CE2003
Lab 4 Report

Name: Gao He

Matric Number: U1220847J

Group: FEP1

1. Introduction

   The aim of this lab is to build a project to process pixel input from a camera and to show them on the screen in real time.

   During this lab, we mainly need to apply the knowledge of **pipelining** to increase clock frequency, and **signal aligning** to synchronize related signals. In addition, to detect human skin better, we need to **convert colourspaces** (RGB pixels captured from the camera to YUV pixels) first, and then add logics to detect human skin. Finally we need to use **case and if statements** and **saturation logic** to create different effects.

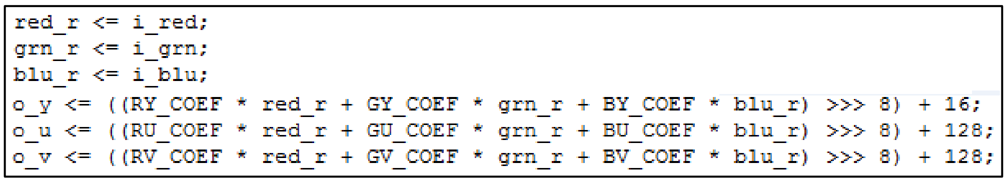
   There are 3 modules for us to design. We need to focus on pipelining design in *rgbyuv* module; aligning signals in *delay_line* module; and process the streamed pixels in *pixsel* module.

   In this way, when we set the switches to a certain value, we can observe special visual effects such as turning the entire screen to black and white or changing my face to red with other regions normal – just like a simpler version of Photo Booth.

2. Procedures

   A. Task 1

   1) First we need to convert the pixel signals from RGB colourspace to YUV colourspace based on the given example code. The screenshot of the finished code is shown below as Figure 1.

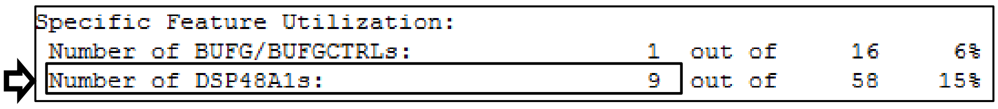


```
red_r <= i_red;
grn_r <= i_grn;
blu_r <= i_blu;
o_y <= ((RY_COEF * red_r + GY_COEF * grn_r + BY_COEF * blu_r) >>> 8) + 16;
o_u <= ((RU_COEF * red_r + GU_COEF * grn_r + BU_COEF * blu_r) >>> 8) + 128;
o_v <= ((RV_COEF * red_r + GV_COEF * grn_r + BV_COEF * blu_r) >>> 8) + 128;
```

Figure 1

   In this code, the RGB input signals *i_* are passed in the corresponding registers (named as *_r*). And then the registers will be put in existing 3 formulas to convert to YUV output signals (named as *o_*).

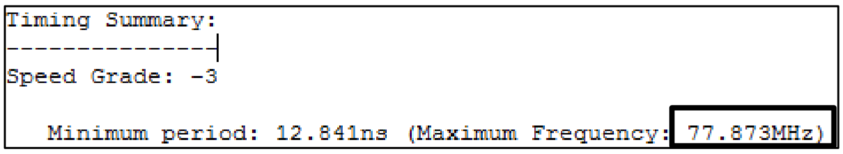
   2) Synthesize the existing design and check the synthesis report. Under the synthesis report, there is a **device utilization summary**. We can see from here that, 9 DSP blocks are in utilization, shown by Figure 2.



```
Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                1  out of     16     6%
 Number of DSP48A1s:                      9  out of     58    15%
```

Figure 2

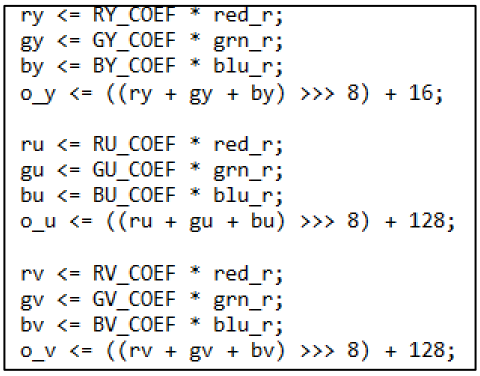
   3) In the synthesis report, there is also a **timing summary** inside. From there we can find the max frequency here, which is **77.873MHz**. (Figure 3)



```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 12.841ns (Maximum Frequency: 77.873MHz)
```

Figure 3

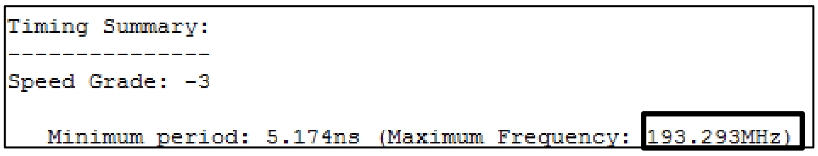
4) In order to speed up the clock frequency to meet the requirement of video datapath frequency 108MHz, we need to pipeline the design.
I created new registers (named *ry,ru...* accordingly) to pass in the value of the coefficients multiplied by the *_r* register signals, and then put the new registers to the formulas. In this way, I have added a single pipeline stage to my code. The changed code is shown in Figure 4 below.



```
ry <= RY_COEF * red_r;
gy <= GY_COEF * grn_r;
by <= BY_COEF * blu_r;
o_y <= ((ry + gy + by) >>> 8) + 16;

ru <= RU_COEF * red_r;
gu <= GU_COEF * grn_r;
bu <= BU_COEF * blu_r;
o_u <= ((ru + gu + bu) >>> 8) + 128;

rv <= RV_COEF * red_r;
gv <= GV_COEF * grn_r;
bv <= BV_COEF * blu_r;
o_v <= ((rv + gv + bv) >>> 8) + 128;
```

Figure 4

After synthesis, I can find the new max frequency to be **193.293** (Figure 5), which meets the requirement of 108Mhz this time.

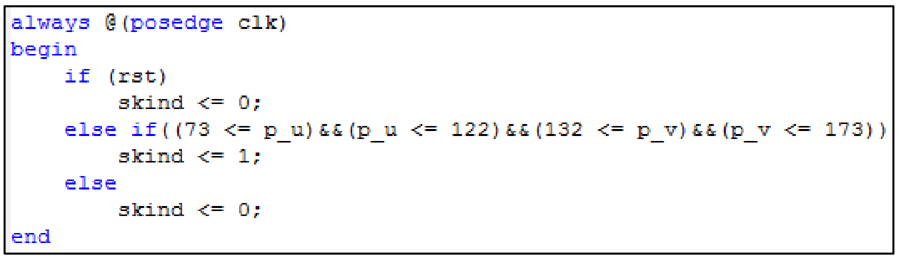


```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 5.174ns (Maximum Frequency: 193.293MHz)
```

Figure 5

(More discussion on pipelining will be given at part 4 – Discussion.) For the rest of the procedures, I will keep using this design.

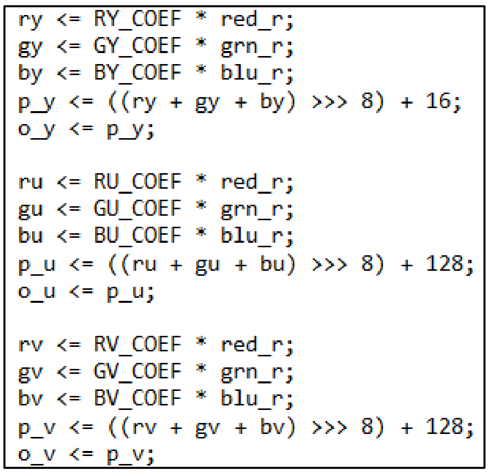
5) We need a second always block to let the computer to recognize whether or not a pixel is in skin color, and store the Boolean value in an output register named *"skind"*. I wrote the code based on the given inequalities, shown in Figure 6 below.



```
always @(posedge clk)
begin
    if (rst)
        skind <= 0;
    else if((73 <= p_u)&&(p_u <= 122)&&(132 <= p_v)&&(p_v <= 173))
        skind <= 1;
    else
        skind <= 0;
end
```

Figure 6

To align the *o_* signals with *skind*, new registers *p_* need to be introduced to the first always block. This will let the converted YUV signals to wait for *skind* for one cycle. The changed code is shown below.



```
ry <= RY_COEF * red_r;
gy <= GY_COEF * grn_r;
by <= BY_COEF * blu_r;
p_y <= ((ry + gy + by) >>> 8) + 16;
o_y <= p_y;

ru <= RU_COEF * red_r;
gu <= GU_COEF * grn_r;
bu <= BU_COEF * blu_r;
p_u <= ((ru + gu + bu) >>> 8) + 128;
o_u <= p_u;

rv <= RV_COEF * red_r;
gv <= GV_COEF * grn_r;
bv <= BV_COEF * blu_r;
p_v <= ((rv + gv + bv) >>> 8) + 128;
o_v <= p_v;
```

Figure 7

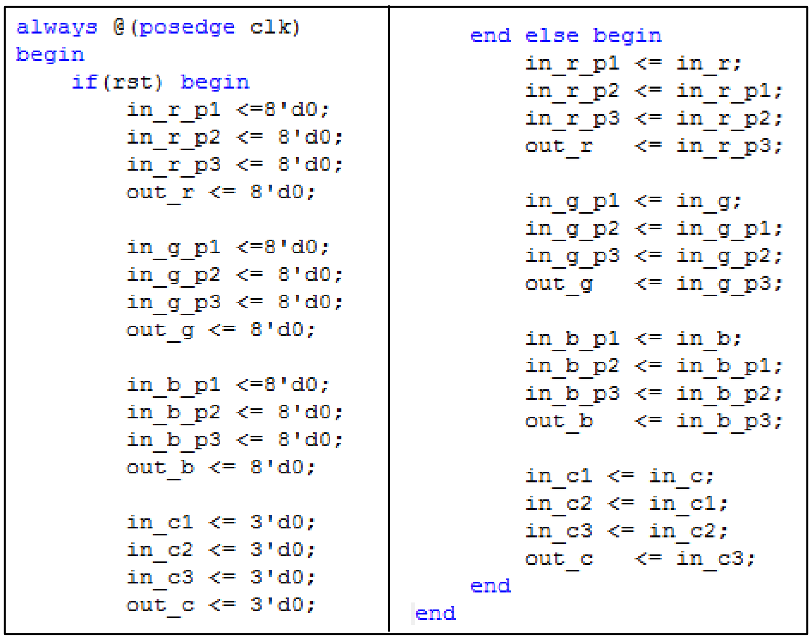
(Details of signal aligning can be found in discussion part)

6) Then we need to save all the changes and synthesize the design again. As I didn't change pipelining design, the maximum clock frequency remains the same, **193.293MHz** with **4 stages** in total (**1 added pipeline**).

B. Task 2

Here we need to start a new project and include the revised *rgbyuv* module. Now we need to edit *"delay_line.v"* to generate delays for input pixel and control signals. As I have 4 stages currently in *"rgbyuv.v",* I also need to set **4 stages** in *"delay_line.v"*. The code is shown below (Figure 8).

(Details of signal aligning can be found in discussion part)



```
always @(posedge clk)
begin
    if(rst) begin
        in_r_p1 <=8'd0;
        in_r_p2 <= 8'd0;
        in_r_p3 <= 8'd0;
        out_r <= 8'd0;

        in_g_p1 <=8'd0;
        in_g_p2 <= 8'd0;
        in_g_p3 <= 8'd0;
        out_g <= 8'd0;

        in_b_p1 <=8'd0;
        in_b_p2 <= 8'd0;
        in_b_p3 <= 8'd0;
        out_b <= 8'd0;

        in_c1 <= 3'd0;
        in_c2 <= 3'd0;
        in_c3 <= 3'd0;
        out_c <= 3'd0;
```

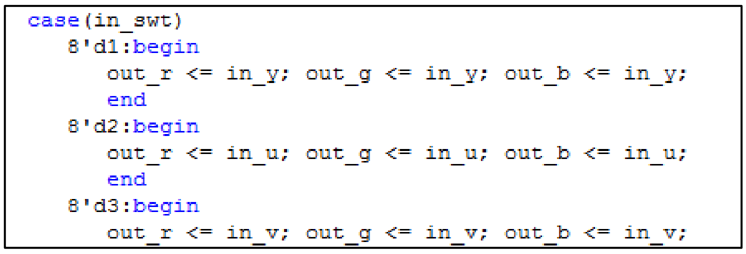
```
    end else begin
        in_r_p1 <= in_r;
        in_r_p2 <= in_r_p1;
        in_r_p3 <= in_r_p2;
        out_r   <= in_r_p3;

        in_g_p1 <= in_g;
        in_g_p2 <= in_g_p1;
        in_g_p3 <= in_g_p2;
        out_g   <= in_g_p3;

        in_b_p1 <= in_b;
        in_b_p2 <= in_b_p1;
        in_b_p3 <= in_b_p2;
        out_b   <= in_b_p3;

        in_c1 <= in_c;
        in_c2 <= in_c1;
        in_c3 <= in_c2;
        out_c   <= in_c3;
    end
end
```

Figure 8

C. Task 3

1) Now we need to add case statements for case 1-3 to output all the RGB signals based on *in_y, in_u* and *in_v* correspondingly.

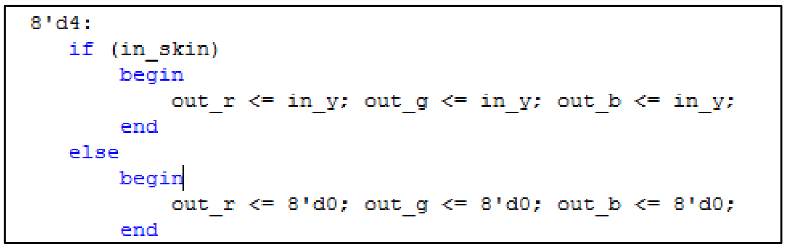
The regarding code is shown below.



```
case(in_swt)
   8'd1:begin
      out_r <= in_y; out_g <= in_y; out_b <= in_y;
      end
   8'd2:begin
      out_r <= in_u; out_g <= in_u; out_b <= in_u;
      end
   8'd3:begin
      out_r <= in_v; out_g <= in_v; out_b <= in_v;
```

Figure 9

2) Next we can implement the design and generate programming file. With the switches, I can see the entire screen turning black and white in switch 1, turning inversed darkness and in grey in switch 2 and turning palely grey in switch 3.
(Final visual effect pictures are attached to appendix for reference)
3) For case 4, we need to write an **if-statement** to show only the skin pixels. Here we can use "*skind*" to build the code. The code is shown below.

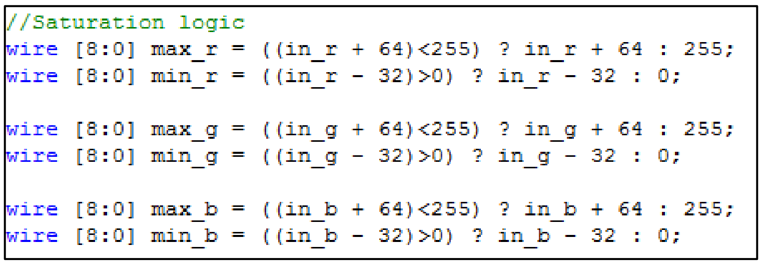


```
8'd4:
   if (in_skin)
       begin
           out_r <= in_y; out_g <= in_y; out_b <= in_y;
       end
   else
       begin
           out_r <= 8'd0; out_g <= 8'd0; out_b <= 8'd0;
       end
```

Figure 10

4) Now I need to implement the design again. With the switch set to 4, I could see that, the whole screen is black and white again, but this time my skin color looked very light and all other things turned black.

D. Task 4
1) Add the statements for **saturation logic** based on the formula given and the example code for red. The code is shown below.



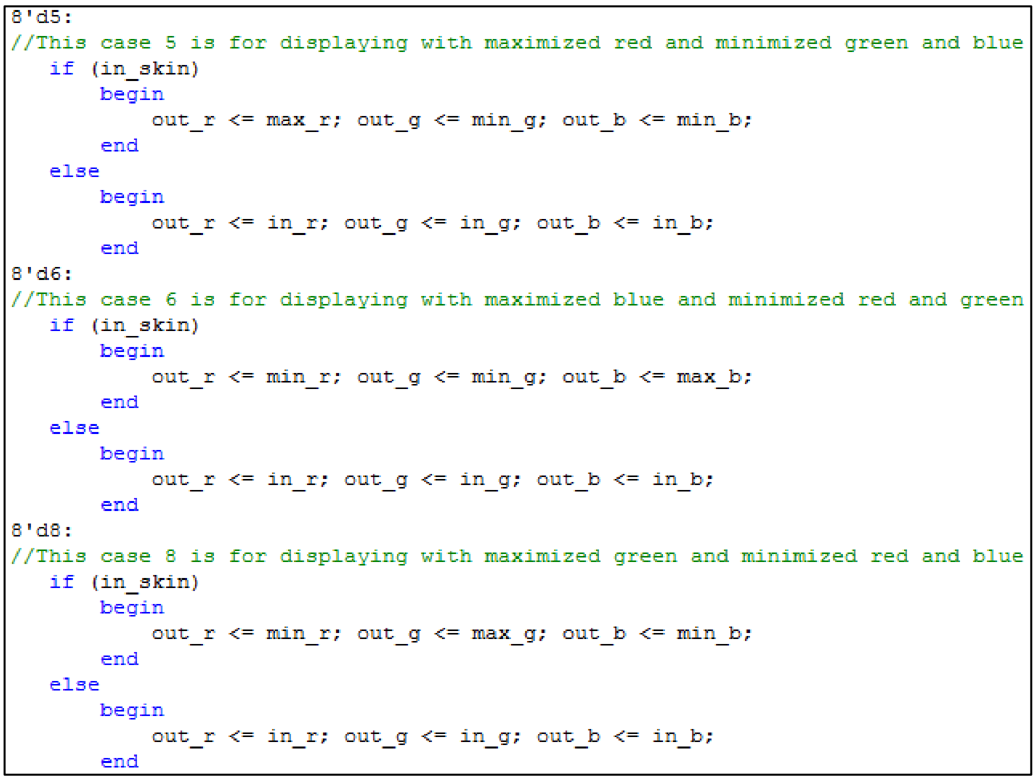
```
//Saturation logic
wire [8:0] max_r = ((in_r + 64)<255) ? in_r + 64 : 255;
wire [8:0] min_r = ((in_r - 32)>0) ? in_r - 32 : 0;

wire [8:0] max_g = ((in_g + 64)<255) ? in_g + 64 : 255;
wire [8:0] min_g = ((in_g - 32)>0) ? in_g - 32 : 0;

wire [8:0] max_b = ((in_b + 64)<255) ? in_b + 64 : 255;
wire [8:0] min_b = ((in_b - 32)>0) ? in_b - 32 : 0;
```

Figure 11

2) Next we need to add 3 more case statements such that we can strengthen red, green or blue for skin color.

For **case 8**, I set the maximized green and minimized the other 2 colors for skin pixels and unchanged color for other pixels.
Similarly, I set maximized red for **case 5** and blue for **case 6**.
As **case 7** is not defined, so for this case, the screen should display the default case, and hence display unchanged pixels.

The regarding code is shown below.



```
8'd5:
//This case 5 is for displaying with maximized red and minimized green and blue
   if (in_skin)
       begin
           out_r <= max_r; out_g <= min_g; out_b <= min_b;
       end
   else
       begin
           out_r <= in_r; out_g <= in_g; out_b <= in_b;
       end
8'd6:
//This case 6 is for displaying with maximized blue and minimized red and green
   if (in_skin)
       begin
           out_r <= min_r; out_g <= min_g; out_b <= max_b;
       end
   else
       begin
           out_r <= in_r; out_g <= in_g; out_b <= in_b;
       end
8'd8:
//This case 8 is for displaying with maximized green and minimized red and blue
   if (in_skin)
       begin
           out_r <= min_r; out_g <= max_g; out_b <= min_b;
       end
   else
       begin
           out_r <= in_r; out_g <= in_g; out_b <= in_b;
       end
```

Figure 12

3) Finally, we can implement the whole design and observe different visual effects from switch 1 to 8. (Pictures attached to appendix.)

3. Discussion

A. On pipelining design.

1) **Non-pipelined**.

Maximum Frequency: 77.873MHz

Stages: 2 (Aligning stage not included, since it's not in overall design)

Multipliers used: 9

Design Diagram:

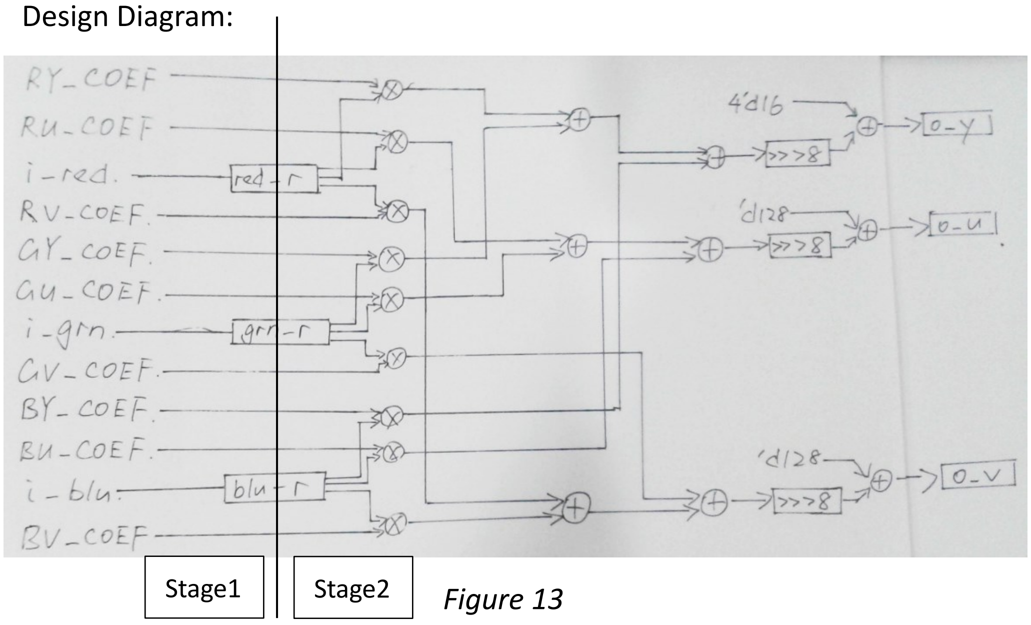


Figure 13

2) **Pipelined with a single added stage**.
Maximum Frequency: 193.293 MHz
Stages: 4 (Aligning stage included)
Multipliers used: 9
Design Diagram:

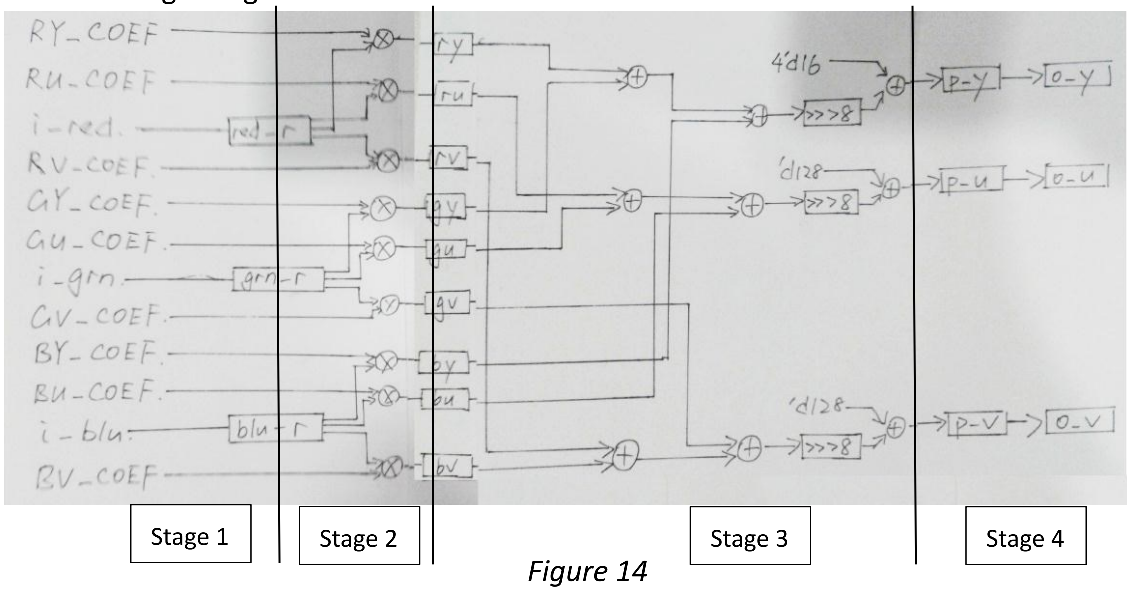


Figure 14

3) **Fully pipelined**. (Every calculation is split up here)
Maximum Frequency: 456.621MHz
Stages: 6 (Aligning stage not included, since it's not in overall design)
Multipliers used: 9
Design Diagram:

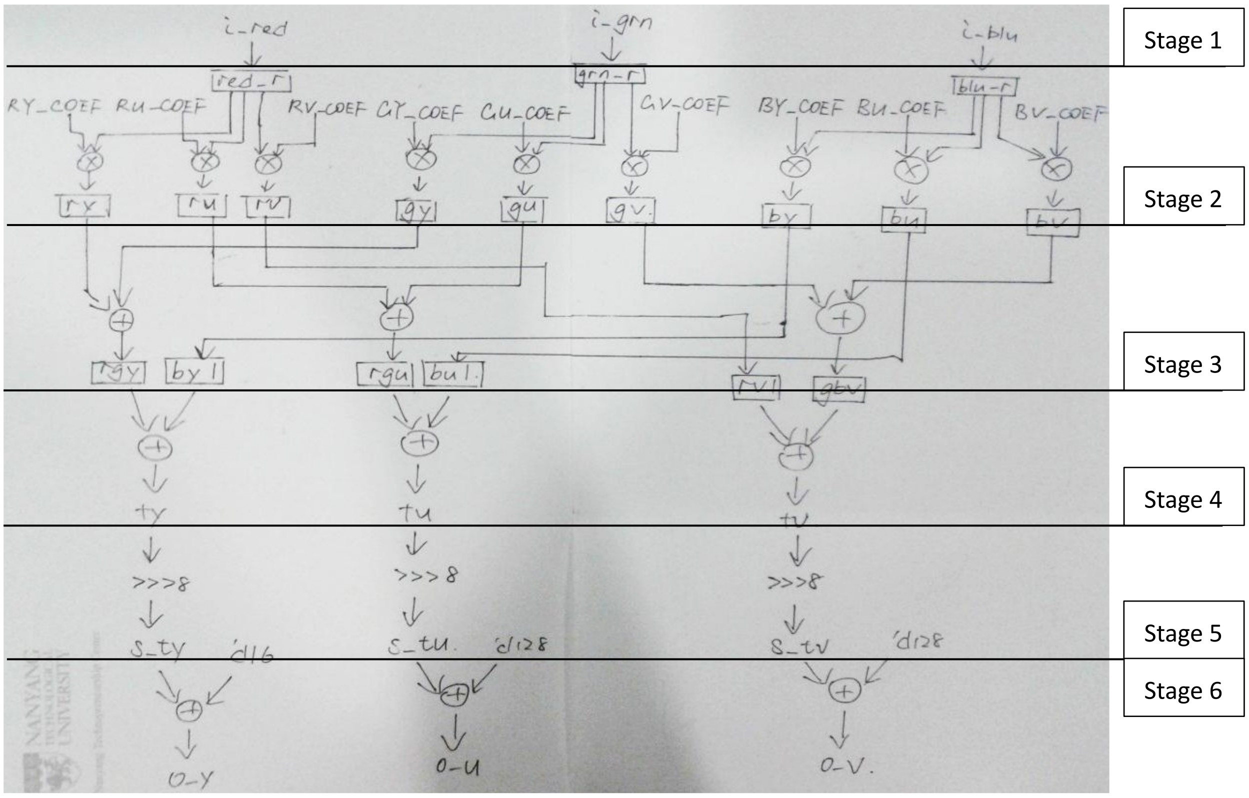
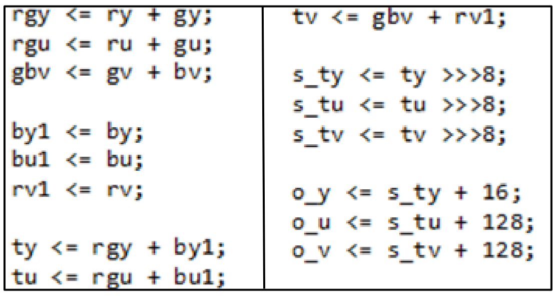


Figure 15

Revised part of the code:

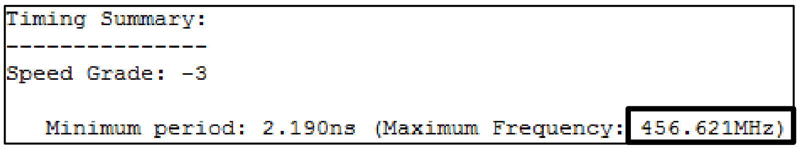


```
rgy <= ry + gy;
rgu <= ru + gu;
gbv <= gv + bv;

by1 <= by;
bu1 <= bu;
rv1 <= rv;

ty <= rgy + by1;
tu <= rgu + bu1;
```

```
tv <= gbv + rv1;

s_ty <= ty >>>8;
s_tu <= tu >>>8;
s_tv <= tv >>>8;

o_y <= s_ty + 16;
o_u <= s_tu + 128;
o_v <= s_tv + 128;
```

Figure 16

Timing summary:



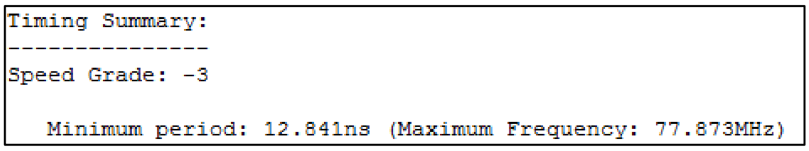
```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 2.190ns (Maximum Frequency: 456.621MHz)
```

Figure 17

4) During pipelining, we should be careful of which project to use.

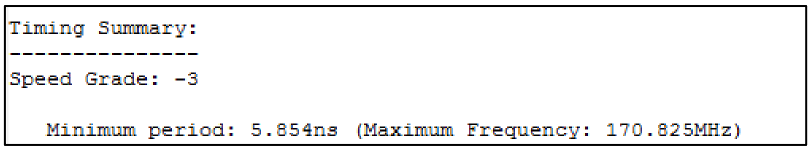
   We have 2 projects in this lab: one only have *rgbyuv* module, and the other one includes all the 3 modules and all the files in the zipped folder from edventure.
   To design pipelining, we need to use the 1st **project** to get the maximum clock frequencies (**77.873MHz, 193.293 MHz and 456.621MHz**).

   If I use the 2nd project in this case, I would only be able to get an overall clock frequency of the 3 modules, which is limited by the module with the lowest frequency.
   For example, if I use the exactly same code for the above 3 pipelining designs in the 2nd project, the maximum frequencies would be **77.873MHz, 170.825 MHz and 170.825MHz** respectively, which is **wrong in testing pipelining**. (Shown in Figure 18 and 19)



```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 12.841ns (Maximum Frequency: 77.873MHz)
```

Figure 18 Non-pipelining design synthesized in 2nd project (unwanted)



```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 5.854ns (Maximum Frequency: 170.825MHz)
```

Figure 19 Single pipelining and fully pipelining synthesized in 2nd project (unwanted)

B. On signal aligning.

In order to align different signal to the same cycle signal aligning is need in 2 places in this experiment.

1) In the 5th step of Task 1.
   It takes one clock cycle to decide whether the converted signal is a skin pixel, and store the Boolean value to *skind*. So to make *skind* and the output signal *o_* come out at the same cycle, signal aligning is needed. Hence we need new registers *p_* and pass them to *o_ signals*, to delay the output for one clock cycle.

2) Task 2.
   The entire *delay_line* module is meant for signal aligning.

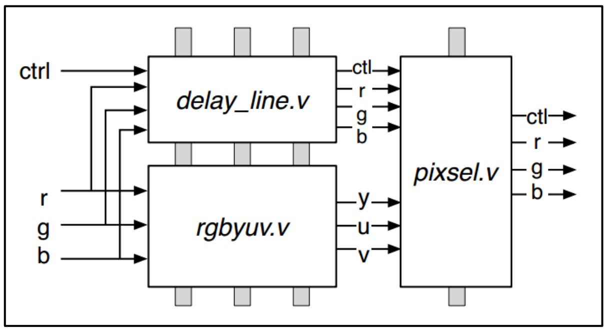


Figure 16

From the figure above, we can see that the output signals (*ctl, r, g, b*) of *delay_line* module need to be synchronous with those (*y, u, v*) of *rgbyuv* . As we have 4 stages in *rgbyuv* module, we also need 4 stages in *delay_line* module. Hence we need *in_ _p1/2/3* to transparently pass on the signals all the way to *out_* signals. Hence each input signal will be passed to output signal after 3 more cycles, and so the outputs of *rgbyuv* module and *delay_line* module will represent the video input at the same instant.

4. Conclusion

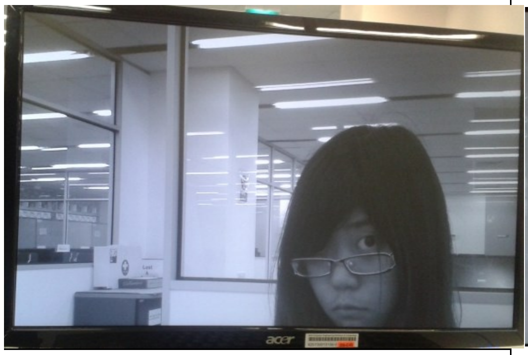
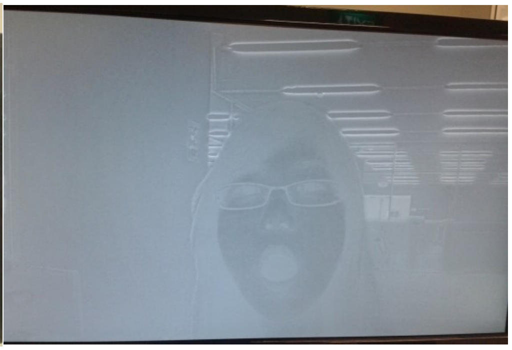
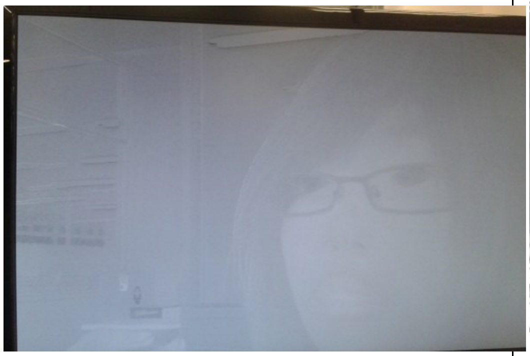
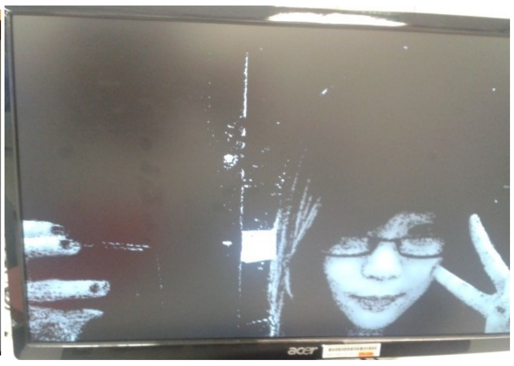
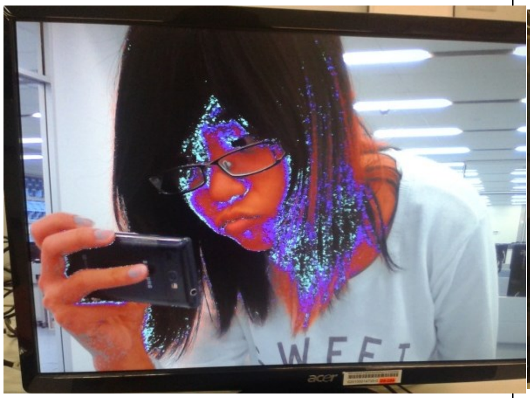
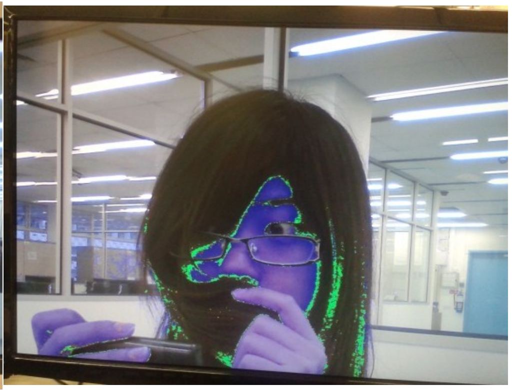
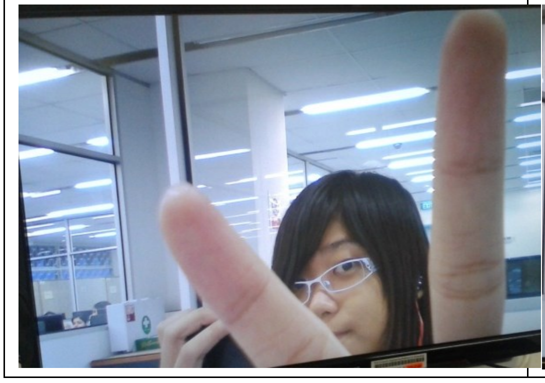
In this experiment, we mainly applied pipelining and signal aligning techniques to process the input signal, and applied conversion of colourspaces and some given logics to help produce the special visual effects.

During pipelining, we could see how much it can improve the whole project's performance. However, too much pipelining may seem redundant and cause aligning problems if handled carelessly.

This lab is a good practice to know how pipelining and signal aligning works in real projects. And through this experiment I knew more about how to arrange different logic modules together. Such as *rgbyuv* module to provide pipelining, *delay_line* module to add signal aligning, and *pixsel* module to process signals.

5. Reference: Lab Manual

6. Appendix: Special effects achieved

Case 1: 	Case 2: 
Case 3: 	Case 4: 
Case 5: 	Case 6: 
Case 7: 	Case 8: