«به نام او»

گزارش پروژه پایانی درس مدارهای منطقی برنامه پذیر

استاد شریفیان

مهندس غضنفری

مهدی قصاب ۹۷۲۳۵۰۵

علیرضا علیزاده ۹۵۲۳۰۸۷

فهرست

بخش اول : توضیح کد متلب

- گام اول : تنظیمات اولیه هر کد متلب

```
clc;
clear
close all
```

- گام دوم : تعریف فیلترها

```
average_filter = [ 1/9,1/9,1/9;
                   1/9,1/9,1/9;
                   1/9,1/9,1/9; ];

edge_filter = [  0,-1, 0;
                -1, 4,-1;
                 0,-1, 0; ];
```

- گام سوم : دریافت عکس مورد نظر به صورت آرایه و تبدیل آن به آرایه های دو بعدی R و G و B

```
img = imread('image.jpg');

R_img = img(:,:,1);
G_img = img(:,:,2);
B_img = img(:,:,3);
```

تصویر 1 تصویر اصلی

- گام چهارم : اعملال هر دو فیلتر average و edge به سه آرایه دو بعدی قسمت قبل

```
average_R = uint8(filter2(average_filter,R_img));
```

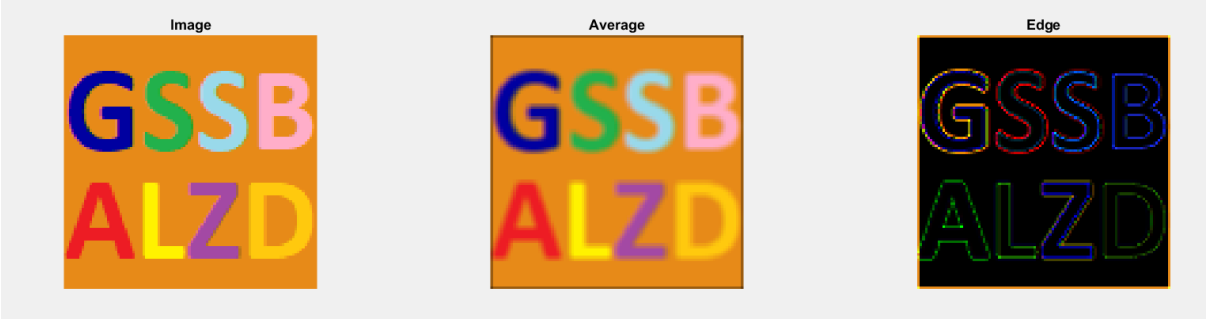
```
edge_R = uint8(filter2(edge_filter,R_img));

average_G = uint8(filter2(average_filter,G_img));
edge_G = uint8(filter2(edge_filter,G_img));

average_B = uint8(filter2(average_filter,B_img));
edge_B = uint8(filter2(edge_filter,B_img));
```

- گام پنجم : تلفیق نتایج قسمت قبل به یک آرایه سه بعدی

```
average_img = cat(3, average_R, average_G,average_B);
edge_img = cat(3, edge_R, edge_G,edge_B);
```

- گام ششم : چاپ تصاویر نهایی شامل تصویر اصلی، تصویر خروجی فیلتر average و تصویر خروجی فیلتر edge

```
subplot(2,3,1);
imshow(img);
title('Image')
subplot(2,3,2);
imshow(average_img);
title('Average')
subplot(2,3,3);
imshow(edge_img);
title('Edge')
```



- گام هفتم : ذخیره مقادیر R و G و B هر پیکسل

```
dlmwrite('R_img.txt',R_img);
dlmwrite('G_img.txt',G_img);
dlmwrite('B_img.txt',B_img);
```

- گام هشتم : اجرای همین اعمال روی بخش کوچکی از تصویر

از آنجایی که تصویر اصلی بسیار برای اجرا بزرگ است بخشی از تصویر را که ۷۰ پیکسل است (۱۴*۵) را هم به همین کد داده تا بعدا با خروجی مشاهده شده از مدار FPGA آن را مقایسه نماییم. تصاویر این بخش در انتهای گزارش کار آمده است.

بخش دوم : توضیح ساختار FPGA

برای هر کدام از فیلتر ها یک entity جداگانه به نام های average و edge تعریف نمودیم. سپس ورودی های ماتریس تصویر را در فایل testbench به این دو component داده و خروجی ها را تحویل میگیریم.

بخش سوم : توضیح کد average.vhd

ورودی ها و خروجی های کامپوننت

- rst : تک بیت برای ریست کردن سیستم
- clk : کلاک سیستم
- start : تک بیت برای شروع به کار سیستم
- input : مقدار عددی بین ۰ تا ۲۵۵ برای هر پیکسل (R و G و B را جداگانه به این ورودی میدهیم. اول R برای همه پیکسل ها، دوم G برای همه پیکسل ها، سوم B برای همه پیکسل ها) به دلیل ماکزیمم ۲۵۵ این ورودی ۸ بیتی است. فرض میکنیم قرار است با هر کلاک مقدار هر پیکسل وارد این ورودی شود. پس باید با هر کلاک یک ورودی بگیریم تا هیچ مقدار ورودی از دست نرود.
- row : عکس مورد نظر چند سطر پیکسل دارد
- column : عکس مورد نظر چند ستون پیکسل دارد
- output : مقدار خروجی به ازای ورودی های وارد شده تا آن لحظه

State ها

این ماشین فقط سه state دارد.

۱. idle : در این state هیچ کاری انجام نمی شود و منتظر فرمان start از بیرون هستیم. اگر این فرمان بیاید به state بعدی می رویم.

۲. start_rows : در این حالت فقط سه سطر اول خوانده می شوند و سپس سیستم به حالت بعدی می رود. این به خاطر ابعاد فیلتر ما می باشد که برای تولید خروجی نیازمند حداقل سه سطر می باشد.

۳. proces : در این حالت هم ورودی های بعدی برای سطر های بعدی دریافت می شود و هم خروجی همزمان تولید می شود.

Ram ها

از آنجایی که سایز ورودی و خروجی میتواند متغیر باشد ما یک RAM هزار خانه ای که هر خانه آن ۸ بیت است در نظر گرفته ایم.

۱. in_ram : داده های ورودی در این رم ریخته می شود.

۲. out_avg_ram : داده های خروجی در این RAM ریخته می شود.

Register ها و signal ها

state_reg , state_next : حالت فعلی و بعدی(بعد از کلاک مورد نظر) سیستم در آن ها ذخیره می شوند.

data_out_1 to 9 : فیلتر ما ۹ درایه دارد که باید در درایه های یک بخش ۳*۳ از هر ماتریس ضرب شوند و مجموع آن ها بشود خروجی فیلتر مورد نظر. از آنجایی که مقدار تمامی درایه های فیلتر average برابر $\frac{1}{9}$ می باشد، پس یا باید مقادیر هر درایه متناسب با آن بخش را تقسیم بر ۹ کنیم سپس با هم همه را جمع کنیم(که چون مقادیر اعشار را ذخیره نمیکنیم این روش خطای زیادی دارد) و یا اینکه همه را با هم جمع نماییم سپس مجموع را بر ۹ تقسیم کنیم. از آنجایی که روش اول خطای زیادی دارد روش دوم را انتخاب کردیم. ولی در روش دوم چون مجموع ها بیشتر از ۲۵۵ می شود لذا این سیگنال ها را ۱۶ بیتی در نظر گرفتیم.

data_in : این سیگنال مستقیما به ورودی وصل است و عملیات ذخیره سازی ورودی در رم ورودی با استفاده از این سیگنال صورت می گیرد.

data_outram : سیگنالی که مجموع سیگنال های قبلی تقسیم بر ۹ برای هر خروجی در آن ریخته می شود و در انتهای کد ۸ بیت انتهایی آن را به خروجی (out put) متصل نمودیم.

row_cnt_reg , row_cnt_next : نشان میدهد تا الان چند سطر از تصویر اصلی را دریافت کرده ایم.

counter_reg , counter_next : نشان میدهد بعدی شماره ستون مرکزی ماتریس ۳*۳ از پیکسل های تصویر ورودی که قرار است در فیلتر ضرب شود کدام است.

row_up_reg , row_up_next : نشان میدهد بعدی ردیف بالای ماتریس ۳*۳ از پیکسل های تصویر ورودی که قرار است در فیلتر ضرب شود کدام است.

row_center_reg , row_center_next : نشان میدهد ردیف وسط ماتریس ۳*۳ از پیکسل های تصویر ورودی که قرار است در فیلتر ضرب شود کدام است.

row_down_reg , row_down_next : نشان میدهد ردیف پایین ماتریس ۳*۳ از پیکسل های تصویر ورودی که قرار است در فیلتر ضرب شود کدام است.

add_inram_reg , add_inram_next : نشان میدهد ورودی باید در چه آدرسی از رم ورودی ذخیره شود.

add_outram_reg , add_outram_next : نشان میدهند خروجی باید در چه آدرسی از رم خروجی ذخیره شود.

Constant ها

nine : عدد ۹ که قرار است مجموع سیگنال های data_out_1 to 9 بر آن تقسیم شوند و حاصل آن در سیگنال data_outram ذخیره شود.

Process مربوط به رجیسترها و حافظه ها

در این process ۴ کار انجام می شود.

۱. اگر سیگنال ریست آمده باشد مقادیر همه رجیستر ها initial می شود.

```
if (rst = '1')  then
      state_reg <=  idle;
      row_cnt_reg <= (others => '0');
      add_inram_reg <= (others => '0');
      add_outram_reg <= (others => '0');
      counter_reg <= ("00000001");
      row_up_reg <= (others => '0');
      row_center_reg <= (others => '0');
      row_down_reg <= (others => '0');
```

۲. با هر کلاک مقادیر رجیستر به روزرسانی می شوند.

```
  state_reg <=  state_next;
      row_cnt_reg <= row_cnt_next;
      add_inram_reg <= add_inram_next;
      add_outram_reg <= add_outram_next;
      counter_reg <= counter_next;

      row_up_reg <= row_up_next;
      row_center_reg <= row_center_next;
      row_down_reg <= row_down_next;
```

۳. با هر کلاک مقدار data_in درون in_ram و data_outram درون out_average_ram در آدرس های add_inram_reg و add_outram_reg ذخیره می شوند.

```
      in_ram(to_integer(unsigned(add_inram_reg))) <=
data_in;
```

```
out_avg_ram(to_integer(unsigned(add_outram_reg))) <=
data_outram(7 downto 0);
```

۴. با هر کلاک مقادیر سیگنال های data_out_1 to 9 که از رم ورودی با توجه به رجیستر های counter استخراج شده و با هشت بیت صفر concatenate می شود.

```
        data_out_1 <= "00000000" &
in_ram(to_integer((unsigned(row_up_reg)*unsigned(colu
mn))+unsigned(counter_reg)-1));

        data_out_2 <= "00000000" &
in_ram(to_integer((unsigned(row_up_reg)*unsigned(colu
mn))+unsigned(counter_reg)));

        data_out_3 <= "00000000" &
in_ram(to_integer((unsigned(row_up_reg)*unsigned(colu
mn))+unsigned(counter_reg)+1));


        data_out_4 <= "00000000" &
in_ram(to_integer((unsigned(row_center_reg)*unsigned(
column))+unsigned(counter_reg)-1));

        data_out_5 <= "00000000" &
in_ram(to_integer((unsigned(row_center_reg)*unsigned(
column))+unsigned(counter_reg))); --center house

        data_out_6 <= "00000000" &
in_ram(to_integer((unsigned(row_center_reg)*unsigned(
column))+unsigned(counter_reg)+1));


        data_out_7 <= "00000000" &
in_ram(to_integer((unsigned(row_down_reg)*unsigned(co
lumn))+unsigned(counter_reg)-1));

        data_out_8 <= "00000000" &
in_ram(to_integer((unsigned(row_down_reg)*unsigned(co
lumn))+unsigned(counter_reg)));

        data_out_9 <= "00000000" &
in_ram(to_integer((unsigned(row_down_reg)*unsigned(co
lumn))+unsigned(counter_reg)+1));
```

Process مربوط به عملیات های هر state

در این process با استفاده از case-when تعیین شده که در هر state چه اتفاقی بیفتد.

۱. idle : در این حالت باید منتظر این بماند تا ورودی start یک شود تا رجیستر ها را مقدار دهی کند و به حالت بعدی برود.

```
if (start = '1') then
    state_next     <= start_rows;
    row_cnt_next   <= (others => '0');
    add_inram_next <= (others => '0');
    add_outram_next <= (others => '0');
end if;
```

۲. start_rows : در این حالت ابتدا i nput ها را به data_in اطلاق می دهد تا در in_ram ذخیره شود و آدرس ذخیره سازی بعدی در in_ram را یکی افزایش میدهد .

```
data_in <= input;
add_inram_next <=
std_logic_vector(unsigned(add_inram_reg) + 1);
```

تا مقدار add_inram_reg به مقدار ۳ برابر تعداد ستون ها منهای یکی برسد. یعنی قرار باشد آخرین پیکسل سطر سوم به ورودی بیاید. در این صورت ابتدا حالت بعدی را به proces تغییر داده و مقادیر بعدی رجیسترهای counter را مقداردهی می کند و همان کار قبلی(ذخیره سازی در رم ورودی) را برای درایه آخر سطر سوم تکرار می کند.

```
state_next <= proces;
row_cnt_next    <= "00000100";
counter_next    <= "00000001";
add_outram_next <= (others => '0');
row_up_next     <= "00000000";
row_center_next <= "00000001";
row_down_next   <= "00000010";
data_in <= input;
```

```
                    add_inram_next <=
std_logic_vector(unsigned(add_inram_reg) + 1);
```

۳. proces : اگر row_cnt_reg به ۲ تا بیشتر از تعداد سطر ها برسد یعنی تمام خروجی ها محاسبه شده و تمام ورودی ها نیز وارد شده اند پس دیگر نیازی نیست در این حالت بمانیم و باید به حالت idle بر گردیم.

```
state_next <= idle;
```

در غیر این صورت اگر counter_reg به یکی کمتر از تعداد ستون ها باشد(یعنی هنوز فیلتر لغزنده به پوشش دادن آخرین ستون نرسیده باشد) باید ابتدا یکی مقدار بعدی این counter را افزایش داده و data_outram را برابر مجموع data_out_1 to 9 ها تقسیم بر ۹ محاسبه نماییم و در انتها مکان ذخیره سازی در out_avg_ram نیز یکی افزایش دهیم.

```
                if (counter_reg <
std_logic_vector(unsigned(column)-1)) then
                        counter_next <=
std_logic_vector(unsigned(counter_reg) + 1);
std_logic_vector(((unsigned(data_out_1)+unsigned(data
_out_2)+unsigned(data_out_3)+unsigned(data_out_4)+uns
igned(data_out_5)+unsigned(data_out_6))+unsigned(data
_out_7)+unsigned(data_out_8)+unsigned(data_out_9))/un
signed(nine));
                        add_outram_next <=
std_logic_vector(unsigned(add_outram_reg) + 1);
                end if;
```

و اگر counter_reg به دوتا کمتر از تعداد ستون ها رسیده باشد(یعنی فقط آخرین ستون پیکسل ها هنوز توسط فیلتر لغزنده پوشش داده نشده باشد) مقدار بعدی این counter را دوباره به ۱ تغییر داده و مقادیر counter های ردیف ها را یکی افزایش می دهیم.

```
                    counter_next <= "00000001";


                    row_down_next   <=
std_logic_vector(unsigned(row_down_reg) + 1);
                    row_center_next <=
std_logic_vector(unsigned(row_center_reg) + 1);
```

```
                row_up_next     <=
std_logic_vector(unsigned(row_up_reg) + 1);
                row_cnt_next <=
std_logic_vector(unsigned(row_cnt_reg) + 1);
```

با توجه با این که حجم تصویر ورودی ممکن است بسیار زیاد باشد ما در آن واحد فقط ۴ سطر از پیکسل های تصویر ورودی را درون in_ram ذخیره میکنیم. پس باید counter های ردیف برای محاسبه خروجی به صورت circular بین این چهار سطر جابجا شوند.

```
        if(row_down_reg = "00000011") then
                    row_down_next <= "00000000";
                elsif(row_center_reg = "00000011")
then
                    row_down_next <= "00000000";
                elsif(row_up_reg = "00000011") then
                    row_up_next <= "00000000";
                end if;
```

و هر گاه مقدار محل ذخیره سازی در in_ram به ۴ برابر تعداد ستون های تصویر منهای یک رسید(به یکی مانده به آخرین درایه چهارمین سطر از in_ram رسیدیم)، دوباره مقدار add_inram_next را صفر نماییم که از سطر اول in_ram پر شود.

```
                if(add_inram_reg =
std_logic_vector((unsigned(column)+unsigned(column)+u
nsigned(column)+unsigned(column))-1)) then
                data_in <= input;
                add_inram_next <= (others => '0');
```

و در غیر اینصورت هر دفعه یکی به مقدار add_inram_next اضافه نموده و مقدار input را به data_in اطلاق دهیم.

```
                data_in <= input;
                add_inram_next <=
std_logic_vector(unsigned(add_inram_reg) + 1);
```

بخش چهارم : توضیح کد edge.vhd

این کد کاملا شبیه کد قبلی میباشد با چند تفاوت :

۱. اسم ram خروجی out_edge_ram میباشد.

۲. data_out_1 to 9 ها ۸ بیتی میباشند. این دفعه از ترفند دیگری برای جلوگیری از overflow استفاده نموده ایم. به این صورت که متناسب با این که مقدار پیکسل مورد نظر برای محاسبه خروجی کجای ماتریس ۳*۳ است، مقدار ثابت one (عدد یک ۸ بیتی) یا four (عدد چهار ۸ بیتی) را در آن ها ضرب کرده ایم و مجموع آن قسمت هایی را که در ۱ ضرب می شوند در temp1 که یک سیگنال ۱۶ بیتی است ذخیره میکنیم (میدانیم اگر ورودی های ضرب کننده، n بیت باشد، خروجی آن 2n بیت میشود) و خروجی قسمتی که در ۴ ضرب میشود را در temp4 که یک سیگنال ۱۶ بیتی است ذخیره میکنیم.

۳. در نهایت اگر temp4 بزرگتر از temp1 بود ، data_outram برابر با temp4 منهای temp1 می شود؛ در غیر اینصورت چون مقدار پیکسل باید بین ۰ تا ۲۵۵ باشد data_outram را برابر صفر قرار می دهیم.

بخش پنجم : توضیح کد average_tb.vhd و خروجی آن

تعریف کامپوننت ها و عملیات portmap

```
avrage:     entity  work.average(average_beh)
              Port map(
                    clk    => clk,
                    rst    => rst,
                    start  => start,
                    input  => input,
                    row    => row,
                    column => column,
                    output => output_avrg
              );

   edge:      entity  work.edge(edge_beh)
              Port map(
                    clk    => clk,
                    rst    => rst,
                    start  => start,
                    input  => input,
                    row    => row,
                    column => column,
                    output => output_edge
              );
```

ساخت کلاک

```
clk_process :process
 begin
 clk <= '0';
```

```
wait for T/2;
clk <= '1';
wait for T/2;
end process;
```

دادن پیکسل های به صورت متوالی به ورودی کامپوننت ها و دریافت خروجی ها

```
wait for  T;
  input <= std_logic_vector(to_unsigned( مقدار عدد مورد نظر بین ۰ تا ۲۵۵ ,8));
```

به توالی بالا داده های جدول زیر را به عنوان مقادیر Red به input دادیم.

231	231	231	52	0	0	0	0	0	52	231	231	231	231
231	231	198	0	0	0	0	0	0	165	231	231	231	231
231	231	129	0	0	0	0	0	52	231	231	231	231	231
231	231	92	0	0	0	0	0	92	231	231	231	231	231
231	231	52	0	0	0	0	0	165	231	231	231	231	231

خروجی این داده ها در عکس های ذخیره شده در پوشه result به نام های R_Average.PNG و R_Edge.PNG آمده است.

داده های جدول زیر را به عنوان مقادیر Green به input دادیم.

138	138	119	0	0	0	0	0	0	77	138	138	138	138
138	138	77	0	0	0	0	0	0	138	138	138	138	138
138	138	31	0	0	0	0	0	77	138	138	138	138	138
138	138	0	0	0	0	0	0	98	138	138	138	138	138
138	119	0	0	0	0	0	0	138	138	138	138	138	138

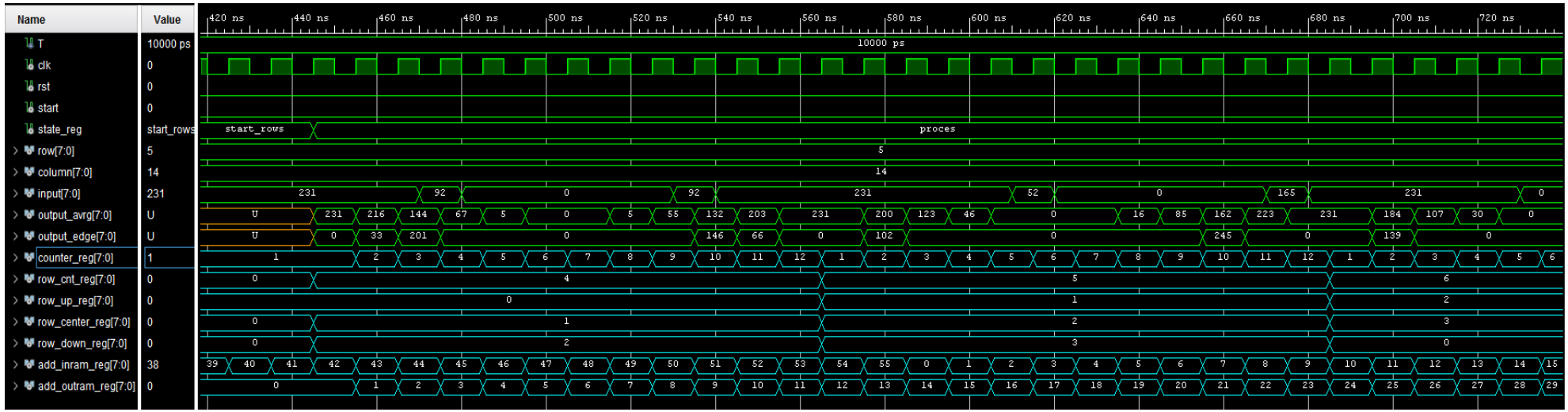
خروجی این داده ها در عکس های ذخیره شده در result پروژه به نام های G_Average.PNG و G_Edge.PNG آمده است.

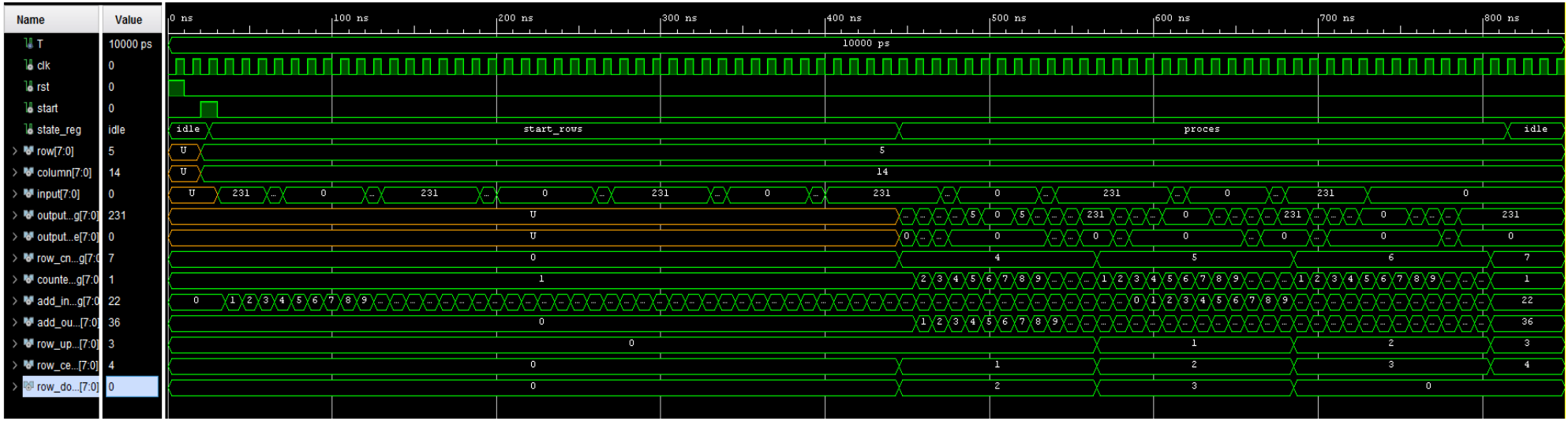
به توالی بالا داده های جدول زیر را به عنوان مقادیر Blue به input دادیم.

24	24	98	160	160	160	160	160	160	51	24	24	24	24
24	24	140	160	160	160	160	160	119	24	24	24	24	24
24	51	160	160	160	160	160	160	51	24	24	24	24	24
24	75	160	160	160	160	160	160	24	24	24	24	24	24
24	98	160	160	160	160	160	119	24	24	24	24	24	24

خروجی این داده ها در عکس های ذخیره شده در result پروژه به نام های B_Average.PNG و B_Edge.PNG آمده است.

تغییرات سیگنال های خروجی و سیگنال های میانی





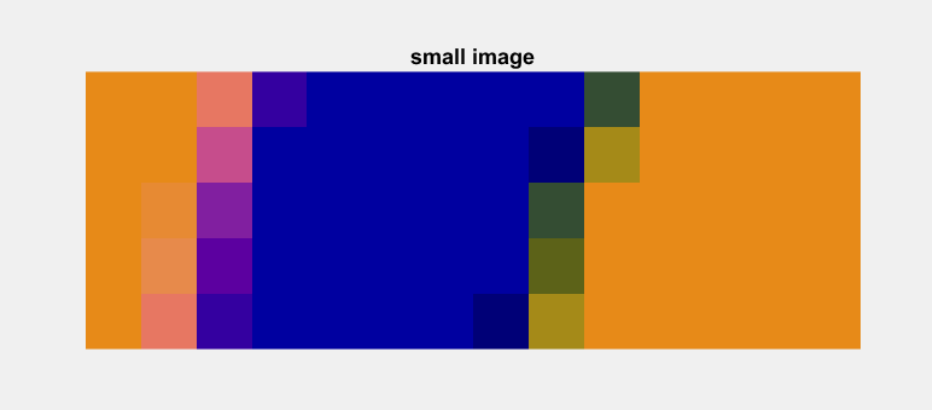
بخش ششم : مقایسه خروجی متلب و FPGA

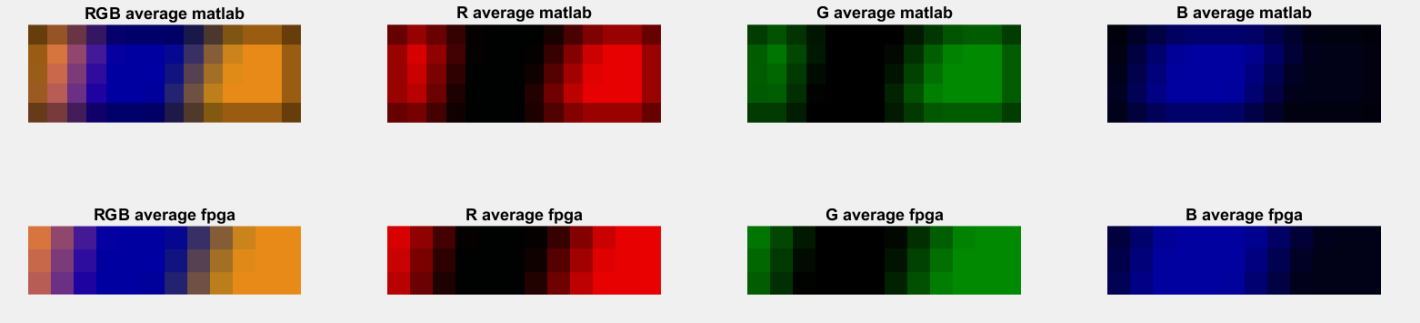
چند تفاوت در خروجی های متلب و FPGA وجود دارد:

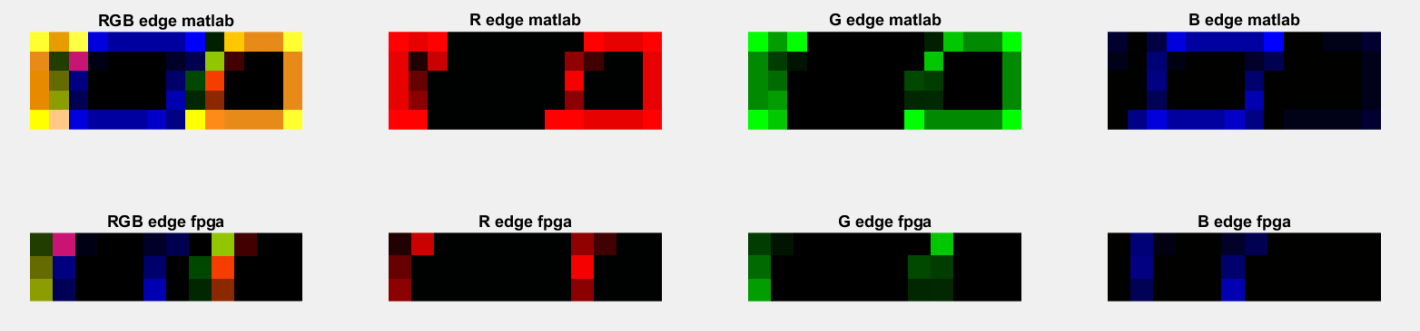
۱. تابع استفاده شده در متلب خودش به تصویر padding اضافه می نماید. اما ما این کار را در FPGA انجام ندادیم و درنتیجه ابعاد خروجی ما متفاوت است.

۲. تابع استفاده شده در متلب اگر مقدار عدد مورد نظر اعشاری شود به سمت بالا گرد می کند. در صورتی که ما به سمت پایین اعداد اعشاری را گرد میکنیم. لذا نهایت اختلاف بین اعداد ما و متلب فقط در فیلتر average است که آن هم برخی اعداد ۱ واحد با خروجی متلب تفاوت دارند.

در نهایت برای مقایسه خروجی کد متلب و FPGA یک فایل به نام compare.m نوشتیم که در زیر تصویر هر دو خروجی را برای یک بخش ۷۰ پیکسلی از تصویر مشاهده می نمایید.(تمامی خروجی های متلب و FPGA در پوشه ی result/fpga_compare با فرمت txt ذخیره شده اند.)







R_average_fpga.txt

```
216,144,67,5,0,0,5,55,132,203,231,231
200,123,46,0,0,0,16,85,162,223,231,231
184,107,30,0,0,0,34,111,188,231,231,231
```

R_average_matlab.txt

```
103,150,105,53,6,0,0,0,24,75,127,154,154,103
154,216,145,68,6,0,0,6,56,133,204,231,231,154
154,201,124,47,0,0,0,16,86,163,224,231,231,154
154,184,107,30,0,0,0,34,111,188,231,231,231,154
103,119,67,16,0,0,0,29,80,131,154,154,154,103
```

R_edge_fpga.txt

```
33,201,0,0,0,0,0,0,146,66,0,0
102,0,0,0,0,0,0,0,245,0,0,0
139,0,0,0,0,0,0,0,139,0,0,0
```

R_edge_matlab.txt

```
255,231,255,0,0,0,0,0,0,0,255,231,231,255
231,33,201,0,0,0,0,0,0,146,66,0,0,231
231,102,0,0,0,0,0,0,0,245,0,0,0,231
231,139,0,0,0,0,0,0,0,139,0,0,0,231
255,255,0,0,0,0,0,0,255,255,231,231,231,255
```

G_average_fpga.txt

```
117,71,25,0,0,0,8,47,93,131,138,138
104,58,12,0,0,0,19,65,111,138,138,138
93,47,3,0,0,0,34,80,126,138,138,138
```

G_average_matlab.txt

```
61,83,52,22,0,0,0,0,24,55,85,92,92,61
92,117,71,25,0,0,0,9,48,94,131,138,138,92
92,104,58,12,0,0,0,19,65,111,138,138,138,92
90,93,47,3,0,0,0,35,81,127,138,138,138,92
59,59,29,0,0,0,0,26,57,88,92,92,92,61
```

G_edge_fpga.txt

```
61,20,0,0,0,0,0,0,199,0,0,0
107,0,0,0,0,0,0,72,61,0,0,0
157,0,0,0,0,0,0,39,40,0,0,0
```

G_edge_matlab.txt

```
255,157,255,0,0,0,0,0,0,32,199,138,138,255
138,61,20,0,0,0,0,0,0,199,0,0,0,138
138,107,0,0,0,0,0,0,72,61,0,0,0,138
138,157,0,0,0,0,0,0,39,40,0,0,0,138
255,200,0,0,0,0,0,0,255,138,138,138,138,255
```

B_average_fpga.txt

```
1  63,108,150,160,160,160,143,101,55,27,24,24
2  75,121,157,160,160,160,128,82,37,24,24,24
3  86,131,160,160,160,155,113,67,27,24,24,24
```

B_average_matlab.txt

```
1  11,37,67,98,107,107,107,102,75,45,19,16,16,11
2  19,63,109,151,160,160,160,143,101,56,27,24,24,16
3  25,76,121,158,160,160,160,128,83,38,24,24,24,16
4  33,86,132,160,160,160,155,113,68,27,24,24,24,16
5  25,60,90,107,107,107,102,72,42,16,16,16,16,11
```

B_edge_fpga.txt

```
1  0,118,20,0,0,41,81,0,0,0,0,0
2  0,129,0,0,0,109,0,0,0,0,0,0
3  0,85,0,0,0,177,0,0,0,0,0,0
```

B_edge_matlab.txt

```
1  48,0,68,222,160,160,160,160,255,0,0,24,24,48
2  24,0,118,20,0,0,0,41,81,0,0,0,0,24
3  0,0,129,0,0,0,0,109,0,0,0,0,0,24
4  0,0,85,0,0,0,0,177,0,0,0,0,0,24
5  0,133,222,160,160,160,201,132,0,24,24,24,24,48
```