ELEC 4440 Term Project Report

Color Space Conversion

HAN Haiyang

20025666

December 16, 2014

1. Project Description

This project utilizes the tools of MATLAB, Simulink System Generator, and Xilinx to implement an image color space converter on an FPGA device.

2. Objective

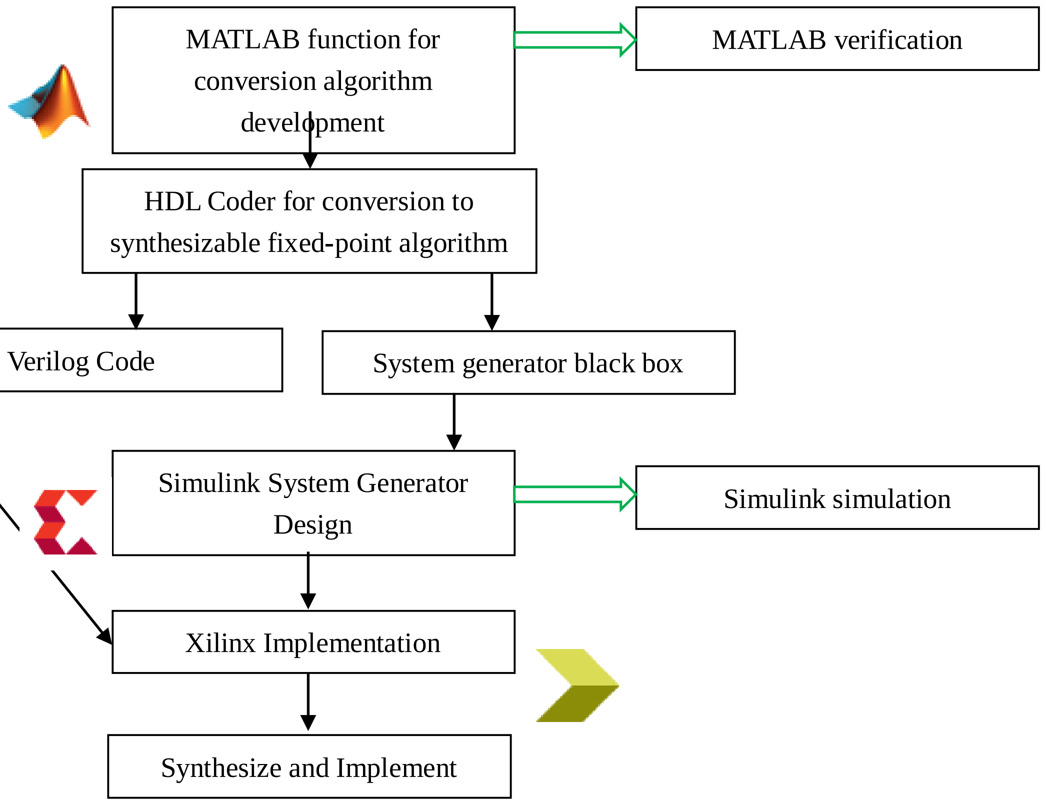
The objective of the color space converter is to convert an image of arbitrary size in RGB color space to either HSV, XYZ, or YIQ color space and then back to RGB again. The user is able to choose which one of the three color spaces the result image should be in. The original RGB image, the converted image, as well as the HSV/XYZ/YIQ to RGB image should all be shown.

3. Project Specification

1) Input: the project model takes in one RGB image in PNG format as well as a two-bit integer ranging from 0 to 2 to specify which color space conversion process the user would like to perform.
2) Output: the project model has two outputs, the converted image_converted in either one of HSV, XYZ or YIQ color space in PNG format, as well as the image image_back_to_RGB that is converted back to RGB from image_converted.
3) For MATLAB verification and Simulink simulation, the output images are written to file in PNG format.

4. System Level Diagram and Design Methodology

The diagram below shows the system level design methodology of the project.


MATLAB function for conversion algorithm development
MATLAB verification
HDL Coder for conversion to synthesizable fixed-point algorithm
Verilog Code
System generator black box
Simulink System Generator Design
Simulink simulation
Xilinx Implementation
Synthesize and Implement

5. Implementation

1) Algorithms

The algorithms for the conversion process were implemented as MATLAB function files. All of them process the input images pixel by pixel by taking in three values in uint8 format, and producing three outputs in uint8 format. The three values represent the three components that make up a particular pixel. Note that internally the pixels are cast to double before processing and cast back to uint8 after processing so that they can be properly displayed in MATLAB. All inputs and outputs are in uint8 format and are in the range from 0 to 255.

The details of the algorithms used are described in detail below.

● RGB to HSV

The R, G, B values are divided by 255 to change the range from 0-255 to 0-1:

R' = R/255

G' = G/255

B' = B/255

Cmax = max(R', G', B')

Cmin = min(R', G', B')

Δ = Cmax – Cmin

Hue calculation:

$$H = \begin{cases} 60^\circ \times \left(\frac{G'-B'}{\Delta} mod 6\right) & , Cmax = R' \\ 60^\circ \times \left(\frac{B'-R'}{\Delta} + 2\right) & , Cmax = G' \\ 60^\circ \times \left(\frac{R'-G'}{\Delta} + 4\right) & , Cmax = B' \end{cases}$$

Saturation calculation:

$$S = \begin{cases} 0 & , C_{max} = 0 \\ \frac{\Delta}{C_{max}} & , C_{max} \neq 0 \end{cases}$$

Value calculation:

V = Cmax

Finally, the H value is divided by 360 and multiplied by 255, and the S and V values are multiplied by 255 so that it is possible to view the results with MATLAB or Windows Image Viewer.

● HSV to RGB

When $0 \leq H < 360$, $0 \leq S \leq 1$ and $0 \leq V \leq 1$:

C = V × S

X = C × (1 - |(H / 60°) mod 2 - 1|)

m = V – C

$$(R', G', B') = \begin{cases} (C, X, 0) & , 0^\circ \leq H < 60^\circ \\ (X, C, 0) & , 60^\circ \leq H < 120^\circ \\ (0, C, X) & , 120^\circ \leq H < 180^\circ \\ (0, X, C) & , 180^\circ \leq H < 240^\circ \\ (X, 0, C) & , 240^\circ \leq H < 300^\circ \\ (C, 0, X) & , 300^\circ \leq H < 360^\circ \end{cases}$$

(R, G, B) = (R' + m, G' + m, B' + m)

- RGB to YIQ

The Y, I, Q values are computed by multiplying a matrix to the R, G, B values.

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.2989 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.212 & -0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

- YIQ to RGB

The R, G, B values are computed by multiplying a matrix to the Y, I, Q values.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0.956 & 0.621 \\ 1 & -0.272 & -0.647 \\ 1 & -1.105 & 1.702 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

- RGB to XYZ

The X, Y, Z values are computed by multiplying a matrix to the R, G, B values.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.5767309 & 0.1855540 & 0.1881852 \\ 0.2973769 & 0.6273491 & 0.0752741 \\ 0.0270343 & 0.0706872 & 0.9911085 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

- XYZ to RGB

The R, G, B values are computed by multiplying a matrix to the X, Y, Z values.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 2.0413690 & -0.5649464 & -0.3446944 \\ -1 & 1.8760108 & 0.0415560 \\ 0.0134474 & -0.1183897 & 1.0154096 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

- Verification with MATLAB test bench

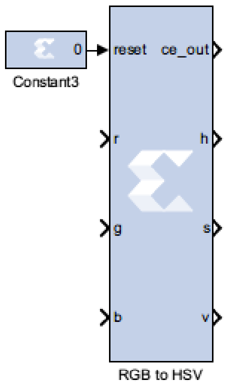
Test benches in MATLAB are written as MATLAB functions. The general process is to read and display image to be converted from file, store the values in a *width* x *height* x *3* array, run the corresponding conversion function pixel by pixel in a nested loop, save the converted values in a new 3D array, display the converted image, and finally write the converted image to file in PNG format.

2) Fix-point Conversion and Verilog Generation

For the functions to be implementable in FPGA, all floating operations must be transformed to fix-point functions, and all internal floating point variables must also be represented by fix-point data types. For Verilog module to be generated, all variable cannot have dynamic data types. That is, their format and size must be specified and are not allowed to change inside function. The basic steps to generating fix-point function, Verilog module, and the system generator black box using the MATLAB HDL Coder are as follows:

i. Specify MATLAB function file and test bench file
ii. Define the numeric types of input, output, and internal variables. The types for input and output should be set to uint8 format, i.e. numeric (0, 8, 0).
iii. To avoid overflow in calculating the values and casting them to fix-point unsigned 8-bit integer format, which may cause pixels to have unnatural colors and deviate from the values they should hold, the overflow action of the fimath function (used to convert floating point functions to fix-point functions) should be set to “Saturate”.
iv. Validate types and run the generated fix-point function to test whether numeric and data types have been set correctly as well as whether calculation errors have occurred.
v. Select code generation target. For this project, the device selected is Xilinx Virtex6 xc6vcx130t-ff1156-1.
vi. Generate Verilog HDL code and system generator black box. The clock enable input port should be set to “CE”, and the drive clock enable at box should be set to “DUT base rate”.

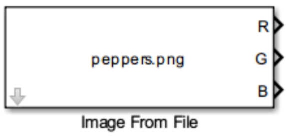
After this process, the Verilog codes as well as system generator black boxes for each of the six conversion processed have been generated. Each black box now has three data inputs, three data outputs, one reset port which is always set to false, and one ce_out output which is not used. Each black box is programmed to implement one of the six conversion algorithms.



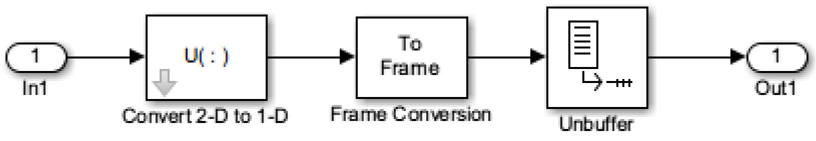
3) System Generator Design and System Block Diagram

i. Image Preprocessing

The goal is to process the input image so that the correct data types can be input into the Gateway In blocks as well as the black boxes. First the image is read using the Image From File block, which is set to output the R, G, B values separately.



Then each one of the R, G, B values are preprocessed using the following blocks and operations:

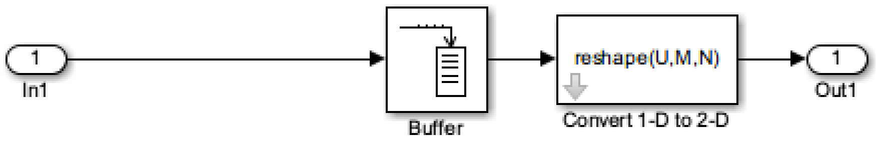


Now the image is ready to be converted.

ii. Image Conversion and Output Processing

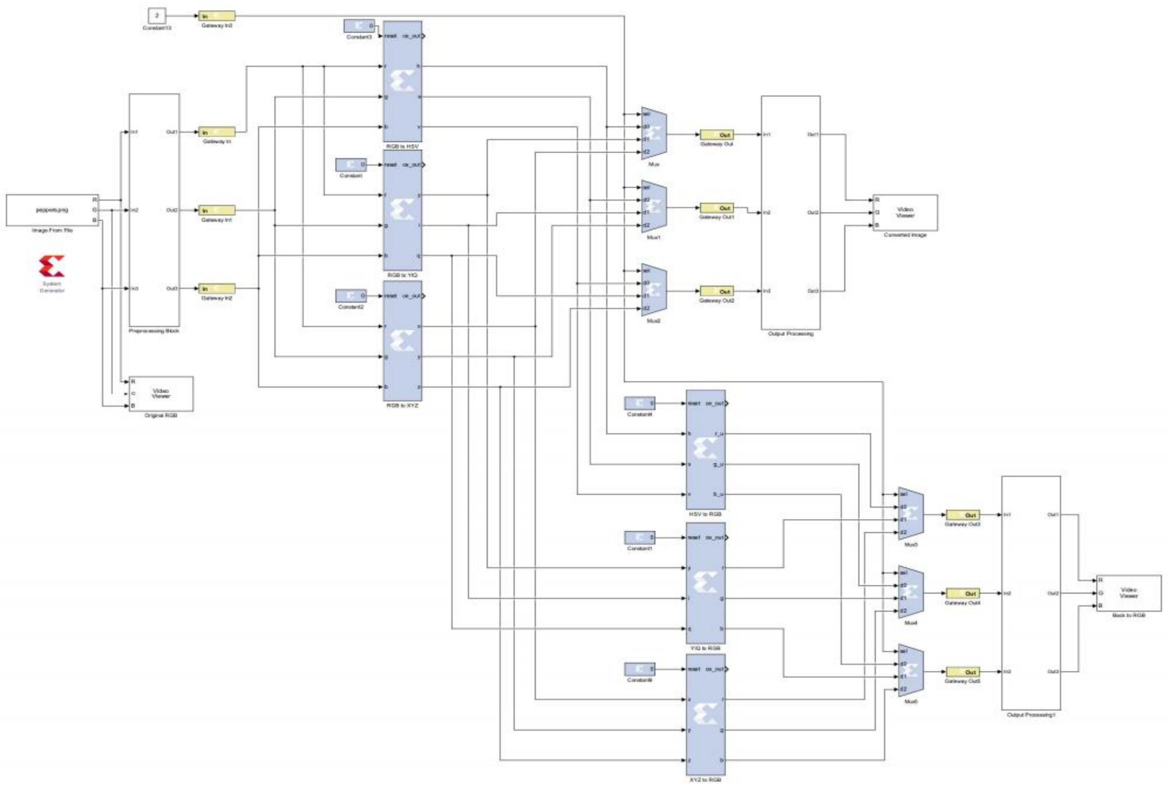
The three inputs corresponding to the R, G, B values are connected to the inputs of the RGB to HSV, RGB to YIQ, and RGB to YXZ blocks. The output of those are connected to the inputs of the HSV to RGB, YIQ to RGB, and XYZ to RGB blocks respectively. The outputs of all six blocks are connected to six three-input multiplexors, controlled by a two-bit constant control variable that the user can change. Then the outputs from the multiplexors are connected to two Output Processing blocks. One of them is in charge of rearranging the bits into matrix form for the converted image, and the other one in charge of the same process for the reconstructed RGB image.

Each output is processed by the following Output Processing blocks:



iii. The output images can be viewed using the Video Viewer block.

iv. System Block Diagram:



See appendix for a larger version of the diagram.

4) Xilinx Project

   i. Generating Xilinx Project

   The Xilinx Project is generated for the Xilinx Virtex6 xc6vcx130t-ff1156-1 device. The top module is implemented as color_space_cw.v and the main block is implemented as color_space.v. The black boxes for the conversion algorithms are instantiated inside the color_space module.

   ii. IP Core Generation

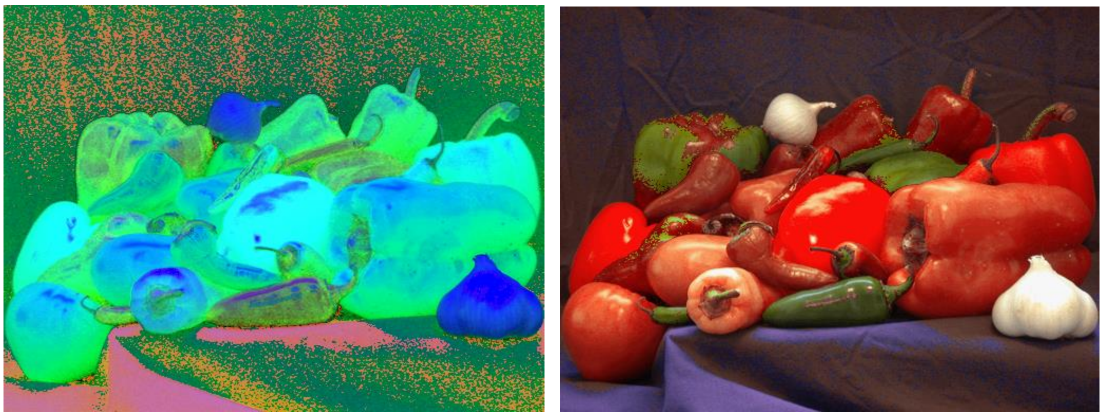
   The RGB to HSV algorithm involves four division of doubles. In the generated Verilog code, division is implemented as "quotient = dividend / divisor". Because the target device can only implement division using shift in Verilog, i.e. the divisor must be a power of 2, this function cannot be synthesized. Two IP Cores dividers are generated. One implements a high-radix division of a 32-bit number by a 16-bit number, and another implements that of a 33-bit number by a 16-bit number. One module of divider0 and three modules of divider1 instantiated, and the corresponding division operations inside the source code are replaced by assigning the wires to the quotient outputs of the dividers.

   iii. Now the project is ready for synthesis and simulation.

5) Simulink Simulation Results

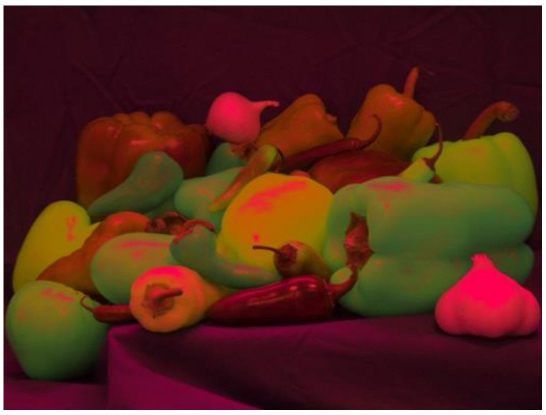
   i. Original Image: peppers.png. Resolution: 512 x 384.

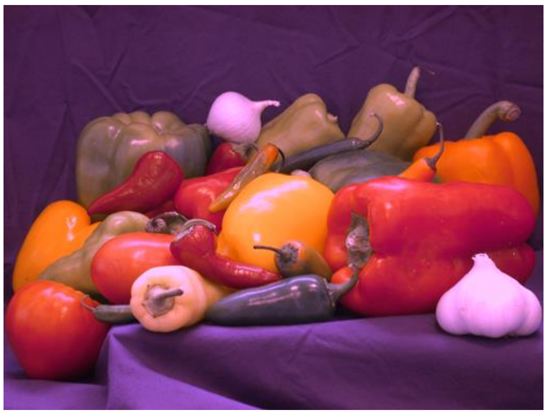
   ii. RGB to HSV and HSV to RGB Results



   After the RGBtoHSV and HSVtoRGB conversion, the red, white, dark green, and purple colors are preserved well. However yellow turns to red, light green turns to dark red, and orange turns to bright red. Noise is also added to the background.

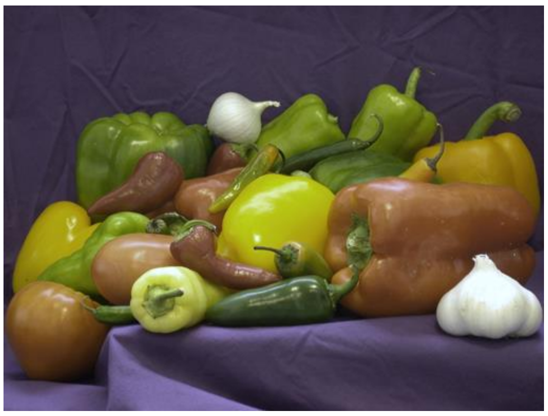
iii. RGB to YIQ and YIQ to RGB Results

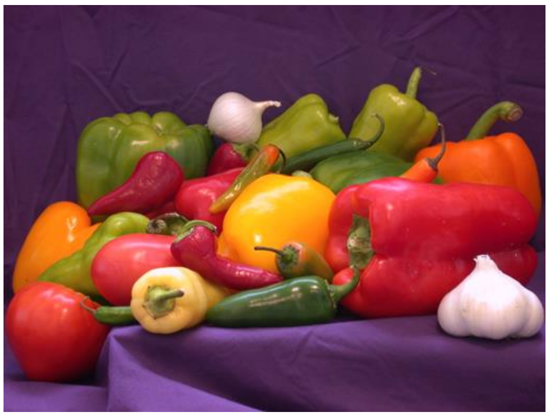




After the RGBtoYIQ and YIQtoRGB conversion, almost all colors are preserved well. The only flaw is that green loses saturation and white turns to light pink. The whole image is also reduced in terms of brightness.

iv. RGB to XYZ and XYZ





After the RGBtoXYZ and XYZtoRGB conversion, there is no noticeable difference between the original RGB image and the converted RGB image.

6. Xilinx Synthesis and Implementation Results

1) Timing Report

i. Timing Summary

Errors	Connections	Nets	Paths	Score
0	11715	0	13041	0

ii. Timing Constraints

Worst Case Slack	Best Case Achievable
6.807ns 0.003ns	3.193ns

2) Device Utilization

Slice Logic Utilization	Used	Available	Utilization
Slices	1, 203	20, 000	6%
Slice Registers	3, 427	160, 000	2%
Slice LUTs	3, 517	80, 000	4%
DSP48E1	92	480	19%

3) RTL Schematic
   See appendix.

7. Conclusion

The FPGA implementation of the Color Space Conversion algorithms is feasible even though traditionally it is hard for FPGAs to do division and calculations involving floating-point numbers.

In this project, the RGBtoXYZ and XYZtoRGB functions have the best results as the converted RGB image looks the same as the original RGB. However, the other two conversions have problems. The HSV model creates very bright color blocks and brings noise into the background. The YIQ model causes the image to lose saturation and brightness. The above two problems are probably caused by the following:

- Rounding Errors

  There are number of times when numbers are rounded. Inside each function, when the calculated values in double format are cast to uint8 format, there is significant amount of rounding, which can cause distortions in the image. The same problem can occur in the fix-point versions of the functions and thus they are carried into the HDL design.

- Overflow Errors

  Because all data eventually are stored in uint8 format, for unsigned numbers generated in the algorithms that are larger than 255, no matter what true value they have they would be set as 255 when cast to uint8 (when the overflow method is set as “Saturate”). This brings loss into the images as when these values are converted back to RGB, two pixels that have different values in the original RGB image might now have the same value of either 255 or 0 in the converted image. Loss is introduced for a second time when trying to convert it back to RGB format, with the same mechanisms of overflow. In this way, there are two chances for data to overflow in the conversion process, and the reconstructed RGB images of HSVtoRGB and YIQtoRGB are very possibly in the wrong colors because of this issue.

Meanwhile, the results of the project do show that some or most of the colors in the original RGB image can be converted to another color field and back again with little deviation from the original. Thus it provides a basis for other image processing techniques to be implemented using Simulink, System Generator, Xilinx and FPGA.

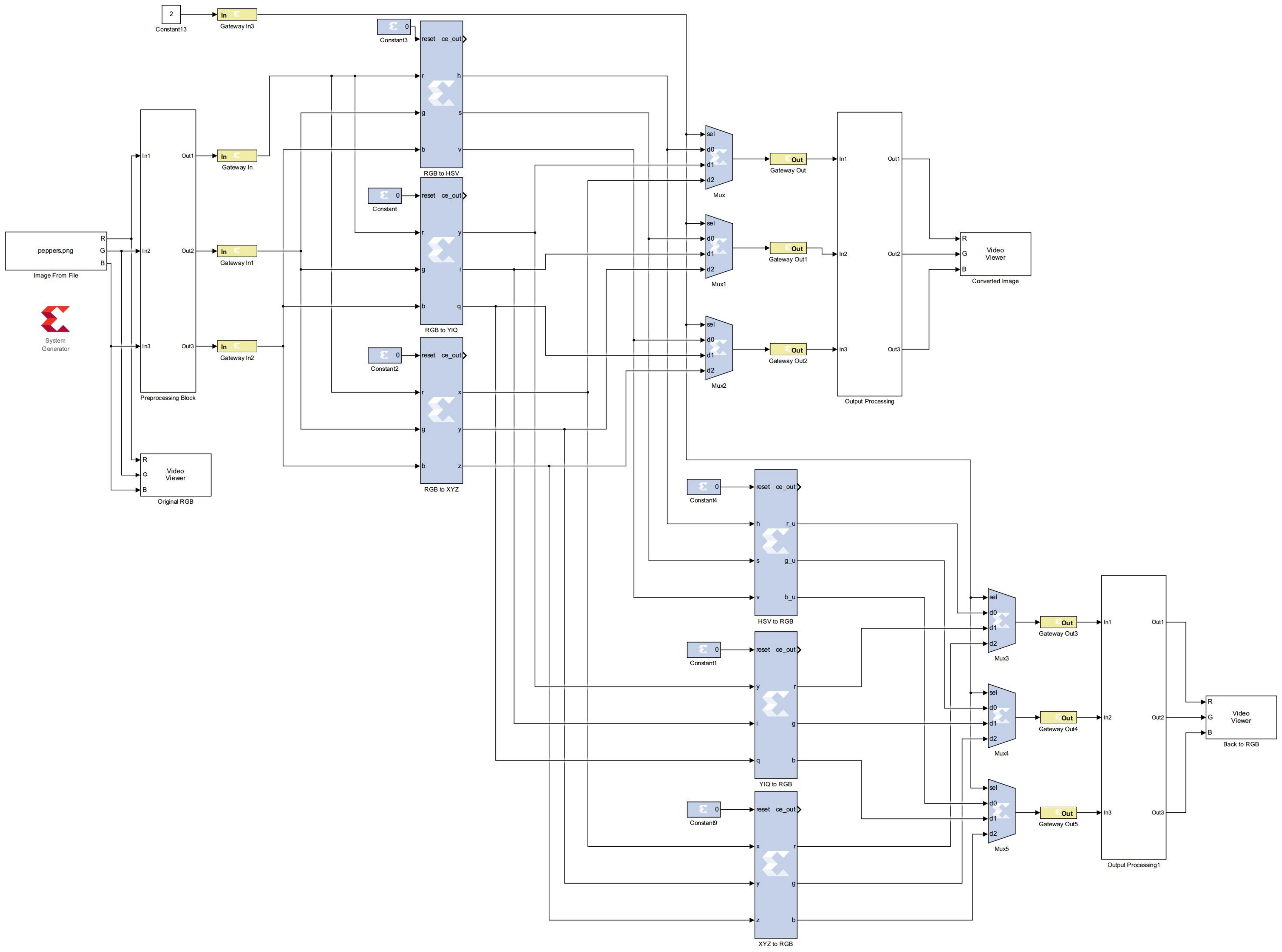
8. References

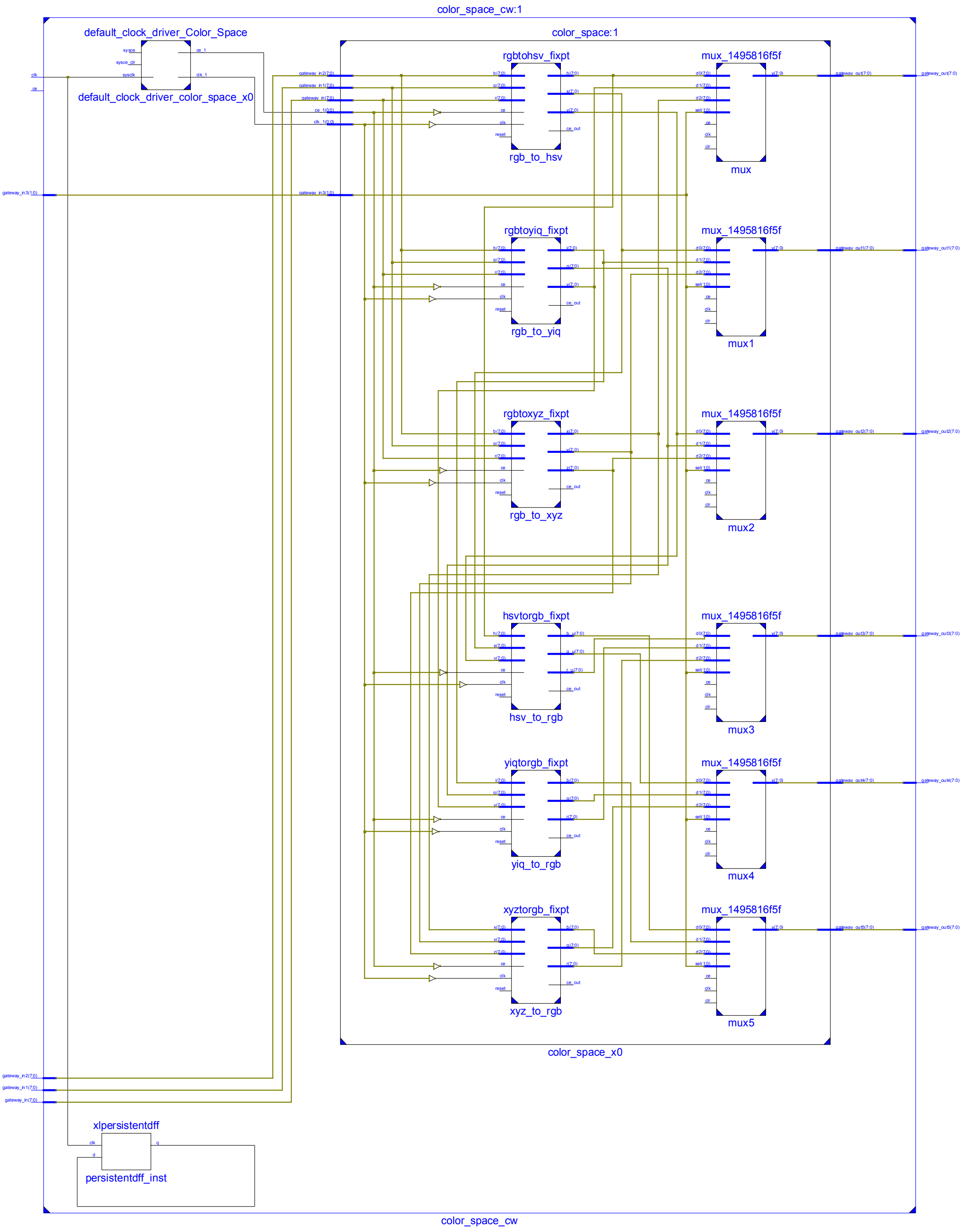
http://www.iosrjournals.org/iosr-jvlsi/papers/vol2-issue4/E0242636.pdf
http://forums.xilinx.com/t5/DSP-Tools/Image-Processing-in-Xilinx-System-Generator-Simple-Q/td-p/84908
http://www.cs.rit.edu/~ncs/color/t_convert.html

http://www.rapidtables.com/convert/color/rgb-to-hsv.htm
http://www.rapidtables.com/convert/color/hsv-to-rgb.htm
http://angeljohnsy.blogspot.com/2011/04/rgb-image-to-yiq-image.html
http://angeljohnsy.blogspot.com/2011/04/yiq-image-to-rgb-image.html
http://www.brucelindbloom.com/index.html?Eqn_RGB_XYZ_Matrix.html

9. List of Appendices

1) System Block Diagram
2) RTL Schematic
3) IP Core Divider Instantiation Code in Verilog
4) rgbtohsv.m
5) hsvtorgb.m
6) rgbtoyiq.m
7) rgbtohsv_tb.m


2
Constant13
Gateway In3
Constant3
reset ce_out
r h
g s
b v
RGB to HSV
Constant
r y
g i
b q
RGB to YIQ
Constant2
r x
g y
b z
RGB to XYZ
peppers.png
R G B
Image From File
System Generator
In1 Out1
In2 Out2
In3 Out3
Preprocessing Block
Gateway In
Gateway In1
Gateway In2
Video Viewer
Original RGB
sel d0 d1 d2
Mux
Mux1
Mux2
Gateway Out
Gateway Out1
Gateway Out2
Output Processing
Video Viewer
Converted Image
Constant4
h r_u
s g_u
v b_u
HSV to RGB
Constant1
y r
i g
q b
YIQ to RGB
Constant9
x r
y g
z b
XYZ to RGB
Mux3
Mux4
Mux5
Gateway Out3
Gateway Out4
Gateway Out5
Output Processing1
Video Viewer
Back to RGB



color_space_cw:1
default_clock_driver_Color_Space
sysce
sysce_clr
sysclk
ce_1
clk_1
default_clock_driver_color_space_x0
clk
ce
color_space:1
gateway_in2(7:0)
gateway_in1(7:0)
gateway_in(7:0)
ce_1(0:0)
clk_1(0:0)
gateway_in3(1:0)
rgbtohsv_fixpt
rgb_to_hsv
rgbtoyiq_fixpt
rgb_to_yiq
rgbtoxyz_fixpt
rgb_to_xyz
hsvtorgb_fixpt
hsv_to_rgb
yiqtorgb_fixpt
yiq_to_rgb
xyztorgb_fixpt
xyz_to_rgb
b(7:0)
g(7:0)
r(7:0)
h(7:0)
s(7:0)
v(7:0)
y(7:0)
i(7:0)
q(7:0)
x(7:0)
z(7:0)
b_u(7:0)
g_u(7:0)
r_u(7:0)
ce
clk
reset
ce_out
mux_1495816f5f
mux
mux1
mux2
mux3
mux4
mux5
d0(7:0)
d1(7:0)
d2(7:0)
sel(1:0)
clr
y(7:0)
gateway_out(7:0)
gateway_out1(7:0)
gateway_out2(7:0)
gateway_out3(7:0)
gateway_out4(7:0)
gateway_out5(7:0)
color_space_x0
xlpersistentdff
clk
d
q
persistentdff_inst
color_space_cw

```
/* Added by HAN Haiyang */
wire [31:0] divider_quotient0;
wire [32:0] divider_quotient1;
wire [32:0] divider_quotient2;
wire [32:0] divider_quotient3;

wire [15:0] divider_fractional0;
wire [7:0] divider_fractional1;
wire [7:0] divider_fractional2;
wire [7:0] divider_fractional3;

divider0 divider0 (
  .clk(clk), // input clk
  .nd(1), // input nd
  //.rdy(1), // output rdy
  //.rfd(1), // output rfd
  .dividend(rgbtohsv_fixpt_cast_5_1), // input [31 : 0] dividend
  .divisor(rgbtohsv_fixpt_Cmax_1), // input [15 : 0] divisor
  .quotient(divider_quotient0), // output [31 : 0] quotient
  .fractional(divider_fractional0)); // output [15 : 0] fractional

  divider1 divider1 (
  .clk(clk), // input clk
  .nd(1), // input nd
  //.rdy(1), // output rdy
  //.rfd(1), // output rfd
  .dividend(rgbtohsv_fixpt_cast_11_1), // input [32 : 0] dividend
  .divisor(rgbtohsv_fixpt_delta_1), // input [15 : 0] divisor
  .quotient(divider_quotient1), // output [32 : 0] quotient
  .fractional(divider_fractional1)); // output [7 : 0] fractional

 divider1 divider2 (
  .clk(clk), // input clk
  .nd(1), // input nd
  //.rdy(1), // output rdy
  //.rfd(1), // output rfd
  .dividend(rgbtohsv_fixpt_cast_15_1), // input [32 : 0] dividend
  .divisor(rgbtohsv_fixpt_delta_1), // input [15 : 0] divisor
  .quotient(divider_quotient2), // output [32 : 0] quotient
  .fractional(divider_fractional2)); // output [7 : 0] fractional

   divider1 divider3 (
  .clk(clk), // input clk
  .nd(1), // input nd
  //.rdy(1), // output rdy
  //.rfd(1), // output rfd
  .dividend(rgbtohsv_fixpt_cast_17_1), // input [32 : 0] dividend
  .divisor(rgbtohsv_fixpt_delta_1), // input [15 : 0] divisor
  .quotient(divider_quotient3), // output [32 : 0] quotient
  .fractional(divider_fractional3)); // output [7 : 0] fractional

  /*end of code added by HAN Haiyang */
```

```
function [h, s, v] = rgbtohsv(r, g, b)
    r_d = double(r);
    g_d = double(g);
    b_d = double(b);
    r_d = r_d / 255;
    g_d = g_d / 255;
    b_d = b_d / 255;

    Cmax = max(max(r_d,g_d),b_d);
    Cmin = min(min(r_d,g_d),b_d);
    delta = Cmax - Cmin;

    h_d = 0;
    v_d = Cmax;
    s_d = delta;

    if Cmax == 0,
        s_d = 0;
    else
        s_d = delta / Cmax;
    end

    if delta == 0,
        h_d = 0;
    elseif r_d == Cmax,
        h_d = (g_d - b_d) / delta;
    elseif g_d == Cmax,
        h_d = 2 + (b_d - r_d) / delta;
    elseif b_d == Cmax,
        h_d = 4 + (r_d - g_d) / delta;
    end

    h_d = h_d / 6;
    if h_d < 0,
        h_d = h_d + 1;
    end

    h=uint8(h_d*255);
    s=uint8(s_d*255);
    v=uint8(v_d*255);
end
```

```
function [r_u, g_u, b_u] = hsvtorgb(h, s, v)
    h_d=double(h);
    s_d=double(s);
    v_d=double(v);
    h_d=h_d/255;
    s_d=s_d/255;
    v_d=v_d/255;

    h_d = h_d*6;
    i=floor(h_d);
    f = h_d-i;
    p = v_d*(1-s_d);
    q = v_d*(1-s_d*p);
    t = v_d*(1-(s_d*(1-p)));

    switch i
        case 0
            r = v_d;
            g = t;
            b = p;
        case 1
            r = q;
            g = v_d;
            b = p;
        case 2
            r = p;
            g = v_d;
            b = t;
        case 3
            r = p;
            g = q;
            b = v_d;
        case 4
            r = t;
            g = p;
            b = v_d;
        case 5
            r = v_d;
            g = p;
            b = q;
        otherwise
            r = v_d;
            g = t;
            b = p;
    end

    r_u = uint8(r*255);
    g_u = uint8(g*255);
    b_u = uint8(b*255);
end
```

```
function [y, i, q] = rgbtoyiq(r, g, b)
    y=uint8(0.2989*r+0.5870*g+0.1140*b);
    i=uint8(0.596*r-0.274*g-0.322*b);
    q=uint8(0.211*r-0.523*g+0.312*b);
end
```

```
image_rgb=imread('peppers.png');
figure(1);
imshow(image_rgb);
image_hsv=uint8(zeros(size(image_rgb)));
for j=1:size(image_rgb,1)
    for k=1:size(image_rgb,2)
        [h, s ,v]=rgbtohsv(image_rgb(j,k,1),image_rgb(j,k,2),image_rgb(j,k,3));
        image_hsv(j,k,1)=h;
        image_hsv(j,k,2)=s;
        image_hsv(j,k,3)=v;
    end
end
figure(2);
imshow(image_hsv);
imwrite(image_hsv,'peppers_hsv.png');
```