CE2003 Laboratory 4

Lu Shengliang

SLU001

School of Computer Engineering

Nanyang Technological University

SLU001@e.ntu.edu.sg

November 3, 2013

Abstract

The aim of this lab is to build a project to process pixel input from a camera and then output on the screen after a series of process in real time. Several stages of pipelines were employed during this lab session to increase clock frequency, which results a better performance than non-pipelined procedure.

1 Introduction

1.1 Design Modules

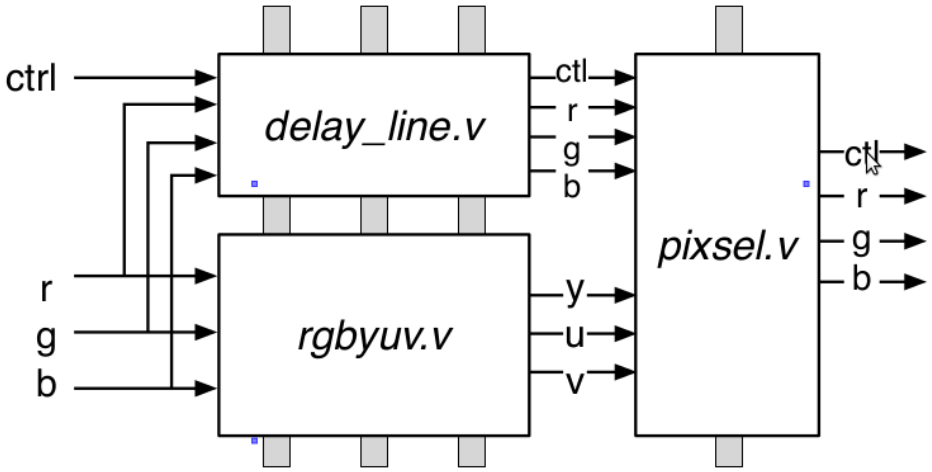

Figure 1: Design Diagram

There are 3 modules in this design. They are ***delay_line***, ***rgbyuv*** and ***pixsel***. Basically, ***rgbyuv*** is used to convert RGB signals from a camera to YUV; ***Delay_line*** module is used to align signal in order to synchronize related input and control signal; Module ***pixsel*** is used to select wanted signal from the previous two module according to control signal, output it to the top module, and then display on the screen.

1.2 Structure of the rest of the paper

The rest of the paper first describes the verilog implementations of ***Task 1: rgbyuv Module*** in *Section 2*, ***Task 2: delay_line Module*** in *Section 3*; And *Section 4* presents the experimental results ***pixels*** module's selections and color effects. Lastly, *Section 5* presents our conclusions and discussions.

2 Task 1: rgbyuv Module

2.1 Verilog Code rgbyuv.v Non-pipelined

```
ry<=((RY_COEF * red_r + GY_COEF * grn_r + BY_COEF * blu_r)>>>8)+16;
ru<=((RU_COEF * red_r + GU_COEF * grn_r + BU_COEF * blu_r)>>>8)+128;
rv<=((RV_COEF * red_r + GV_COEF * grn_r + BV_COEF * blu_r)>>>8)+128;
```

Synthesize this top module and check the synthesis report. Under the synthesis report, there is a device utilization summary. It shows that, 9 DSP blocks are in utilization.

Number of bonded IOBs	80
Number of BUFG/BUFGCTRLs	1
Number of DSP48A1s	9

Figure 2: DSP Blocks

In the synthesis report, the maximum frequency is 77.873MHz.

```
Minimum period: 12.841ns (Maximum Frequency: 77.873MHz)
Minimum input arrival time before clock: 2.957ns
Maximum output required time after clock: 3.597ns
Maximum combinational path delay: No path found
```

Figure 3: Frequency 77.873MHz

2.2 Verilog Code rgbyuv.v with 2-stage pipelines

If we simply add one more stage, which will result in a 2-stage pipelined computation steps. The pipeline diagram is shown below.

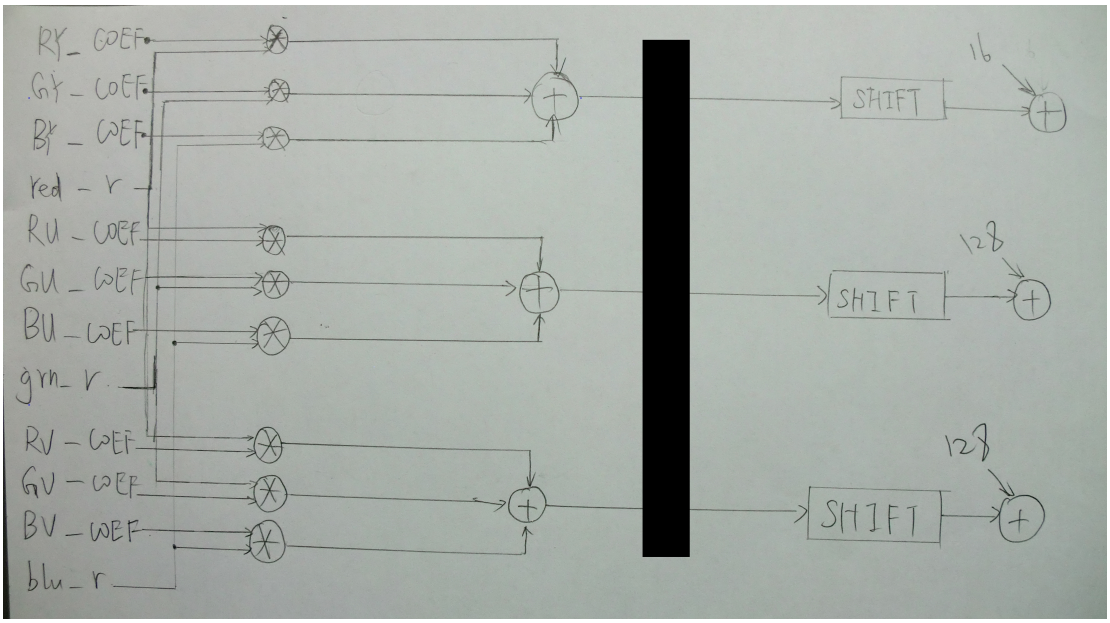

Figure 4: 2-stage pipeline

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

After registers *ry, ru, rv* inserted in the computation procedure, which split the step into a 2-stage pipelines like what is shown in Figure 4. In the synthesis report, the maximum frequency is 96.034MHz.

```
Minimum period: 10.413ns (Maximum Frequency: 96.034MHz)
Minimum input arrival time before clock: 3.428ns
Maximum output required time after clock: 3.597ns
Maximum combinational path delay: No path found
```

Figure 5: Frequency 96.034MHz

Compared to the previous two, the last implementation, which has 3 pipeline stages, is much better and manage to achieve 108MHz.

2.3 Verilog Code rgbyuv.v with 3-stage pipelines

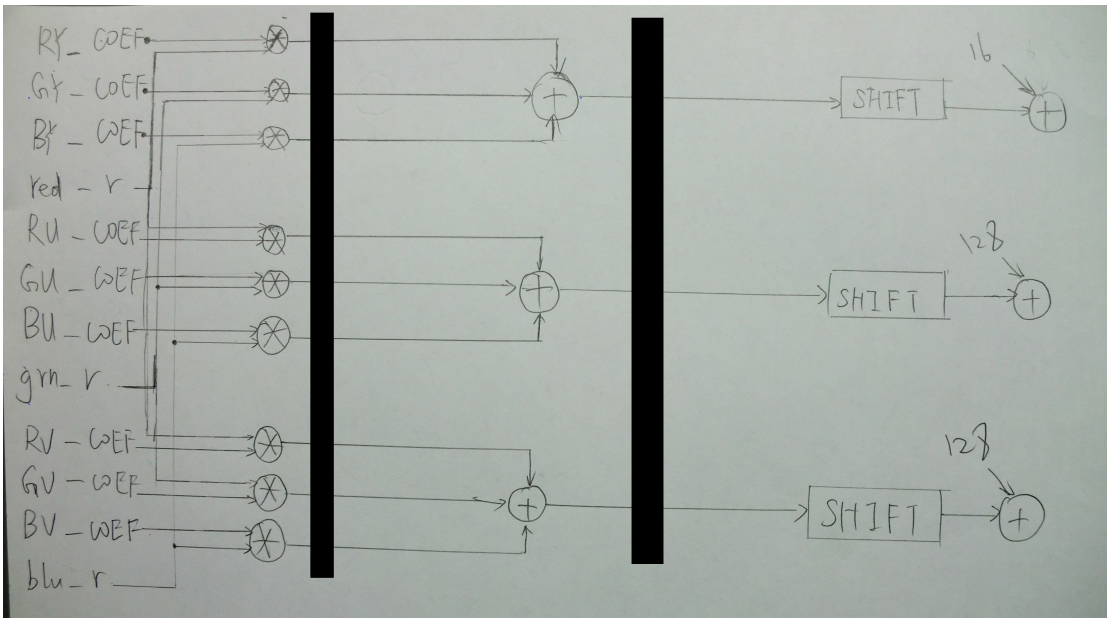

Figure 6: 3-stage pipeline

The multiplications part is the critical path for the whole pipelines among previous ***rgbyuv.v*** implementations. So they are mostly likely to be splitted and it is supposed to give us a good result.

```
ry1 <= RY_COEF * red_r;
ry2 <= GY_COEF * grn_r;
ry3 <= BY_COEF * blu_r;
ru1 <= RU_COEF * red_r;
ru2 <= GU_COEF * grn_r;
ru3 <= BU_COEF * blu_r;
rv1 <= RV_COEF * red_r;
rv2 <= GV_COEF * grn_r;
rv3 <= BV_COEF * blu_r;
ry <= ry1 + ry2 + ry3;
ru <= ru1 + ru2 + ru3;
rv <= rv1 + rv2 + rv3;
o_y <= (ry >>> 8) + 16;
o_u <= (ru >>> 8) + 128;
o_v <= (rv >>> 8) + 128;
```

The critical path was splitted into 3 similar part, which results in a high frequency achievement.
This time, the frequency is 209.249MHz.

```
Minimum period: 4.779ns (Maximum Frequency: 209.249MHz)
Minimum input arrival time before clock: 4.432ns
Maximum output required time after clock: 3.597ns
Maximum combinational path delay: No path found
```

Figure 7: Frequency 209.249MHz

One last improvement, if we put all of the operations into pipelines, the maximum frequency will reach about 456MHz. However, too many pipeline stages will cause more complicated signal alignment in next section, which is *section 3*.

```
Minimum period: 2.190ns (Maximum Frequency: 456.621MHz)
Minimum input arrival time before clock: 4.460ns
Maximum output required time after clock: 3.597ns
Maximum combinational path delay: No path found
```

Figure 8: Frequency 456.621MHz

2.4 skin pixel detection with a 4-stage pipeline

```
      //add one more stage
      p_y <= (ry >>> 8) + 16;
      p_u <= (ru >>> 8) + 128;
      p_v <= (rv >>> 8) + 128;
      o_y <= p_y; o_u <= p_u; o_v <= p_v;
    end
  end // always @ (posedge clk)

  always @(posedge clk) begin
    if (rst) begin
      skind <= 1'b0;
    end else begin
      if (73 <= p_u && p_u <= 122 && 132 <= p_v && p_v <= 173) begin
        skind <= 1'b1;
      end else begin
        skind <= 1'b0;
      end
    end
  end // always @ (posedge clk)
```

If a pixel is detected having a skin-like color by using YUV analysis, a 1-bit register *skind* will be asserted to 1, otherwise 0.

3 Task 2: delay_line Module

3.1 Verilog Code delay_line.v

```
always @(posedge clk) begin
    if(rst) begin
      in_r_p1 <=8'd0;
      in_r_p2 <= 8'd0;
      in_r_p3 <= 8'd0;
      in_r_p4 <= 8'd0;
      out_r <= 8'd0;

      in_g_p1 <=8'd0;
      in_g_p2 <= 8'd0;
      in_g_p3 <= 8'd0;
      in_g_p4 <= 8'd0;
      out_g <= 8'd0;

      in_b_p1 <=8'd0;
      in_b_p2 <= 8'd0;
      in_b_p3 <= 8'd0;
      in_b_p4 <= 8'd0;
      out_b <= 8'd0;

      in_c_p1 <=8'd0;
      in_c_p2 <= 8'd0;
      in_c_p3 <= 8'd0;
      in_c_p4 <= 8'd0;
      out_c <= 8'd0;
    end else begin
      in_r_p1 <= in_r;
      in_r_p2 <= in_r_p1;
      in_r_p3 <= in_r_p2;
      in_r_p4 <= in_r_p3;
      out_r   <= in_r_p3;

      in_g_p1 <= in_g;
      in_g_p2 <= in_g_p1;
      in_g_p3 <= in_g_p2;
      in_g_p4 <= in_g_p3;
      out_g   <= in_g_p4;

      in_b_p1 <= in_b;
      in_b_p2 <= in_b_p1;
      in_b_p3 <= in_b_p2;
      in_b_p4 <= in_b_p3;
      out_b   <= in_b_p4;

      in_c_p1 <= in_c;
      in_c_p2 <= in_c_p1;
      in_c_p3 <= in_c_p2;
      in_c_p4 <= in_c_p3;
      out_c   <= in_c_p4;
    end // else: !if(rst)
  end // always @ (posedge clk)
endmodule // delay_line
```

As we can see from the module name, ***delayline*** is used to delay the input signal by using pipelines. Because we are currently using a 5-stage pipelined ***rgbyuv.v***, ***delayline*** are supposed to have the same pipelining stages as it.

4 Task 3 & 4: Logically Display

Task 3 and task 4 are combined together, because task 4 is roughly improvement of task 3 with more pixel operations.

4.1 Verilog Code pixsel.v

```
  always @(posedge clk) begin
    if(rst) begin
      out_r <= 8'd0;
      out_g <= 8'd0;
      out_b <= 8'd0;
      out_ctrl <= 3'd0;
```

```
      end else begin
        case(in_swt)
          1 : begin
            out_r <= in_y; out_g <= in_y; out_b <= in_y;
          end
          2 : begin
            out_r <= in_u; out_g <= in_u; out_b <= in_u;
          end
          4 : begin
            out_r <= in_v; out_g <= in_v; out_b <= in_v;
          end
          /*8 : begin
            if (in_skin) begin
              out_r <= in_y; out_g <= in_y; out_b <= in_y;
            end else begin
              out_r <= 1'b0; out_g <= 1'b0; out_b <= 1'b0;
            end
          end*/
          //display with maximum green
          8 : begin
            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
          end
          //display with maximum red
          16 : begin
            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
          end
          //display with maximum blue
          32 : begin
            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
          end
          //display with maximum red, green and blue
          64 : begin
            if (in_skin) begin
              out_r <= max_r; out_g <= max_g; out_b <= max_b;
            end else begin
              out_r <= in_r; out_g <= in_g; out_b <= in_b;
            end
          end
          //display original skin color. otherwise max r & g & b
          128 : begin
            if (in_skin) begin
              out_r <= in_r; out_g <= in_g; out_b <= in_b;
            end else begin
              out_r <= max_r; out_g <= max_g; out_b <= max_b;
            end
          end
          //for fun
          255 : begin
            if (in_skin) begin
              out_r <= in_r; out_g <= in_g; out_b <= in_b;
            end else begin
              out_r <= min_r; out_g <= min_g; out_b <= min_b;
            end
          end
```

```
            default : begin
              out_r <= in_r; out_g <= in_g; out_b <= in_b;
            end
          endcase // case (in_swt)
          out_ctrl <= in_c;
        end // else: !if(rst)
    end // always @ (posedge clk)
```

A one-hot strategy was employed to control the pixel selections. More details are given below. ***Case 1, 2, 4:***

The screen turned black and white in case 1, because there is no color output from *pixsel.v*, except *y*, which is the luminance or brightness of a pixel; The screen turned inverted darkness and grey in switch 2 and turning palely grey in switch 4.

Case 8, 16, 32:

In task 3, fpga is supposed to show those pixels detected as skin ones. The others will be blocked, which results that only skin parts will be shown on the screen.

But the result will not be very good: there will be many incorrect outputs, such like fluorescent lamp, white wall, since we are currently using a basic skin color detection algorithm. In task 4, instead of blocking other pixels and displaying skin ones, fpga is supposed to change skin colors by replace original pixels' data with different colors and keep the other pixels unchanged.

```
  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;
```

```
out_r <= min_r; out_g <= max_g; out_b <= min_b;
```

If max_g was used to replace out_g, the skin pixels will be replaced by green pixels.

```
out_r <= max_r; out_g <= min_g; out_b <= min_b;
```

If max_r was used to replace out_r, the skin pixels will be replaced by red pixels.

```
out_r <= min_r; out_g <= min_g; out_b <= max_b;
```

If max_b was used to replace out_b, the skin pixels will be replaced by blue pixels.

Case 64, 128:

```
if (in_skin)
  out_r <= max_r; out_g <= max_g; out_b <= max_b;
else
  out_r <= in_r; out_g <= in_g; out_b <= in_b;
```

If all out_ were replaced by max_, the skin pixels will become pale instead of original flesh or pink color.

```
if (in_skin)
  out_r <= in_r; out_g <= in_g; out_b <= in_b;
else
  out_r <= max_r; out_g <= max_g; out_b <= max_b;
```

Doing fun, after exchange these two line unblocking assignment, the skin pixels went out directly, and other pixels become greyer than before. But there is a side effect: some pixels on skin may not be detected as a skin pixel, which results that the whole skin looks darker than before.

Case 255:

```
if (in_skin)
  out_r <= in_r; out_g <= in_g; out_b <= in_b;
else
  out_r <= min_r; out_g <= min_g; out_b <= min_b;
```

This time, all the black color shined, and skin pixels were kept unchanged. It because we set, if a pixel is not a skin pixel, then it will be replaced by min_r, min_g and min_b.

5 Conclusions and Future Work

In this lab session, pipelining and signal aligning techniques were employed to process input signals from a camera. In addition, pixels selection and logical color conversion were applied to produce several special visual effects.

5.1 *Problems* occurred in our approach

From the Figure 1 above, we can see that the output signals (ctl, r, g, b) of ***delay_line*** module need to be synchronous with the output signals (y, u, v) of ***rgbyuv*** module. Sine a 4-stage pipeline was employed in ***rgbyuv*** module, the

delay_line module is also required to have a 4-stage pipeline.
If the number of stages in ***delay_line*** module is different from ***rgbyuv***, fpge would not generate video output to screen.
Because all the multiplications at the first stage in ***rgbyuv.v*** will be the critical path for the whole pipelines. So if the most slow part in a pipeline is separated like the code above, it will result in a significant improvement on maximum frequency.

5.2 Available *Improvement*

10 kinds of color effects were conducted based on skin color detection and YUV colorspace. Actually, more achievements are available. For example, one way is that we can change the skin color detect block to other color detect, and even more than color detection.
Another way is that instead of using one-hot strategy, which will only have 8 features, we can actually add more features by changing the output with different color combinations like what was done in this report.
A pipelined ***rgbyuv.v*** at a frequency of 456MHz was conducted during taks 1. But it was not employed during the synthesis procedure, because a pipeline with too many stages is redundant for this lab implementation.

6 Appendix

Please ignore the time stamp on the right corner of each picture. There was something wrong with my camera.

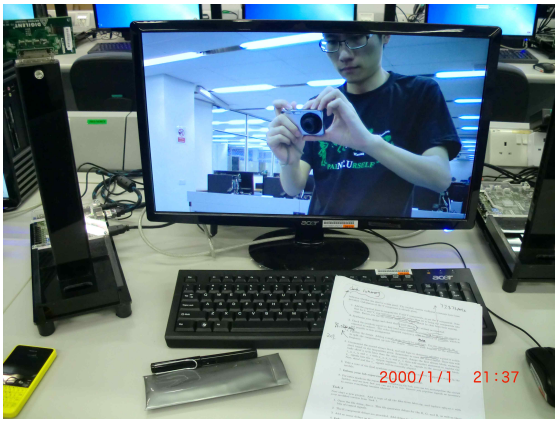

Figure 9: Case default

Figure 11: Case 2

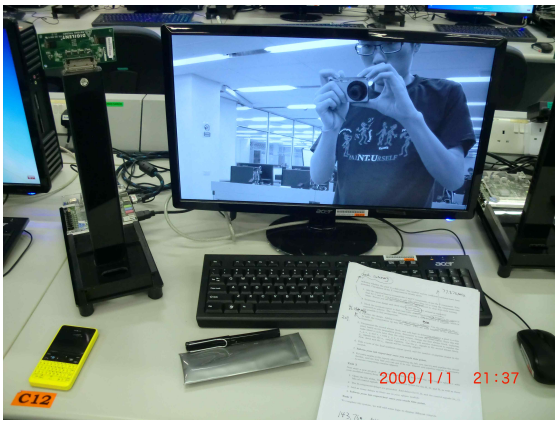

Figure 10: Case 1

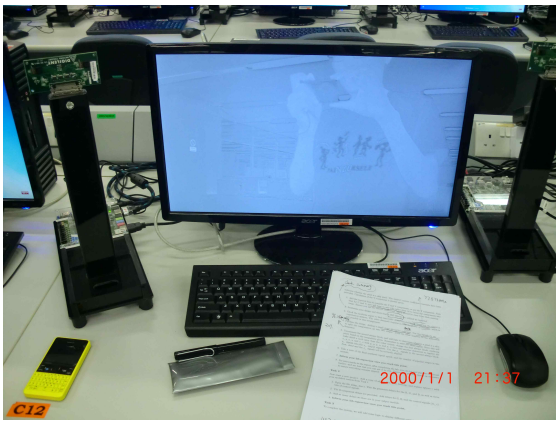

Figure 12: Case 4

Figure 13: Case 8

Figure 14: Case 16

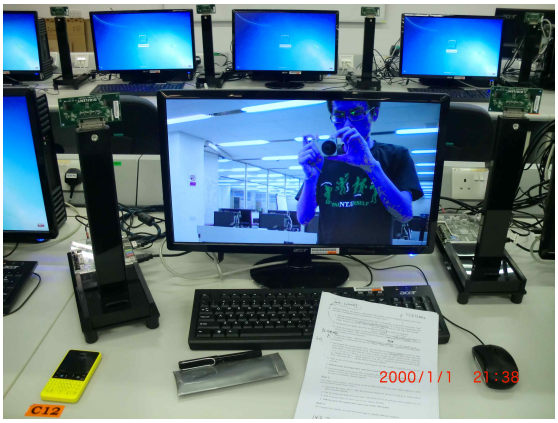

Figure 15: Case 32

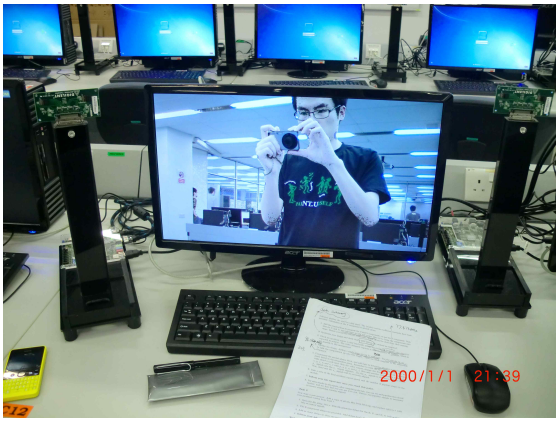

Figure 16: Case 64

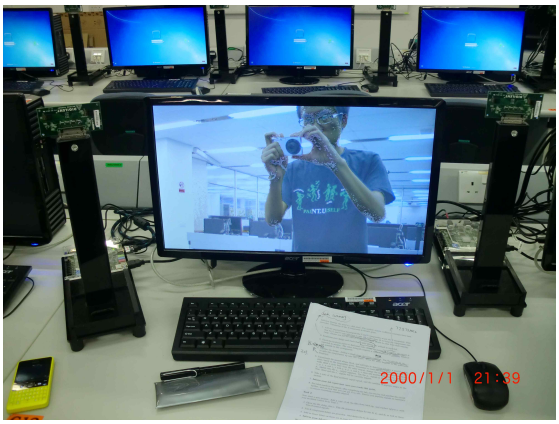

Figure 17: Case 128

Figure 18: Case 255