slika 5: Koda (vhd) 3, izračun naslova in branje

Beremo sinhrono iz vseh, izberemo izhod zato glede prejšnje indekse.

Za **smooth** nastavitev pa sem dodal podmodul color_smooth.vhd, ki sem si ga zastavil z generiki da bi deloval za različne barvne kombinacije. Za njegovo delovanje sem dodal še podmodule seštevalnikov s prelivom. Modul razdeli piksel na barvne komponente ter vsak kanal posebej sešteje in zdeli s številom vhodov (za **M**=2 in **M**=4), pri čemer sem zraven vštel še korekcijski faktor **cor2RGB** in **cor4RGB** ki je konstanta za 1 ali 2 -> to sem dodal ker sicer vedno zaokrožuje navzdol, tako pa dobimo malo boljšo zapolnitev. Seveda bi problem bil veliko manjši če bi imeli več bitov za predstavitev barve ali bi računali z več vhodi.

```vhdl
entity color_smooth is
    generic (R : integer := 3; G : integer := 3; B : integer := 2; M : integer := 4);
    Port ( pixel1 : in  std_logic_vector((R+G+B-1) downto 0);
           pixel2 : in  std_logic_vector((R+G+B-1) downto 0);
           pixel3 : in  std_logic_vector((R+G+B-1) downto 0);
           pixel4 : in  std_logic_vector((R+G+B-1) downto 0);
           pixelavg : out  std_logic_vector((R+G+B-1) downto 0));
end color_smooth;
```

```vhdl
66      r1<= pixel1((R+G+B-1) downto (G+B));
67      r2<= pixel2((R+G+B-1) downto (G+B));
68      r3<= pixel3((R+G+B-1) downto (G+B));
69      r4<= pixel4((R+G+B-1) downto (G+B));
70
71      g1<= pixel1((G+B-1) downto (B));
72      g2<= pixel2((G+B-1) downto (B));
73      g3<= pixel3((G+B-1) downto (B));
74      g4<= pixel4((G+B-1) downto (B));
75
76      b1<= pixel1((B-1) downto 0);
77      b2<= pixel2((B-1) downto 0);
78      b3<= pixel3((B-1) downto 0);
79      b4<= pixel4((B-1) downto 0);
```

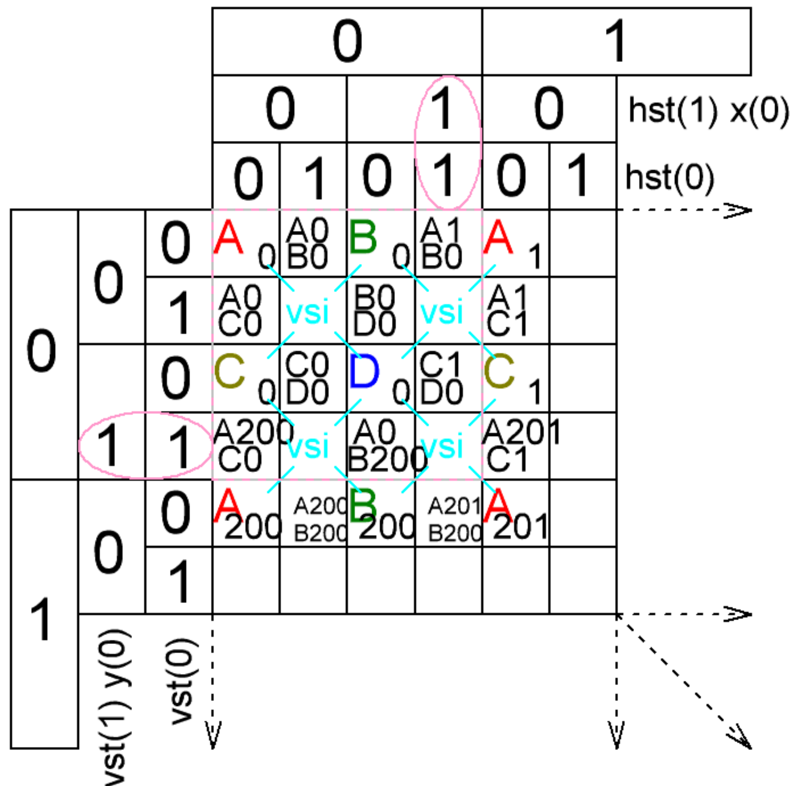
```vhdl
81   gen_two: if M = 2 generate
82      ADD_r: adder generic map (N=>R) port map (r1, r2, rsum2);
83      ADD_g: adder generic map (N=>G) port map (g1, g2, gsum2);
84      ADD_b: adder generic map (N=>B) port map (b1, b2, bsum2);
85      ADD_cr: adder generic map (N=>R+1) port map (rsum2, cor2R, rsum2cor);
86      ADD_cg: adder generic map (N=>G+1) port map (gsum2, cor2G, gsum2cor);
87      ADD_cb: adder generic map (N=>B+1) port map (bsum2, cor2B, bsum2cor);
88
89      ravg<=rsum2cor(R downto 1);
90      gavg<=gsum2cor(G downto 1);
91      bavg<=bsum2cor(B downto 1);
92   end generate;
```

```vhdl
94   gen_four: if M = 4 generate
95      ADD_r: adder_four generic map (N=>R) port map (r1, r2, r3, r4, rsum4);
96      ADD_g: adder_four generic map (N=>G) port map (g1, g2, g3, g4, gsum4);
97      ADD_b: adder_four generic map (N=>B) port map (b1, b2, b3, b4, bsum4);
98      ADD_cr: adder generic map (N=>R+2) port map (rsum4, cor4R, rsum4cor);
99      ADD_cg: adder generic map (N=>G+2) port map (gsum4, cor4G, gsum4cor);
100     ADD_cb: adder generic map (N=>B+2) port map (bsum4, cor4B, bsum4cor);
101
102     ravg<=rsum4cor(R+1 downto 2);
103     gavg<=gsum4cor(G+1 downto 2);
104     bavg<=bsum4cor(B+1 downto 2);
105  end generate;
```

```vhdl
107     pixelavg <= ravg & gavg & bavg;
```

Slika 6: Koda (vhd) 4, opis povprečenja barvnih kanalov

Za pravilno vezavo the moramo pogledati dopolnjeno sliko s povprečji:



Slika 7: Mreža slike pri branju za 'smooth' način

Na sliki se veliko dogaja, zato ločimo problem na več korakov. Prvo je indeksiranje rama: vidimo namreč da se v vrstici, kjer sta spodnja dva bita “11” že pojavi naslednji indeks po vertikali (za A in B ko **vst**(1 downto 0) = “11”) po horizontali (za A in C ko **hst**(1 downto 0) = “11”) in diagonali (za A ko in **vst**(1 downto 0) & **hst**(1 downto 0) = “1111”). To dosežemo z pomožnimi vrednostmi **vstnext** in **hstnext**, te so naslendnji **hst** in **vst** razen ko smo na robu slike.

Iz teh določimo tudi **vmult** in **vmultnext** ter tako dobimo vse možne indexe sosedov in trenutnega kvadratka, kot smo to storili prej za **ramindex** (h/4+v/4*200). Ko imamo te vrednosti le izbiramo glede spodnja dva bita **vst** in **hst**, združena v **selectvsthst**. Izbiralniki so opisani z with – select stavki, ki so za B in C preprosti, za A pa so napisane vse opcije skrajnega roba.

```
200      if ((vst<(ysize-1)) and not(hst=(hstmax-0)) )then
201         vstnext <= vst+1;
202      elsif ((hst=(hstmax-0)) and (vst<(ysize-2)) )then
203         vstnext <= vst+2;
204      else
205         vstnext<=to_unsigned((ysize-1),10);
206      end if;
207
208      if hst<(xsize-2) then
209          hstnext <= hst+2; --za sosede
210      elsif hst=(hstmax-0) then
211         hstnext <= to_unsigned(1,11);
212          else
213         hstnext<=to_unsigned((xsize-1),11);
214          end if;
```

```
246   vmult <= resize((vst(9 downto 2) * (xsize/4)),17);
247   vmultnext <= resize((vstnext(9 downto 2) * (xsize/4)),17);
250   ramindex <= hst(10 downto 2) + vmult;
251   ramindexvnext<= hst(10 downto 2) + vmultnext;
252   ramindexhnext<= hstnext(10 downto 2) + vmult;
253   ramindexdnext<= hstnext(10 downto 2) + vmultnext;
```

```
218      dataA<=ramA(to_integer(unsigned(ramindexA)));
219      dataB<=ramB(to_integer(unsigned(ramindexB)));
220      dataC<=ramC(to_integer(unsigned(ramindexC)));
221      dataD<=ramD(to_integer(unsigned(ramindexD)));
```

```
260   with (selectvsthst) select
261     ramindexA <= ramindexdnext when "1111",
262                  ramindexvnext when "1100",
263                  ramindexvnext when "1101",
264                  ramindexvnext when "1110",
265                  ramindexhnext when "0011",
266                  ramindexhnext when "0111",
267                  ramindexhnext when "1011",
268                  ramindex      when others;
271   with (selectvsthst(3 downto 2)) select
272     ramindexB <= ramindexvnext when "11",
273          ramindex when others;
276   with (selectvsthst(1 downto 0)) select
277     ramindexC <= ramindexhnext when "11",
278          ramindex when others;
281   ramindexD <= ramindex;
```

```
258   selectvsthst <= std_logic_vector(vst(1 downto 0)) & std_logic_vector(hst(1 downto 0));
```

Slika 8: Koda (vhd) 5, računanje in izbiranje naslednjih indexov za rob osnovnega kvadrata

Ko imamo enkrat prave vrednosti za podatke **dataA-D** jih vežemo na vse možne kombinacije v color_smooth module, kot je vidno na vrhu slike:

```
98    avghab: color_smooth generic map (3,3,2,2) port map (dataA, dataB, zero8, zero8, avgAB);
99    avghcd: color_smooth generic map (3,3,2,2) port map (dataC, dataD, zero8, zero8, avgCD);
100   avgvac: color_smooth generic map (3,3,2,2) port map (dataA, dataC, zero8, zero8, avgAC);
101   avgvbd: color_smooth generic map (3,3,2,2) port map (dataB, dataD, zero8, zero8, avgBD);
102   avgmid: color_smooth generic map (3,3,2,4) port map (dataA, dataB, dataC, dataD, avgALL);
```

```
286   datah <= avgCD when std_logic(vst(1)) = '1' else
287            avgAB;
290   datav <= avgBD when std_logic(hst(1)) = '1' else
291            avgAC;
```

```
294   with (hvl) select
295     data0 <= dataA when "00",
296              dataC when "01",
297              dataB when "10",
298              dataD when others;
```

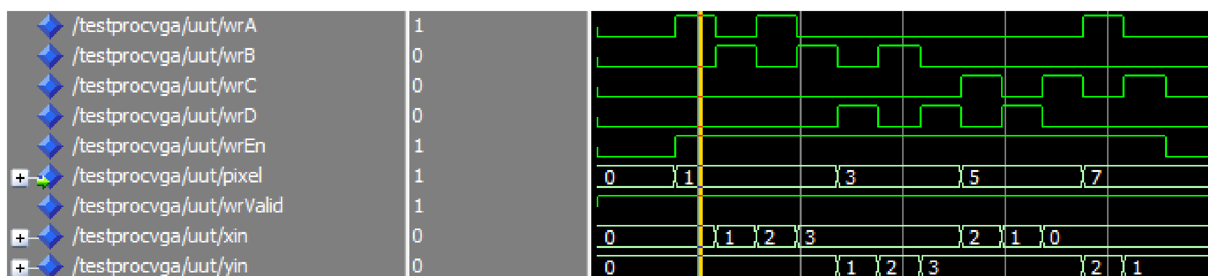
```
139   hvl<= hstold(1) & vstold(1);
140   hv0<= hstold(0) & vstold(0);
```

```
303   selecthv0 <= hv0 & smooth;
304   with (selecthv0) select
305     data <= data0 when "001",
306             datav when "011",
307             datah when "101",
308             avgALL when "111",
309             data0 when others;
```

Slika 9: Koda (vhd) 6, vezava color_smooth in izbira izhodov glede spodnja dva bita hst in vst

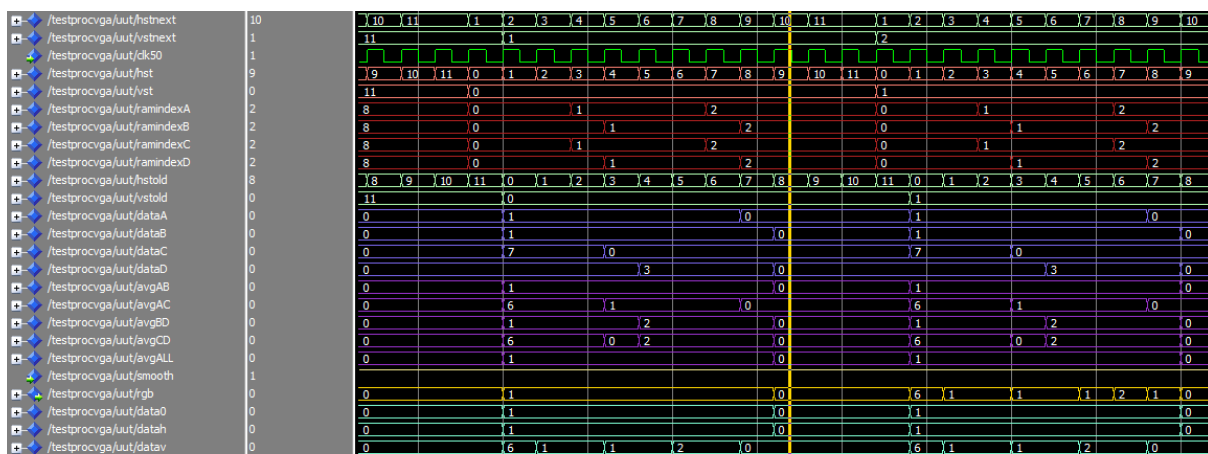
Imamo zdaj **avgAB**, **avgCD**, **avgAC**, **avgBD** in **avgALL**. Če spet pogledamo sliko 7, potem lahko razberemo da **hv0** & **smooth** določa ali beremo **data0**, povprečje po horizontali **datah**, po vertikali **datav** ali povprečje vseh **avgALL**. Kdaj je **avgh** **avgAB** ali **avgCD** ter **avgv** **avgAC** ali **avgBD** pa določa **hv1**.

Delovanje modula vga.vhd sem tako najprej preveril v ModelSIM-u. Povezal sem vse terminale, velikost pa zmanjšal iz 800x600 na 12x12 poleg tega pa glede izhod polnil "signal izhod : ram_type_big;" da si lahko predstavimo celotno sliko. Izrisal sem kvadrat s stranico 1, 3, 5 in 7. Tu je potrebno upoštevati da je 5 vbistvu g=1,b=1 in 7 g=1, b=3, saj sta za modro barvo dodeljena le sponja 2 bita, tako interpretirajmo tudi rezultate (tak izpis sem uporabil zaradi preglednosti, saj celotnega binarnega ne bi spravil lepo na sliko):



Slika 10: vga simulacija, zapisovanje kvadratka

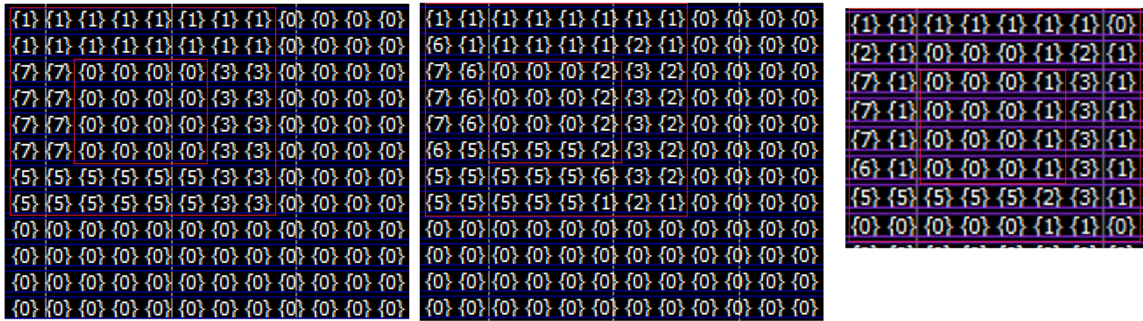
Glede **xin** in **yin** se ko je **wrEn** '1' dvigne ustrezni wr in shrani vrednost na vhodu.



Slika 11: vga simulacija, potek signalov (smooth 1)

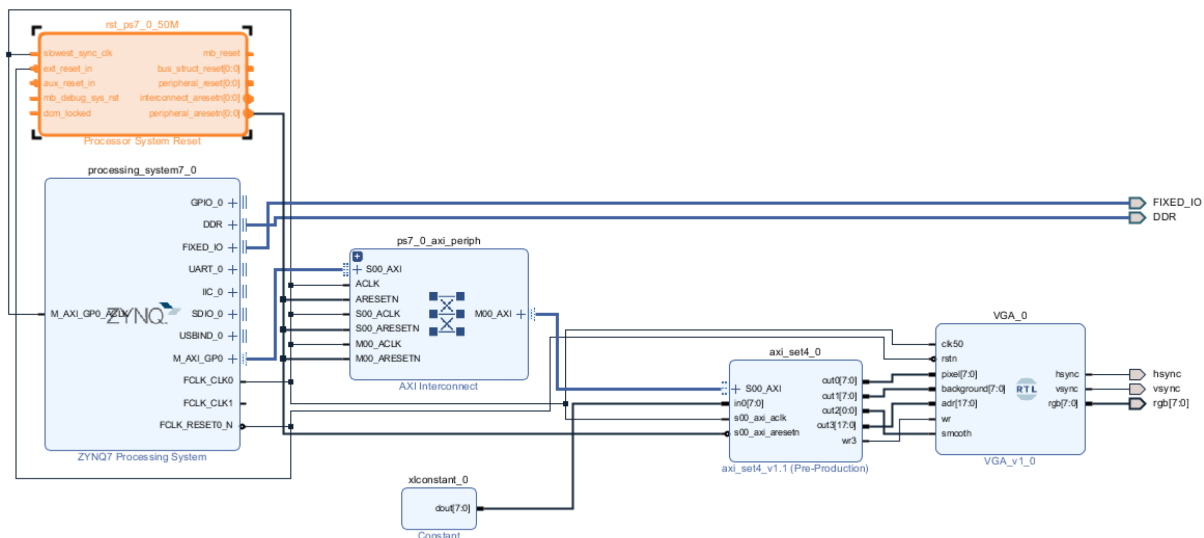
Slika je sicer precej majhna, ampak lahko vidimo kako se spreminjajo glavni signali v modulu. S števcema **hst** in **vst** se spreminjajo vsi ramindexi, ki se pri 3 za A in B že postavita naprej. Data signali se spreminjajo s **hstold** in **vstold**, prav tako avg signali. Če bi bil **smooth** '0' bi **rgb** kar bil enak **data0**, v tem primeru pa izbira med **data0**, **datah**, **datav** in **avgALL** kot smo prej opisali v kodi.

Končna slika je spodaj. Na levi imamo **smooth** '0' na sredini **smooth** '1' s korekcijo in desno **smooth** '1' brez korekcije. Čez sem pri vseh izrisal okvir enake dimenzije, da si lažje primerjamo. Vidimo da smooth '0' zgolj podvoji vrednost na večje piksle velikosti 2x2, smooth '1' pa se trudi narediti mehke prehode (iz črne na močno modro šibkejša modra). Korekcija je bila dodana ker se je v kotih in ozkih črtah povsem zgubil podatek o barvi zaradi zaokroževanja navzdol.



Slika 12: izhodne slike različnih načinov delovanja

Ko sem delovanje tako preveril v simulaciji sem modul prestavil v vivado okolje kjer sem ga vmestil v blokovno shemo podobno kot v zadnji laboratoriski vaji:



Slika 13: Blokovna shema projekta (oddaja 3)

Glavna komunikacija med procesorjem in vga modulom je izvedena prek rahlo spremenjenega axi_set4 vmesnika. Osnovnega naslova nisem spreminjal, sem pa spremenil out2 za nastavljanje smooth in out3 na 18 bitno.

V SDK okolju sem za prvi preizkus popravil oziroma dodal nekaj funkcij. *Pixel* da vzame koordinate iz parametrov in ga pošlje na prav axi naslov, *smoothold* nastavi smooth bit na 1, *clear* čez celoten spomin popiše ničle, *line* pa je Bresenhamov algoritem iz vaje.

V main sem izrisal nekaj črt, jih pobrisal, izrisal nove in spreminjal smooth nastavitev da sem preveril delovanje.

Koda funkcij je na naslednji strani (vzeta iz urejevalnika Notepad++).

Ko sem tako preveril delovanje sem za demonstracijo vzel izris praproti z IFS algoritmom – bolj podrobno v naslednjem poglavju.

```c
16  static inline void pixel(int x, int y) //pisi piksel x y
17  {
18      if(y>300) y=299;
19      if(x>400) y=399;
20      int xy = ((x<<9)+(y)); //doloci adr blok rama iz x in y
21      Xil_Out32(XPAR_AXI_SET4_0_S00_AXI_BASEADDR+12, xy); //na prav naslov da adr
22  }
61  static inline void smoothold(int s)
62  {
63      int one = 0b00000001;
64      if(s==1)
65      {
66          Xil_Out32(XPAR_AXI_SET4_0_S00_AXI_BASEADDR+8, one);
67      }
68      else
69      {
70          Xil_Out32(XPAR_AXI_SET4_0_S00_AXI_BASEADDR+8, 0);
71      }
72  }
218 /* Bresenhamov algoritem za risanje */
219 void line(int x0, int y0, int x1, int y1) {
220
221   int dx = abs(x1-x0), sx = x0<x1 ? 1 : -1;
222   int dy = abs(y1-y0), sy = y0<y1 ? 1 : -1;
223   int err = (dx>dy ? dx : -dy)/2, e2;
224
225   for(;;){
226     pixel(x0,y0);
227
228     if (x0==x1 && y0==y1) break;
229     e2 = err;
230     if (e2 >-dx) { err -= dy; x0 += sx; }
231     if (e2 < dy) { err += dx; y0 += sy; }
232   }
233 }
235 void clear()
236 {
237     Xil_Out8(XPAR_AXI_SET4_0_S00_AXI_BASEADDR, 0x00);
238     for (int i=0; i<400; i++){
239         for (int j=0; j<300; j++){
240             //Xil_Out16(XPAR_AXI_SET4_0_S00_AXI_BASEADDR+12, ((i<<9) + j));
241             pixel(i,j);
242         }
243     }
244 }
```

Slika 14: Koda (C) 1, funkcije za prvi preizkus

IFS algoritem (in implementacija v C kodi):

IFS (Iterated Function Systems) so v matematiki metoda za generiranje fraktalov. Znani primeri tako generiranih fraktalov so Mandelbrotov set, Sierpinskijev trikotnik in Barnsleyjeva praprot. Sistem je sestavljen ponavadi iz več kopij samega sebe na večih velikostnih redih. Algoritem sloni na nizu matematičnih transformacij (v našem primeru uporabljamo afine transformacije).

Izbral sem Barnsleyjevo praprot, katere algoritm se glasi tako:

1. Vzemi poljubno točko, s svojim x in y (v našem primeru (0,0))
2. Naključno izberi set transformacij izmed štirih naštetih vsako s svojo verjetnostjo
3. Izvedi transformacijo na star x in y da dobimo nov x in y
4. Ponavljaj

Transformacije pa so vidne na spodnji sliki:

$$f(x,y) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

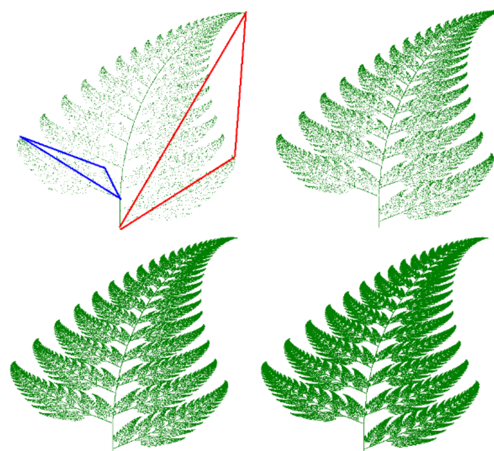
w	a	b	c	d	e	f	p	Portion generated
f_1	0	0	0	0.16	0	0	0.01	Stem
f_2	0.85	0.04	−0.04	0.85	0	1.60	0.85	Successively smaller leaflets
f_3	0.20	−0.26	0.23	0.22	0	1.60	0.07	Largest left-hand leaflet
f_4	−0.15	0.28	0.26	0.24	0	0.44	0.07	Largest right-hand leaflet

$$f_1(x,y) = \begin{bmatrix} 0.00 & 0.00 \\ 0.00 & 0.16 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$f_2(x,y) = \begin{bmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.60 \end{bmatrix}$$

$$f_3(x,y) = \begin{bmatrix} 0.20 & -0.26 \\ 0.23 & 0.22 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.60 \end{bmatrix}$$

$$f_4(x,y) = \begin{bmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 0.44 \end{bmatrix}$$



Slika 15: Algoritem in primer izrisa, pobrano iz Wikipedije

V C-ju je sestavljen tak algoritm iz 2D tabele koeficientov, katere vrsta predstavlja posamezno transformacijo ter pomožne tabele za barve, meje in končno skaliranje na naš zaslon.

Pred začetkom si pripravimo x in y kot realna števila (float) na 0.

V vsaki iteraciji nato generiramo naključno številko s funkcijo rand(), katere ostanek pri deljenju s 100 bo naključno naravno število (integer) od 0 do vključno 99. To število primerjamo z mejami in s tem določimo kater indeks za vrstico transformacije in barve uporabljamo.

Sledi računanje novih vrednosti po transformaciji, ves čas ostanemo v prostoru realnih števil za ustrezno natančnost. Ko izračunamo xnew in ynew dobljene vrednosti skaliramo, transliramo (tabela preslikave) in preslikamo v celoštevilo za izris xout, yout ki pa ga izrišemo s funkcijo pixel. Zadnji korak je da xnew in ynew shranimo nazaj v x in y ter se vrnimo na začetek zanke.

Preslikava je potrebna, ker bi sicer x in y dobili v rangu -2.5 do 2.5 za x in 0 do 10 za y kar je idealno za izris na računalniku v python skripti recimo, ampak ni zaloga vrednosti ustrezna za izris z našim vga modulom. Sliko je treba tako obrniti po y (saj vrednosti naraščajo navzdol) in premestiti za izris, računanje novih koordinat pa poteka z vrednostmi pred preslikavo.

```
130  /* IFS algoritm praprot */
131  void IFS(float f[][6], int meje[], int preslikava[], int niter, int barvaj) {
132  
133    if(barvaj == 0){
134      Xil_Out8(XPAR_AXI_SET4_0_S00_AXI_BASEADDR, 133); // pixel barva zelena
135    }
136  
137    float xf =0;
138    float yf =0;
139    int x =0;
140    int y =0;
141    int r;
142    int c; //stevilka stolpca
143    for(int i=0; i<niter;i++){
144        //dolocimo obmocje in transformacijo nakljucno
145      r= rand() % meje[3];
146      if (r< meje[0]){ c=0;}
147      else if(r<meje[1]){c=1;}
148      else if(r<meje[2]){c=2;}
149      else {c=3;}
150  
151      //uporabimo transformacijo
152      float xfnew=xf*f[c][0]+yf*f[c][1]+f[c][4];
153      float yfnew=xf*f[c][2]+yf*f[c][3]+f[c][5];
154  
155      //shranimo prejsnjo vrednost za naslednjo iteracijo
156      xf=xfnew;
157      yf=yfnew;
158  
159  
160      //skaliramo, zamaknemo in castamo v integer za prikaz
161      x= (int) (xf*preslikava[0]+preslikava[1]) ;
162      y= (int) (yf*preslikava[2] + preslikava[3]);
163  
164  
165      //ce imamo uporabljeno 'barvaj' nastavitev doda barvo tockam za posamezno regijo
166      if(barvaj){
167      Xil_Out8(XPAR_AXI_SET4_0_S00_AXI_BASEADDR, barve[c]); // pixel barva iz globalne tabele
168      }
169  
170      //izrisemo
171      pixel(x,y); //izrise pixel na naslove x y
172  
173    }
174  }
```

Slika 16: Koda (C) 2, IFS algoritem za Barnsleyjevo praprot

Barvaj nam ko je ena vklopi uporabo tabelo barv, ko 0 pa izriše enobarvno. Niter nam poda koliko iteracij rišemo.

Primer kode maina:

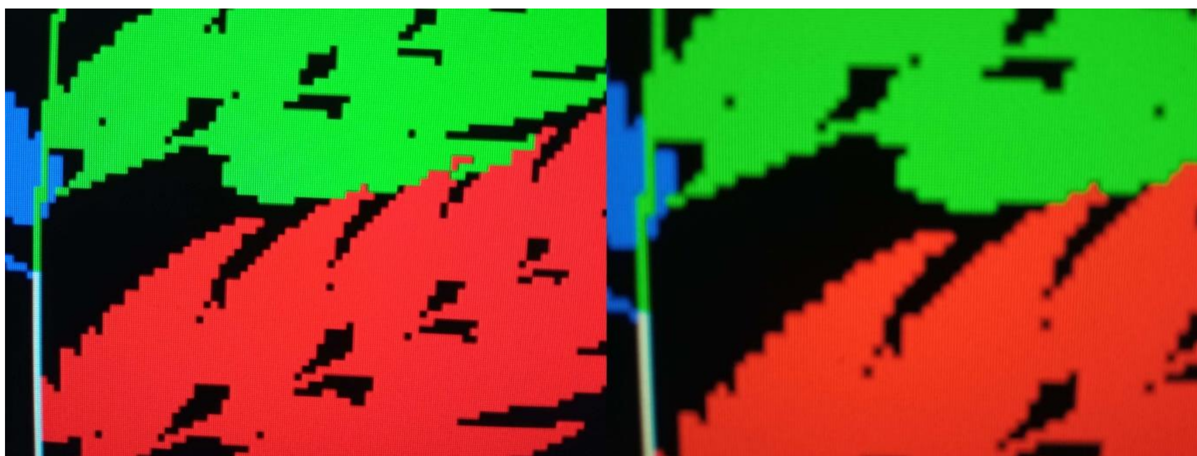
```c
int main()
{
    Xil_Out8(XPAR_AXI_SET4_0_S00_AXI_BASEADDR, 0b01011100); // pixel barva zelena
    Xil_Out8(XPAR_AXI_SET4_0_S00_AXI_BASEADDR+4, 0x00); // background

    float f[4][6] = {
    {0, 0, 0,   0.16,   0,  0},
    {0.85,  0.04, -0.04,    0.85,   0,  1.60},
    {0.20, -0.26,   0.23,   0.22,   0,  1.60},
    {-0.15, 0.28,   0.26,   0.24,   0,  0.44}
    };
    int meje[4] ={1,86,93,100};
    int preslikava[4]= {50,150,-26,280};

    smoothold(0);
    int num_iterations= 500000;

    clear();
    IFS(f, meje, preslikava, num_iterations*5, 0);
    smoothold(1);
    clear();
    IFS(f, meje, preslikava, num_iterations*5, 1);

    for (int i=0; i<5;i++){
        clear();
        //preslikava[1]= preslikava[1]-5;
        for (int j=1; j<4;j++){
            f[j][1]=f[j][1]+0.01;
            f[j][2]=f[j][2]+0.01;
        }
        f[0][3]=f[0][3]*0.98;
        IFS(f, meje, preslikava, num_iterations, 1);
    }
    for (int i=0; i<10;i++){
            clear();
            //preslikava[1]= preslikava[1]+5;
            for (int j=1; j<4;j++){
                f[j][1]=f[j][1]-0.01;
                f[j][2]=f[j][2]-0.01;
            }
            f[0][3]=f[0][3]*1.02;
            IFS(f, meje, preslikava, num_iterations, 1);
        }
    clear();

    Xil_Out8(XPAR_AXI_SET4_0_S00_AXI_BASEADDR, 0b11111100); // pixel barva rumena
    line(0,0,399,299); //crta po diagonali

    sleep(1);



    return 0;
}
```

Slika 17: Koda (C) 3: main za izris praproti

Tako najprej izrišemo enobarvno praprot, nato vklopimo smooth in večbarvno praprot (med vsakim izrisom počistimo zaslon), nato pa v zanki spreminjamo koeficiente in s tem dobimo popačene praproti.

Slika 18: praprot brez in s smooth

Na daleč je (sploh na tej sliki zaslona s telefonom) težko ločiti sliki, ker je večja razlika zaradi osvetlitve, drugačnih lokacij točk, zato sem dodal še približano sliko:



Slika 19: praprot brez in s smooth od blizu

Vidi se kot bi sliko na desni zameglili, učinek podoben filtriranju slike na način “gaussian blur” čeprav to ni. Del tega je tudi da se telefon ni želel izostriti na že zamegljeno sliko.

Končni rezultat ni nič presenetljivega, vendar bi se modul vga lahko uporabljal kot osnovna platforma za kak drug projekt, ki se ukvarja s čim bolj funkcionalnim, ali mogoče igrico, ki bi se tako lahko poganjala čez cel zaslon z 8bitnimi barvami.

Nadgradnja: implementacija algoritma IFS na FPGA

Za implementacijo na FPGA je potrebno nekaj stvari spremeniti: kako se shrani in spreminja koeficiente, kako se generira naključno število za izbiro regije, kako izvesti račune z realnimi števili, ter kako pobrisati spomin med spremembami.

Najprej sem se lotil računanja z realnimi števili. Računanje s plavajočo vejico (float) je zahtevno zato sem si izbral fiksno vejico, ki sem jo izbral z upoštevanjem velikostnega reda števil (ker so vsa števila največ dvomestna sem izbral 32 bitni vektor z 20 biti za predstavitev za decimalno vejico). Drug način predstave je da pred računanjem število pomnožimo z 2^20 in na koncu nazaj zdelimo s toliko. Pri seštevanju ni posebnih omejitev le da seštevamo števili z istoležno vejico (fixedpoint20 z fixedpoint20 in normalni signed z signed). Pri množenju pa se nam podvoji število mest na 64 bitov, ki pa jih moramo prav prestaviti nazaj, torej v primeru množitelja v fixedpoint20 rabimo izhod prebrati od (51 dowtno 20) da dobimo nazaj isto ureditev za zmnožek kot jo ima množenec. Da ohranim predznačenost sem nekoliko spremenil da je MSB enak MSBju velikega zmnožka. Tako množenje sem vmestil v modul multiply_fp.vhd:

```
 5  entity multiply_fp is
 6      generic (N : integer := 32; P: integer := 20);
 7      Port ( in1 : in  std_logic_vector((N-1) downto 0);
 8             in2 : in  std_logic_vector((N-1) downto 0);
 9             product : out  std_logic_vector((N-1) downto 0));
10  end multiply_fp;
11
12  architecture Behavioral of multiply_fp is
13  signal producttemp: std_logic_vector(63 downto 0);
14  begin
15      producttemp <= std_logic_vector(signed(in1) * signed(in2));
16      product(N-2 downto 0) <= producttemp((N+P-2)  downto P );
17      product(N-1) <= producttemp(N+N-1); --omejimo predznak pred overflowi 32 bitov
18  end Behavioral;
```

Slika 20: Koda (vhd) 7, množenje s fiksno decimalno vejico

Tudi koeficiente sem predstavil na isti način, za kar sem spisal C funkcijo pretvorba_f_int. Poleg osnovnih koeficientov sem tudi ostale tabele združil v 4x8 tabelo 32 bitnih vrednosti. Spodnjih 6 mest zaseda tabela koeficientov kot prej, na [6] mesto sem dodal na spodnjih 8 bitov mejo na naslednjih 8 barve, [7] mesto pa je namenjeno preslikavi:

```
176  void pretvorba_f_int(float fin[][6], int fout[][8], int meje[], int preslikava[], int barve[])
177  {
178
179      int row = 4;
180      int col = 6;
181      int meja8 = 0;
182
183      for (int i = 0; i < row; i++) {
184          for (int j = 0; j < col; j++) {
185              int x = (int)(fin[i][j] * (1 << S));
186              //printf("%d", x);
187              fout[i][j] = x;
188          }
189      }
190
191      for (int i = 0; i < row; i++) {
192          meja8 = (255*meje[i])/meje[row-1];
193          fout[i][6] = (barve[i]<<8 | meja8);
194          fout[i][7] = preslikava[i]<<S;
195      }
196
197  }
```

Slika 21: Koda (c) 4, pretvorba koeficientov

Podobna struktura je tudi v VHD, ki je že inicializirana na vrednost osnovne praproti (podobno kodo na sliki 21 sem pognal na računalniku, da izpiše binarno predstavitev števil), ima pa dodatno mesto [4][0], kjer se nahaja na LSB vrednost **smooth**.

Koeficiente sem prenesel iz računalnika na podoben način kot sem v prejšnjih izvedbah pošiljal modulu vga podatke o pikslih (naslov in vrednost):

```
30   static inline void coeff(int x, int y, int c)
31   {//pisi piksel x y
32       int one = 0b00000001;
33       int xy = ((x*16)+(y)); //doloci adr blok rama iz x in y
34       Xil_Out8(XPAR_AXI_SET4_0_S00_AXI_BASEADDR+12, xy); //na prav naslov da adr
35       Xil_Out8(XPAR_AXI_SET4_0_S00_AXI_BASEADDR+8, one); //wr1
36       Xil_Out32(XPAR_AXI_SET4_0_S00_AXI_BASEADDR, c);
37       Xil_Out8(XPAR_AXI_SET4_0_S00_AXI_BASEADDR+8, 0); //wr0
38   }
196  void coeff_upload(int fin[][8])
197  {
198      int row = 4;
199      int col = 8;
200      for (int i = 0; i < row; i++) {
201          for (int j = 0; j < col; j++) {
202              coeff(j,i,fin[i][j]);
203          }
204      }
205  }
40   static inline void smooth(int s)
41   {
42       int one = 0b00000001;
43       Xil_Out8(XPAR_AXI_SET4_0_S00_AXI_BASEADDR+12, 4); //na prav naslov da s
44       Xil_Out8(XPAR_AXI_SET4_0_S00_AXI_BASEADDR+8, one); //wr1
45       if(s==1)
46       {
47           Xil_Out32(XPAR_AXI_SET4_0_S00_AXI_BASEADDR, one);
48       }
49       else
50       {
51           Xil_Out32(XPAR_AXI_SET4_0_S00_AXI_BASEADDR, 0);
52       }
53       Xil_Out8(XPAR_AXI_SET4_0_S00_AXI_BASEADDR+8, 0); //wr0
54   }
```

Slika 22: Koda (c) 5, posiljanje koeficientov

Smooth sem dal v posebno funkcijo ker sprememba smooth ne resetira slike.

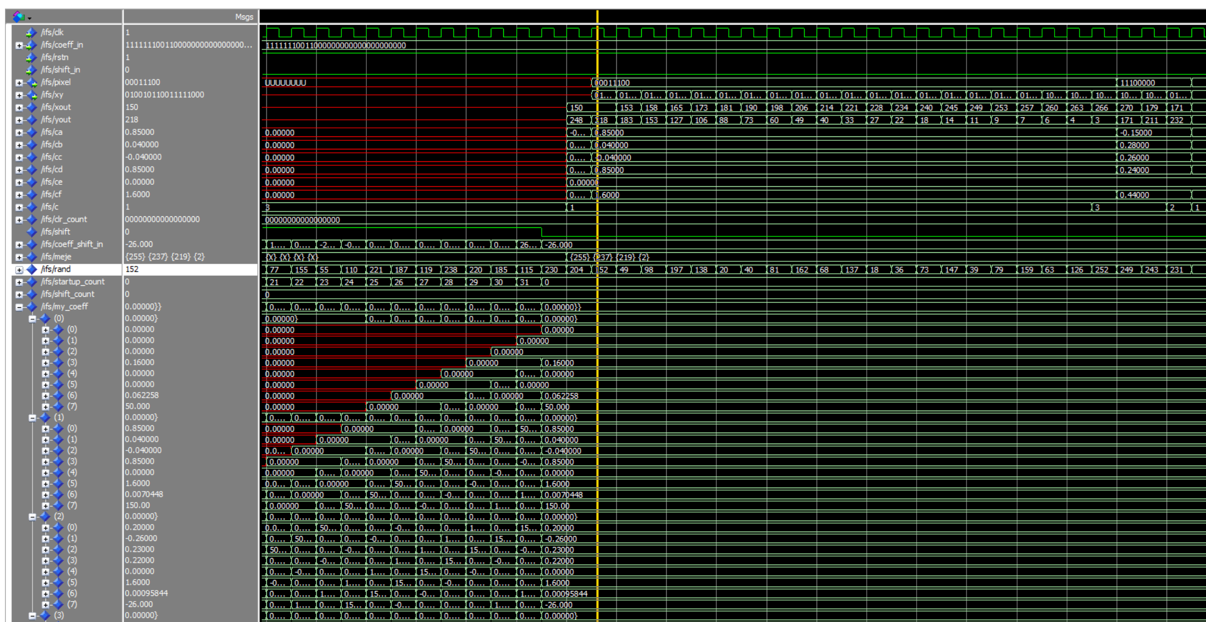
Še zadnja stvar pred združitvijo v modul IFS.vhd je naključno število. Tega sem se odločil generirati s LFSR (linear-feedback shift register), s pipami (mesta iz kjer gremo na xor za prvi bit) vzetimi iz interneta. Ideja delovanja je da imamo pomikalni register, katerega prvi bit je funkcija odvisna od nekaterih ostalih bitov, tako sicer vedno dobimo periodično funkcijo, vendar je perioda lahko zelo dolga (ne popolnoma naključna, ampak psevdo-naključna). Za povečanje naključnosti imam 4 takšne generatorje z različnimi semeni (seed, ki spremeni začetno mesto zaporedja), ki pa jih seštejemo. Mejo je bilo seveda potrebno ustrezno zamakniti iz števil do 100 na števila do 2^16. (komentar: v imenu sem dal Galois, v resnici je bolj podobno v trenutni izvedbi Fibonacci lfsr, Galois ima pipe v smeri zamikanja):

```vhdl
17  process (clk, rstn)
18  begin
19      if (rstn = '0') then
20          reg <= std_logic_vector(to_unsigned(seed,32));
21      elsif rising_edge(clk) then
22          reg(0) <=  reg(15) xor reg(13) xor reg(12) xor reg(10) xor '1';
23          reg(31 downto 1) <= reg(30 downto 0);
24      end if;
25  end process;
```

Slika 23: Koda (vhd) 8, lfsr

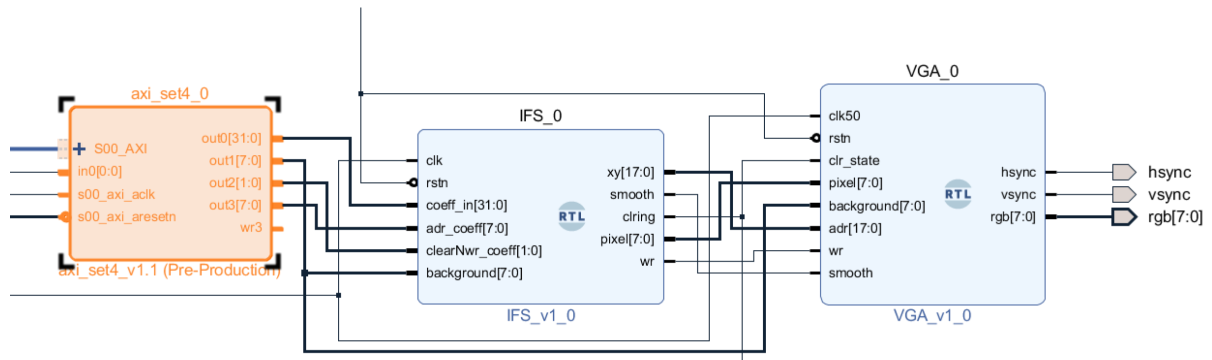
Dodal sem še števec za stanja counter_fsm.vhd, ki v primeru reseta z **rstn**, spremembe koeficientov razen smooth **wrEN**, ali force clear stanja direktno iz vodila axi, gre v stanje **clr_state**, v katerem popiše celoten spomin (preleti vse veljavne naslove x in y) z ozadjem.

Za preverjanje delovanja sem uporabil ModelSIM, tu je vidna še starejša različica v kateri sem pomikal koeficiente s shift registrom, in 8-bitnim lfsr, delovanje pa je v glavnem enako – najprej se nastavijo koeficienti vsi kot je potrebno, se naložijo meje in barve, x in y na 0, ko pa nastavitev konča pa se vsak cikel na izhodu pokaže naslov x in y (točne slike nisem preveri, vendar vrednosti zgledajo ustrezne glede območje in barvo na pravi sliki), ter barva piksla:



Slika 24: test IFS modula

Za izvedbo sem popravil dolžino besed na axi_set4 in povezal modul IFS ter VGA kot je vidno na spodnji sliki (modul procesorskega sistem reseta, axi interkonekta in ZYNQ7 procesorskega sistema izpuščen za večjo čitljivost, saj je enako povezan kot prej). Out0 je koeficient, out3 je naslov (4 bite x 4 y), out2 pa **clear** na bit(1) in wr_**coeff** na bit(0), out1 nastavi ozadje.



Slika 25: del blokovne sheme IFS in VGA

Žal prikaz ni tak kot bi si želel:

Slika 26: test IFS na zaslonu, praprot

VIdimo da imamo manjkajoče piksle, kljub temu da še vedno računamo nove. Če spremenimo našo generacijo naključnih števil vidimo da se nam vzorec praproti nekoliko spremeni, torej sumim periodo našega LFSR kar se tiče pomanjkljivosti izrisa. Drug problem se zgodi pri nalaganju koeficientov:

Slika 27: test IFS na zaslonu, posodobitev koeficientov

Koeficienti se pravilno naložijo in dobimo praprot nižje in bolj zakrivljeno od inicializirane, vendar se nam stara ne zbriše pravilno. Razloga za tem nisem uspel odkriti, saj se v simulaciji brisanje pravilno izvede (za namen tega sem naredil še pomožen števec v vga.vhd da bi se zares zbrisalo).

V takem stanju sem projekt tudi končal, to nadgradnjo pa pustil bolj kot idejo kako bi se izvedlo, če bi bilo potrebno vse implementirati na FPGA.

Priložene datoteke:

Simulacijske datoteke so vsebovane skupaj z modelsim projektom v mapi oddmodelsim.

Vivado projekti za glavno izvedbo so v mapi oddaja3, oddaja5 pa ima projekt z IFS modulom (zadnja izvedba, vendar ne deluje še najbolje).

Ker je bilo dostopanje do razvojne ploščice zaradi problemov z gonilniki iz drugega računalnika sem dodal C kodo posebej v datoteki ckoda_program2b.c (tu je potrebno preimenovati željen main() del nazaj v main() za poganjanje).

Bit-stream za glavno izvedbo je v datoteki "vga_4bank_C_smoothcontrol.bit", za izvedbo z vgrajenim IFS modulom pa "vga_4bank_IFS_axi_extraclr.bit".

V primeru nadeljevanja projekta oziroma uporabe vga modula za projekte naslednjih let so potrebni moduli vga.vhd, color_smooth.vhd, adder_four.vhd in adder.vhd.