PULSE OXIMETRY DESIGN IN THE UMC65LL TECHNOLOGY REPORT

Submitted by:
Tran Minh Quang
Matrikel-Nr: 2697987
Ruolin Huang
Matrikel-Nr: 2590495
Supervision:
M.Sc. Dominic Korner

TECHNISCHE UNIVERSITÄT DARMSTADT

HDL Lab
Integrated Electronic Systems Lab

Contents

1 Introduction . . . 2
1.1 Pulse Oximetry . . . 2
1.2 Outline . . . 2
2 Ideal 8-bit ADC in Verilog-A . . . 3
2.1 Working principle of 8-bits ADC . . . 3
2.2 Implementation in verilog-A . . . 3
2.3 Testbench and simulation . . . 4
3 Controller and FIR filter . . . 5
3.1 Controller . . . 5
3.1.1 Top-level finite state machine . . . 5
3.1.2 DC_Comp approximation state . . . 6
3.1.3 DC_Comp adjustment state . . . 7
3.1.4 PGA_Gain state . . . 8
3.1.5 LED toggle state . . . 8
3.2 FIR filter . . . 9
3.2.1 ADC outputs seperation . . . 9
3.2.2 Working principle of FIR filter . . . 9
3.2.3 FIR filter optimization . . . 9
3.3 Testbench and simulation . . . 11
3.3.1 Fingerclip_Model improvement . . . 11
3.3.2 FIR filter testbench modelsim . . . 11
3.3.3 Simulation result in Modelsim . . . 11
3.3.4 Simulation result in Cadence . . . 11
4 Synthesize and final layout . . . 12
4.1 Maximal synthesizable frequency with Synopsis Design Vision . . . 12
4.2 Power consumption at the nominal frequency . . . 14
4.3 Minimal layout area with Encounter . . . 14
5 Conclusion . . . 16

References 17

Appendix 18

1 Introduction

1.1 Pulse Oximetry

The design goal of this lab is to implement the digital functions of the pulse oximetry method. Pulse Oximetry provides a non-invasive way to measure oxygen saturation and pulse[2]. There are 2 LEDs in the analog part, which generate different wavelengths of light (660 nm and 940 nm). The light goes through human tissue and is collected by a photodiode. Because the absorption of blood is heavily dependent on the amount of oxygen which is carried by hemoglobin in the blood[2], we can get the oxygen saturation according to the received light of the photodiode.

The working principle of the whole project can be divided into three parts: Analog Frontend, 8-bits Verilog-A ADC, Controller and Filter. There are 2 LEDs in Analog Frontend which are responsible for generating different wavelengths of light. The light goes through human tissue and is transformed into electrical signal by a photodiode. Then this signal will be amplified by amplifiers. As a result, we can get a voltage Vppg as an output of Analog Frontend. This analog signal goes through the ADC part and will be transformed into 8 bits digital signal. After that, this signal will go through the filter and we can get a perfect signal without high frequency noise at the output of FIR filters.

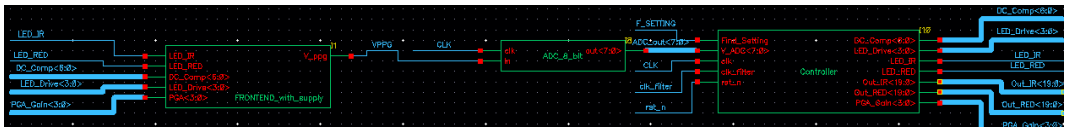

Figure 1: schematic of the whole lab

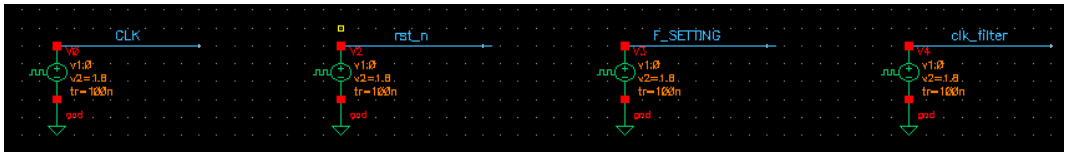

Figure 2: power supplies of the whole lab

1.2 Outline

In section 2 we discuss an approach to building and simulating an 8-bit ADC in Verilog-A. In Part 3 we focus on describing the controller algorithm and the filter optimization in terms of area and critical path. In chapter 3 we present the simulation results that we obtained from Modelsim and Cadence software. In chapter 4 we present the results on power consumption, maximal frequency as well as the minimal area of the circuit, which we obtain through the tools Synopsis design and Encounter. Finally, we conclude the thesis and summarize the work.

2 Ideal 8-bit ADC in Verilog-A

2.1 Working principle of 8-bits ADC

In this part we implement an ADC Converter, which can transform the analog signal Vppg into 8-bits digital signal. For this ADC converter we need a clock which has a frequency of 1kHz. The schematic of the ADC is shown in figure 3.

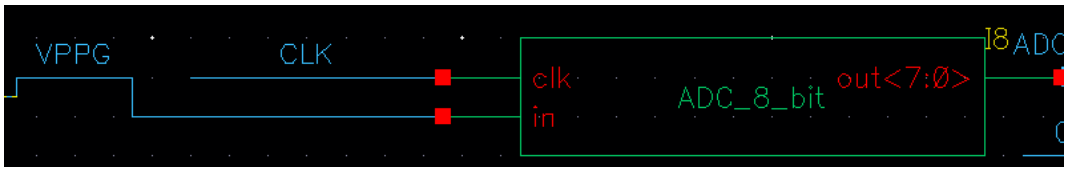

Figure 3: schematic of 8-bits ADC

The idea of this ADC is that we will find the correct value 0 or 1 for each bit in the byte ADC output from MSB to LSB. Firstly, we set the threshold to 0.9 V, which is half of the fullscale voltage. Then, the input voltage is compared to the threshold. If the input voltage is larger than threshold, the current MSB is set to 1, otherwise, it is set to 0. After finding the current MSB bit, the input voltage is reduced by threshold voltage if it is larger than the threshold voltage to clear the current MSB bit. The next step is to multiply the input voltage by 2 to shift the byte being searched, which corresponds to the input voltage, 1 bit to the left. Now the bit next to the last MSB bit becomes the current MSB bit. The process repeats until the LSB bit is shifted to MSB position. After finding the LSB bit successfully, the result is put to the output of the ADC.

2.2 Implementation in verilog-A

Our codes are shown below. This code is based on the implementation of a simple A/D converter in the script of the lecture Computer Aided Design for System on Chips [1]

```
module ADC_8_bit(out, in, clk);
    input in, clk;
    output [7:0] out;
    voltage in, clk;
    voltage [7:0] out;
    parameter real fullscale = 1.8;
    parameter real delay_ = 0, trise = 10n, tfall = 10n;
    parameter real clk_vth = 0.9;
    parameter real out_high = 1.8, out_low = 0 from (-inf:out_high);

    real sample,thresh;
    real result[7:0];
    integer i;

    analog begin
        @(cross(V(clk)-clk_vth, +1))
        begin
            sample = V(in);
            thresh = fullscale/2;
            for(i=7;i>=0;ai=i-1)    begin
                if (sample > thresh)    begin
                    result[i] = out_high;
                    sample = sample - thresh;
                end
                else result[i] = out_low;
                    sample = 2*sample;
                end
            end
            generate i (7, 0) begin
            V(out[i]) <+ transition(result[i], delay_, trise, tfall);
        end
    end
endmodule
```

Listing 1: Idea 8-bit ADC

2.3 Testbench and simulation

We create a testbench for ADC as shown in figure 4

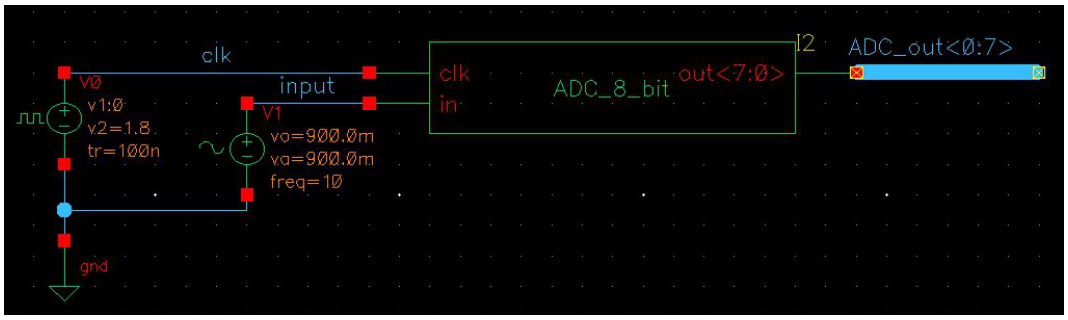

Figure 4: ADC testbench

The input of the ADC is a sine wave with amplitude 0.9 V, offset voltage 0.9 V and frequency 10 Hz. The simulation result from Cadence is shown in figure 5. We can see from the picture that, the output of ADC is almost the same as the input signal. The ADC converter works very well.

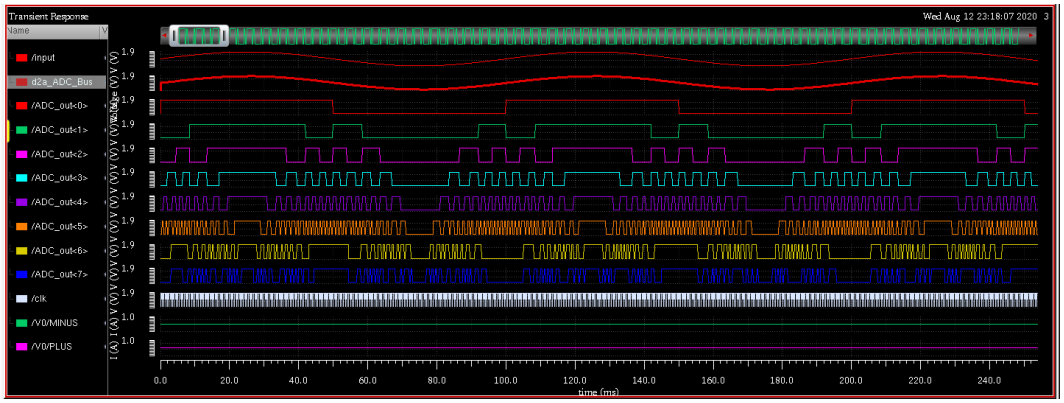

Figure 5: 8-bits ADC simulation result with a sine wave testbench

3 Controller and FIR filter

3.1 Controller

3.1.1 Top-level finite state machine

The goal of the controller is to find a good and fast operating setting for the analog frontend. To find such a setting, the algorithm first finds DC_Comp and then PGA_Gain for the red LED and then finds DC_Comp and PGA_Gain for the infrared LED. After finding the required parameters for both LEDs, it will start flashing 2 LEDs alternately at 100 Hz. The finite state machine is shown in figure 6

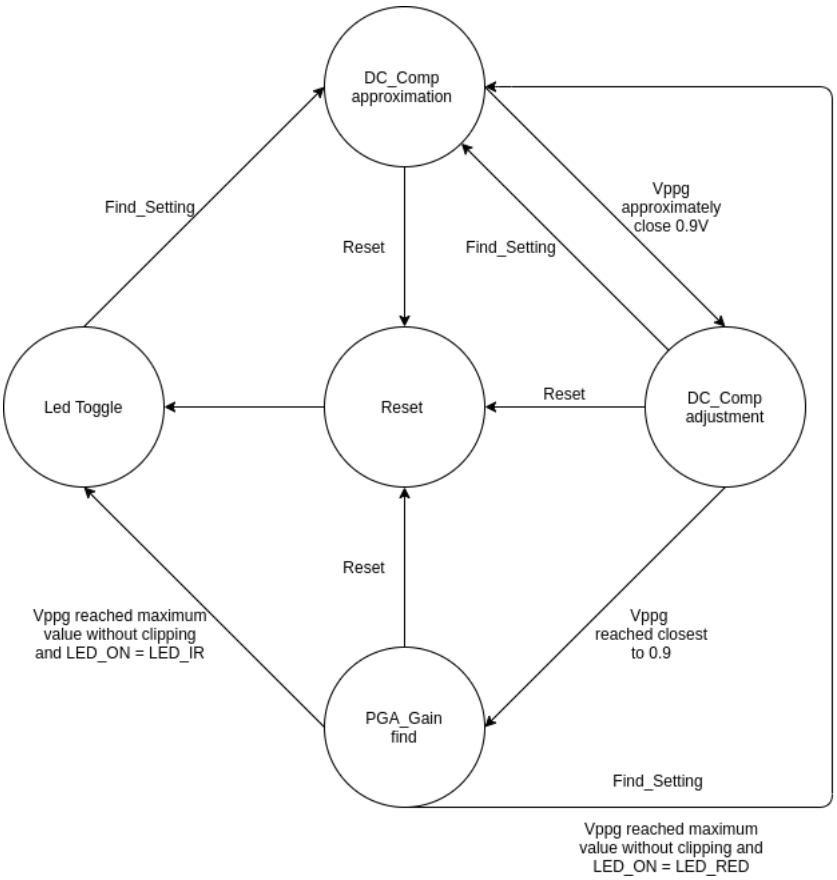

Figure 6: Top-level controller finite state machine

This state machine consists of 5 main states that are Reset, DC_Comp approximation, DC_Comp adjustment, PGA_Gain find and Led toggle . In the reset state, all registers will be reset to zero and the current state immediately transit to the Led toggle state. When there is a rising edge signal of Find_Setting , the process of finding parameters will start. This process starts at DC_Comp approximation state. In this state, the algorithm will try to find quickly the value of DC compensation, making Vppg close to 0.9. Noticeably, when the PGA_Gain is equal to 0, the Vppg signal always fluctuates around the average value corresponding to DC_Comp with an amplitude of about 300 mV, which makes the finding DC compensation have large error compared to the actual desired value. To solve this, a second state was introduced, DC_Comp adjustment. In this state we will not change DC_Comp too quickly, but only change every half cycle of Vppg's oscillation. This slow search return a more accurate result, but take plenty of time. Thanks to the fact that DC compensation is already close to the value to look for after the DC_Comp approximation state, the slow finding process doesn't take too long. After finding DC_Comp, the algorithm will go into PGA_Gain state to find the value of PGA_Gain. Then the process is repeated for the infrared LED. After completing the determination of parameters for both LEDs, the algorithm will jump into the LED toggle state to flash the LEDs in a certain frequency with the found corresponding parameters.

The state DC_Comp approximation, DC_Comp adjustment, PGA_Gain find and Led toggle will be detailed in the next sections.

3.1.2 DC_Comp approximation state

This state tries to find the approximate value of DC_Comp as quickly as possible. First of all, there are a few concepts that need to be clarified: the most right 1 bit of DC_Comp and the desired ADC interval. If DC_Comp is currently 7'b000110 then the most right 1 bit is the bit at position 1 in the bit sequence. In addition, we use two values ADC_half_max and ADC_half_min to define a desired interval around 0.9 V. When changing DC_Comp if the ADC output signal is in this range, the search for DC_Comp is completed. The idea to quickly find DC_Comp is that we will find the exact value 0 or 1 for each bit from left to right. DC_Comp will be initialized as 7b'1000000. If ADC output is greater than ADC_half_max then DC_Comp will be increased to decrease ADC output. This increment is performed by setting the bit next to the current most right 1 bit to 1. In this case, the result will be 7'b1100000. Otherwise, if ADC output is less than ADC_half_min, DC_Comp will be decreased to increase ADC output. This reduction will be performed by setting the current most right 1 bit to 0 and its next bit to 1. In this case, the result would be 7'b0100000. The process will repeat until we find the value DC_Comp making the ADC output signal place within the desired range. After finding the approximate DC_Comp, the algorithm will jump into the DC_Comp adjustment state to tweak the DC_Comp value more accurately. The flowchart of DC_Comp is shown in figure 7

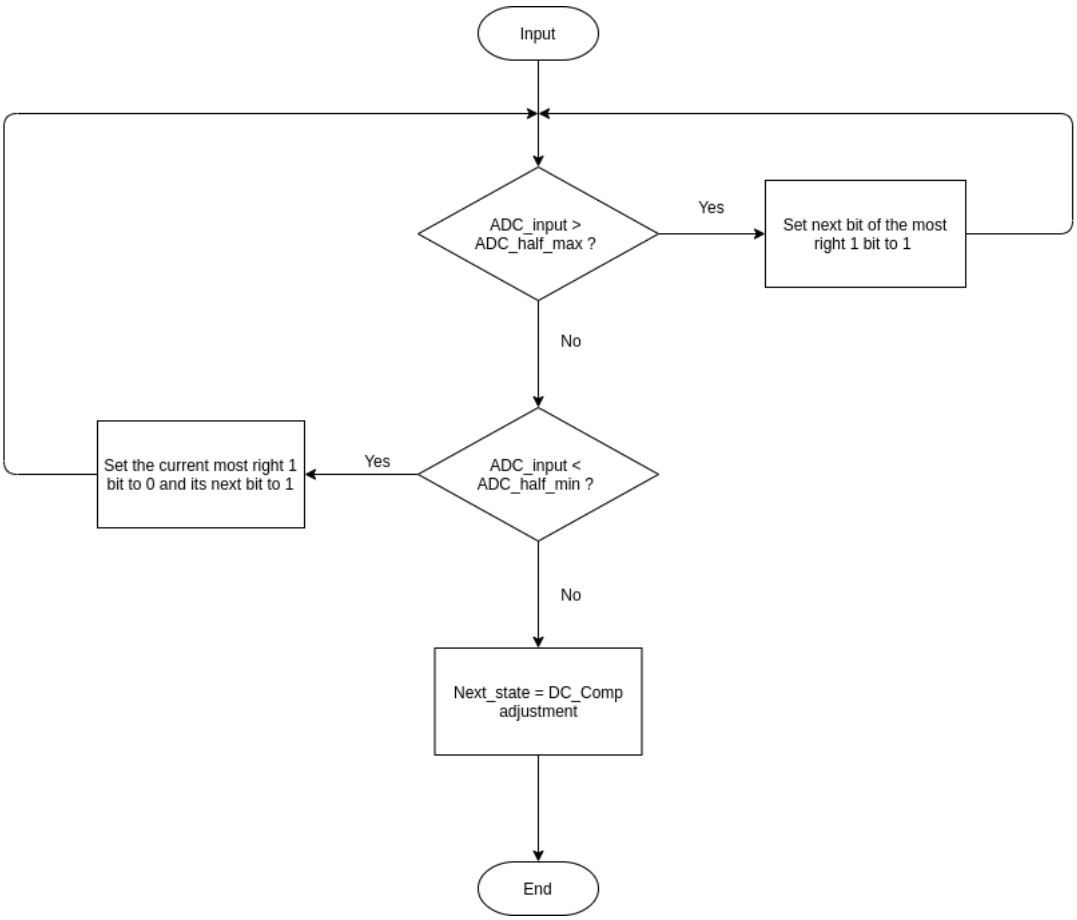

Figure 7: DC_Comp approximation state machine

3.1.3 DC_Comp adjustment state

The Vppg signal always fluctuates with an amplitude of about 300 mV when the gain is zero. Because of this, the value of DC_Comp found in the DC_Comp approximation state will often be inaccurate. To address this, we will wait a 0.5 ms interval. During this time, the algorithm will continuously update the maximum and minimum values of the ADC signal. After 0.5 ms, the mid-value of the ADC output will be calculated and compared with ADC_upper_bound and ADC_under_bound. Then it slowly increases and decreases the DC_Comp value found from the DC_Comp approximation state to let the average of the ADC output stay in the desired regions. In this way, the influence of the fluctuation is greatly reduced. After the fine-tuning is complete, the DC_Comp value will be saved in the register corresponding to the current LED and jump to the next state to find PGA_Gain.

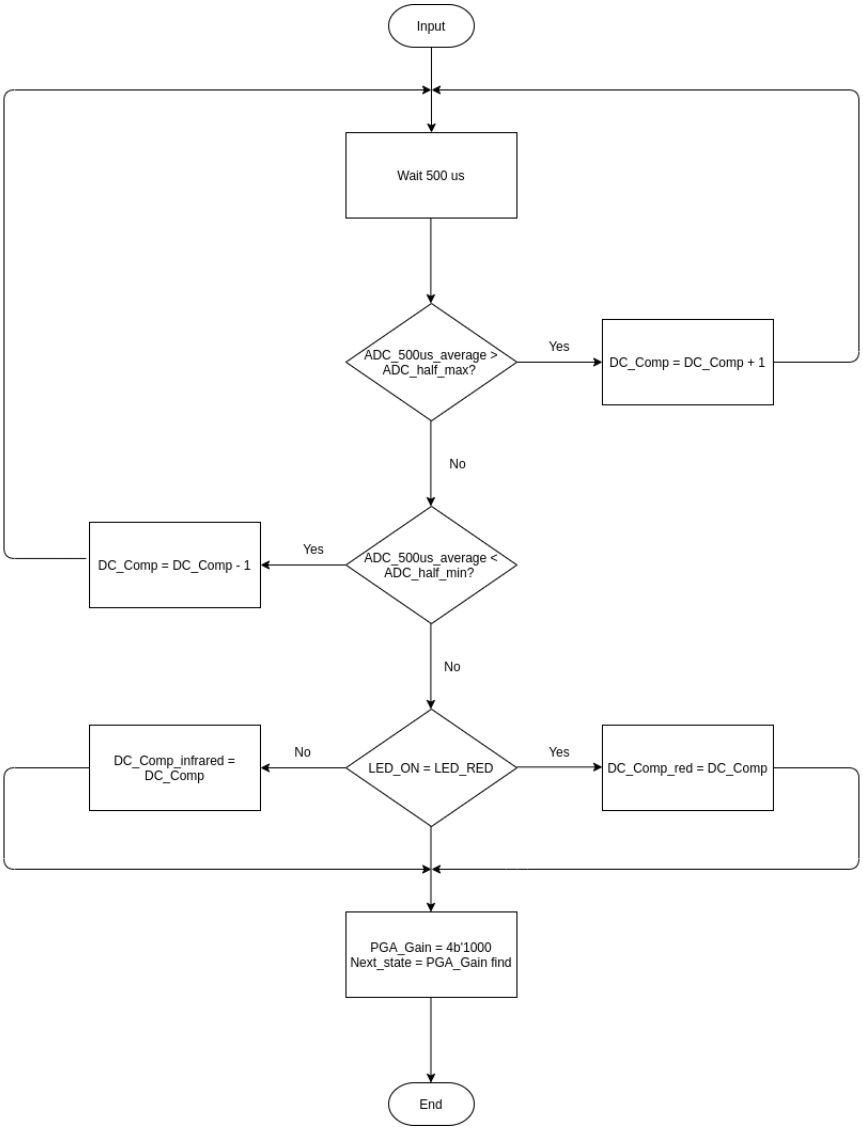

Figure 8: DC_Comp approximate state machine

3.1.4 PGA_Gain state

Since the oscillation period of the Vppg signal is 1 ms we will wait 1 ms every time we want to change the PGA_gain. While waiting, the maximum and minimum value of the Vppg signal in one oscillation cycle will be stored. These maximum and minimum values will be used to compare the desired maximum and minimum ADC output range, which is defined by the boundaries ADC_max_1, ADC_max_2, ADC_min_1, and ADC_min_2 respectively, to check if the Vppg reaches the maximum value without clipping at the ground or supply rail. The change of PGA_Gain is carried out the same as finding DC_Comp in DC_Comp approximation state. When the ADC output is in either of these ranges, the PGA_Gain search ends. After finding the gain, if the current LED is the red LED, the algorithm will continue to find the parameters for the infrared LED. If the current LED is IR LED then the algorithm will jump into LED toggle state. The flowchart of PGA_Gain is shown in figure 9

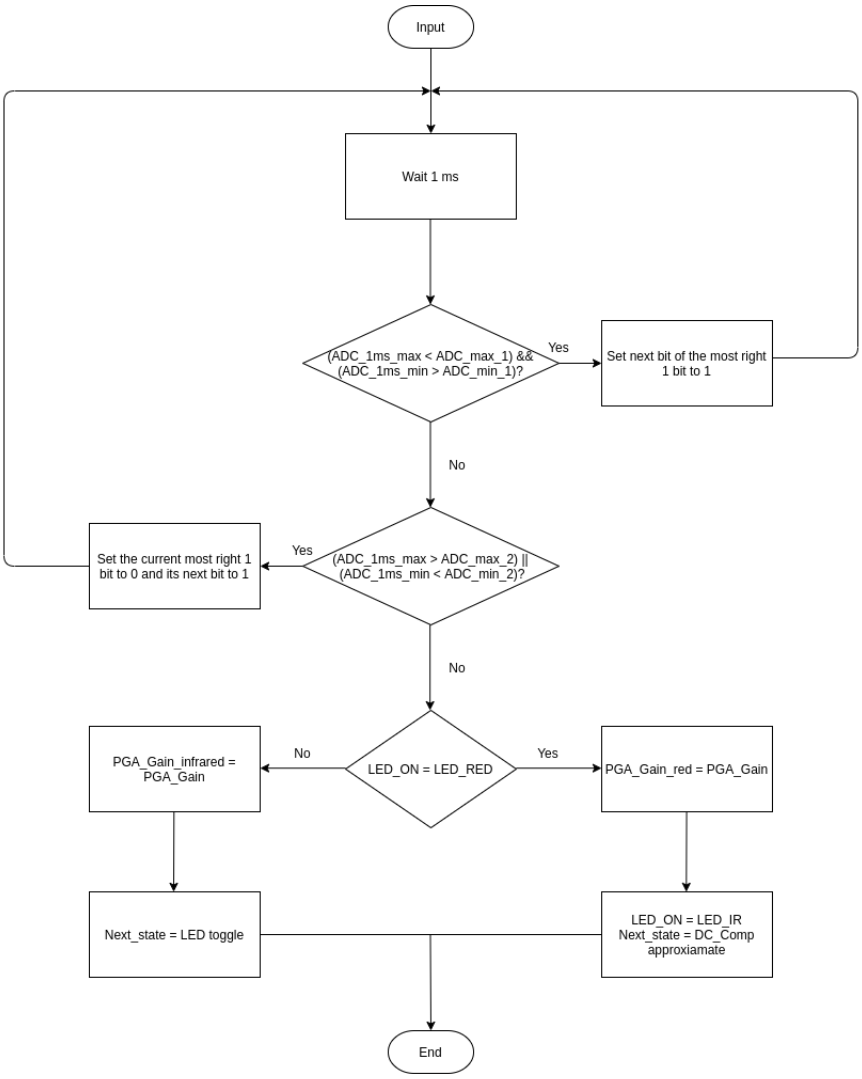

Figure 9: PGA_Gain find flowchart

3.1.5 LED toggle state

In this state, two LEDs will flash alternately at a frequency of 100 Hz. When the LED is turned on, its respective DC_Comp and PGA_Gain values will be set to the controller output.

3.2 FIR filter

3.2.1 ADC outputs seperation

Since there are 2 LEDs that flash alternately at 100 Hz, there are two bitstreams corresponding to each LED. We use a simple demultiplexer to separate the original ADC signal into two separate bitstreams. These bitstreams are the input for the FIR filter. The verilog implementation can be found in the Appendix chapter.

3.2.2 Working principle of FIR filter

This filter is used to filter out the high-frequency components of the output ADC signal. In this lab, we will construct an FIR filter with 22 coefficients given in the HDL lab manual. The basic principle of an FIR filter is shown in the figure 10

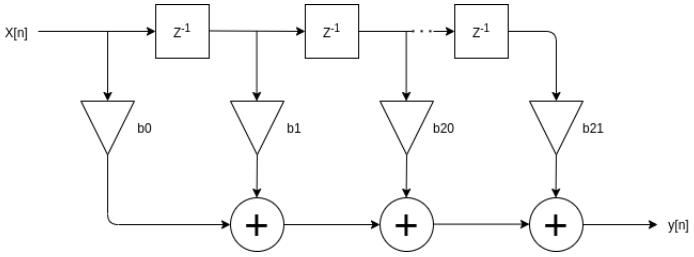

Figure 10: FIR filter working principle

As can be seen from the figure 10, FIR filters require a lot of adders and multipliers. For that reason, if we program this filter according to naive principles as shown in figure 10, we will use a lot of hardware resources. This leads to an increase in the area of the circuit. In addition, having to compute multiple simultaneous multiplication and addition operations in one clock cycle leads to an increase in critical paths and thereby reducing the operating frequency of the circuit. The next section will present some methods to help reduce the area and critical path of the circuit.

3.2.3 FIR filter optimization

To optimize the FIR filter we have 2 main approaches. One is based on symmetry in the filter coefficients and the other is to use an external clock with a high frequency.

For the first approach, we first add the delay taps that have the same coefficients and then multiply with the coefficient as shown in the figure 11. This will halve the number of multipliers and a significant amount of adder.

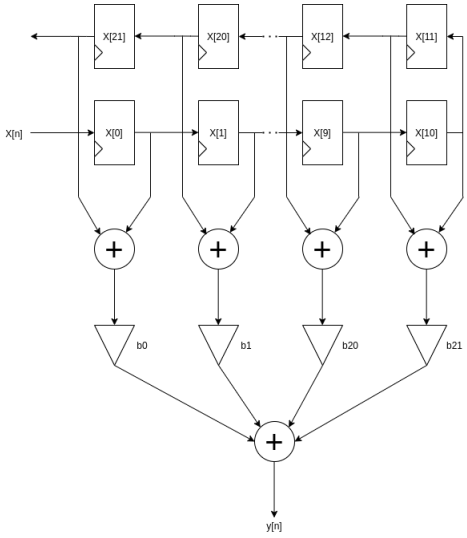

Figure 11: FIR filter optimization base on symmetry of the coefficients

For the second approach, several registers are used to latch the current coefficients, delay taps and results of the calculations will be introduced. In addition, an external clock with high frequency is used, which allows to carry out many calculations in one sampling clock cycle as shown in figure 12 . In this way, the multiplier and adders can be reused and thereby reducing the critical path and area.

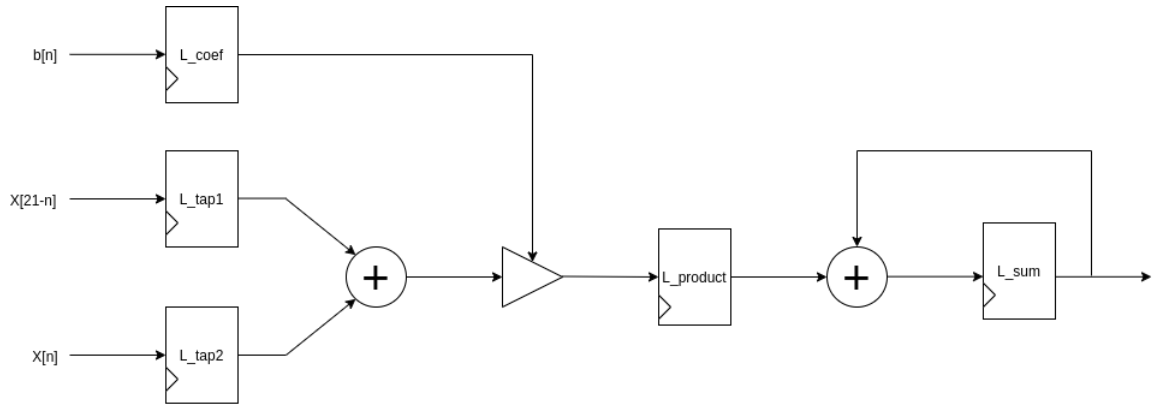

Figure 12: FIR filter with high frequency

As can be seen from figure 12, we need to calculate 10 times to get one output signal. This means we must provide an external clock with the frequency at least 10 times larger than the sampling frequency into the filter. One more step, we can continue to optimize in order to minimize the hardware and critical paths as shown in figure 13.

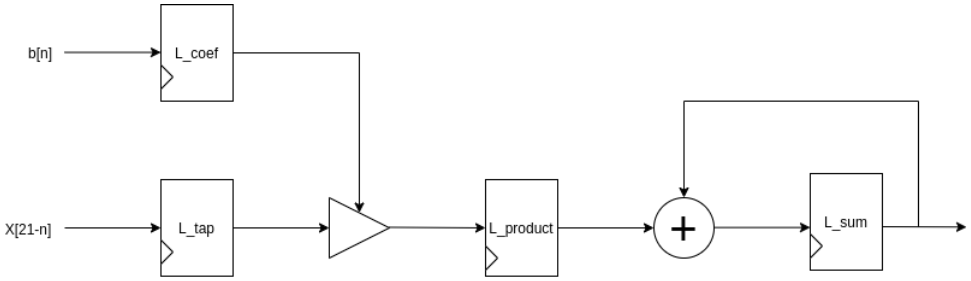

Figure 13: FIR filter with high frequency and minimal critical path

By this way, only 1 multiplier and 1 adder are used. This is the best optimization that we can get. We decided to implement this method in our final design. The only disadvantage is that this FIR filter requires a clock with the frequency at least 22 times larger than the sampling frequency. The Verilog implementation of this FIR filter can be found in the Appendix chapter.

3.3 Testbench and simulation

3.3.1 Fingerclip_Model improvement

Fingerclip_model is used as analog frontend block in Cadence for testing. There is one problem that the output signal of Vppp in Fingerclip_model changes too slow in compared to the analog frontend in Cadence, which lead to different results between the simulations in Modelsim and Cadence. For this reason, we adjusted the conditions of always block in Fingerclip_model, as shown below, to get the same behavior between the software.

```
18 always@(posedge clk or negedge clk or DC_Comp or PGA_Gain)
```

Listing 2: Modify the condition of alwyas block

3.3.2 FIR filter testbench modelsim

We create a testbench for FIR filter which can be found in the Appendix chapter. In this testbench the result of FIR filter ist save to dataout.txt. This result can be used to verify the calculation with the result from matlab or C program.

3.3.3 Simulation result in Modelsim

Here is a picture of the simulation result in Modelsim. We can see that, the whole system works very well. It finds the DC_Comp and PGA_Gain for both of the LEDs. And the signal V_ADC goes through the FIR Filter and becomes the final output Out_IR and Out_RED.

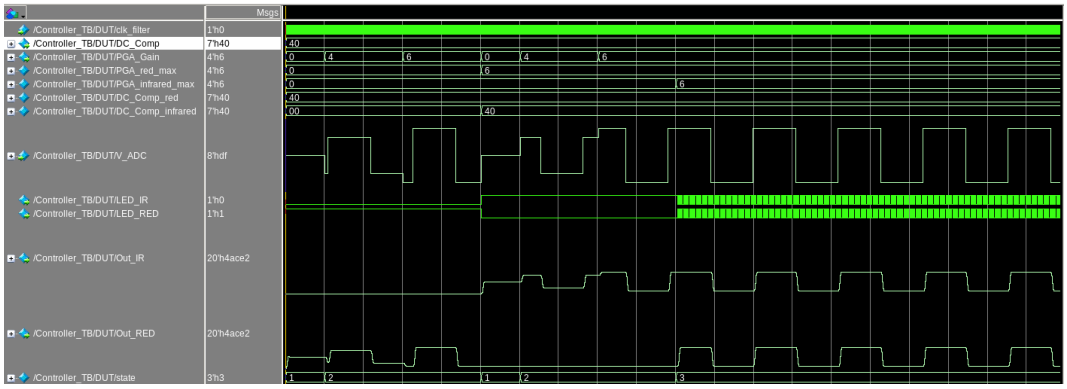

Figure 14: Simulation result in Modelsim

3.3.4 Simulation result in Cadence

We can see from the picture that, the working principle is exactly what we explained before. The controller first find the DC_Comp and PGA_Gain for the red LED, then it find the settings for the infrared LED. In the end the LEDs begin to switch in a certain frequency. The time consumption for finding the settings is about 5.6 seconds. The output signal of the Analog Frontend Vppg will first go through the ADC converter. Then it goes through the FIR filter and is divided into two signals. The final output signals are Out_RED and Out_IR.

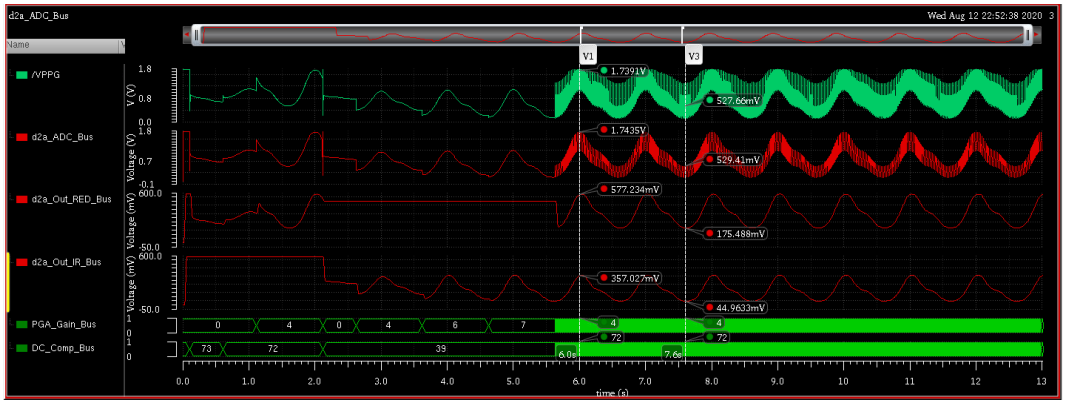

Figure 15: Simulation result in Cadence

After we finish the simulation in ModelSim and Cadence, we can use the Synopsis Design Version to synthesize digital circuits. The smallest period of the clock is 0.8 ns. The report file shows that the slack is 0.00, so it can work properly.

```
# create clock constraint:
create_clock [get_ports clk] -name "CLOCK" -period 0.80
```

Figure 16: setting in constrain.tlc for fastest clk

```
Point                                          Incr       Path
-----------------------------------------------------------------
clock CLOCK (rise edge)                        0.00       0.00
clock network delay (ideal)                    0.00       0.00
count_reg_6_/CK (DFQRM8WA)                     0.00       0.00 r
count_reg_6_/Q (DFQRM8WA)                      0.22       0.22 r
U1303/Z (INVM2W)                               0.10       0.32 f
U1304/Z (NR2M8W)                               0.09       0.41 r
U884/Z (INVM1R)                                0.08       0.48 f
U1322/Z (NR2M4W)                               0.09       0.57 r
U1323/Z (XNR2M6WA)                             0.10       0.67 f
U1198/Z (AO22M8WA)                             0.09       0.76 f
count_reg_7_/D (DFRM8WA)                       0.00       0.76 f
data arrival time                                         0.76

clock CLOCK (rise edge)                        0.80       0.80
clock network delay (ideal)                    0.00       0.80
clock uncertainty                             -0.03       0.77
count_reg_7_/CK (DFRM8WA)                      0.00       0.77 r
library setup time                            -0.01       0.76
data required time                                       0.76
-----------------------------------------------------------------
data required time                                       0.76
data arrival time                                       -0.76
-----------------------------------------------------------------
slack (MET)                                              0.00
```

Figure 17: report of Controller_timing for fastest clk

The area report for this setting is shown below.

```
Number of ports:                           69
Number of nets:                          2163
Number of cells:                         2034
Number of combinational cells:           1417
Number of sequential cells:               617
Number of macros/black boxes:               0
Number of buf/inv:                        228
Number of references:                     225

Combinational area:                4469.760042
Buf/Inv area:                       414.720009
Noncombinational area:             4791.960039
Macro/Black Box area:                 0.000000
Net Interconnect area:            11219.988831

Total cell area:                   9261.720081
Total area:                       20481.708912
```

Figure 18: report of Controller_area for fastest clk

After we find the fastest working frequency of the clock, we change the clock back to a normal frequency. The reports for time and area are shown below.

```
# create clock constraint:
create_clock [get_ports clk] -name "CLOCK" -period 1000000.0
```

Figure 19: setting in constrain.tlc for normal clk

```
clock CLOCK (rise edge)                 1000000.00 1000000.00
clock network delay (ideal)                0.00   1000000.00
clock uncertainty                         -0.03   1000000.00
DC_Comp_reg_0_/CK (DFRM1SA)                0.00   1000000.00 r
library setup time                        -0.13   999999.88
data required time                                999999.88
-----------------------------------------------------------
data required time                                999999.88
data arrival time                                    -7.96
-----------------------------------------------------------
slack (MET)                                       999991.94
```

Figure 20: report of Controller_timing for normal clk

```
Number of ports:                         69
Number of nets:                        2024
Number of cells:                       1863
Number of combinational cells:         1244
Number of sequential cells:             619
Number of macros/black boxes:             0
Number of buf/inv:                      143
Number of references:                    85

Combinational area:              3382.560037
Buf/Inv area:                     182.880006
Noncombinational area:           4799.160054
Macro/Black Box area:               0.000000
Net Interconnect area:          10874.307542

Total cell area:                 8181.720091
Total area:                     19056.027633
1
```

Figure 21: report of Controller_area for normal clk

When we compare the results of two settings, we can find out that, the area of chip increases if the clock work at a higher frequency. So we can not get the smallest chip and fastest clock at the same time, we need to make a balance between them.

4.2 Power consumption at the nominal frequency

The power consumption of the digital part at the nominal frequency 1000 Hz is 3.0422e-04 mW as shown in figure 22

Power Group	Internal Power	Switching Power	Leakage Power	Total Power	(%)	Attrs
io_pad	0.0000	0.0000	0.0000	0.0000	(0.00%)	
memory	0.0000	0.0000	0.0000	0.0000	(0.00%)	
black_box	0.0000	0.0000	0.0000	0.0000	(0.00%)	
clock_network	0.0000	0.0000	0.0000	0.0000	(0.00%)	
register	3.2978e-07	0.0000	1.8511e+04	1.8869e-05	(6.20%)	
sequential	6.3330e-07	0.0000	1.6351e+05	1.6416e-04	(53.96%)	
combinational	0.0000	5.6832e-07	1.2060e+05	1.2120e-04	(39.84%)	
Total	9.9349e-07 mW	6.0609e-07 mW	3.0262e+05 pW	3.0422e-04 mW		

Figure 22: Power consumption at the nominal frequency

4.3 Minimal layout area with Encounter

In the end we use Encounter to place and route our design. The setting is shown below. From the picture we can see that, the ratio for the utilization of the chip area is 0.95

```
# Initialize floorplan based on aspect ratio (Height/Width) and utilization as a first step (GUI: Floorplan -> Specify Floorplan...):
floorPlan -site CORE -r 1 0.95 5.8 5.8 5.8 5.8
#
```

Figure 23: setting for Encounter

Then we get the result, we can see from the picture that, there is no error of connection. The length of the side of the square is 103.145 m. The total area of chip is around 10900 m2

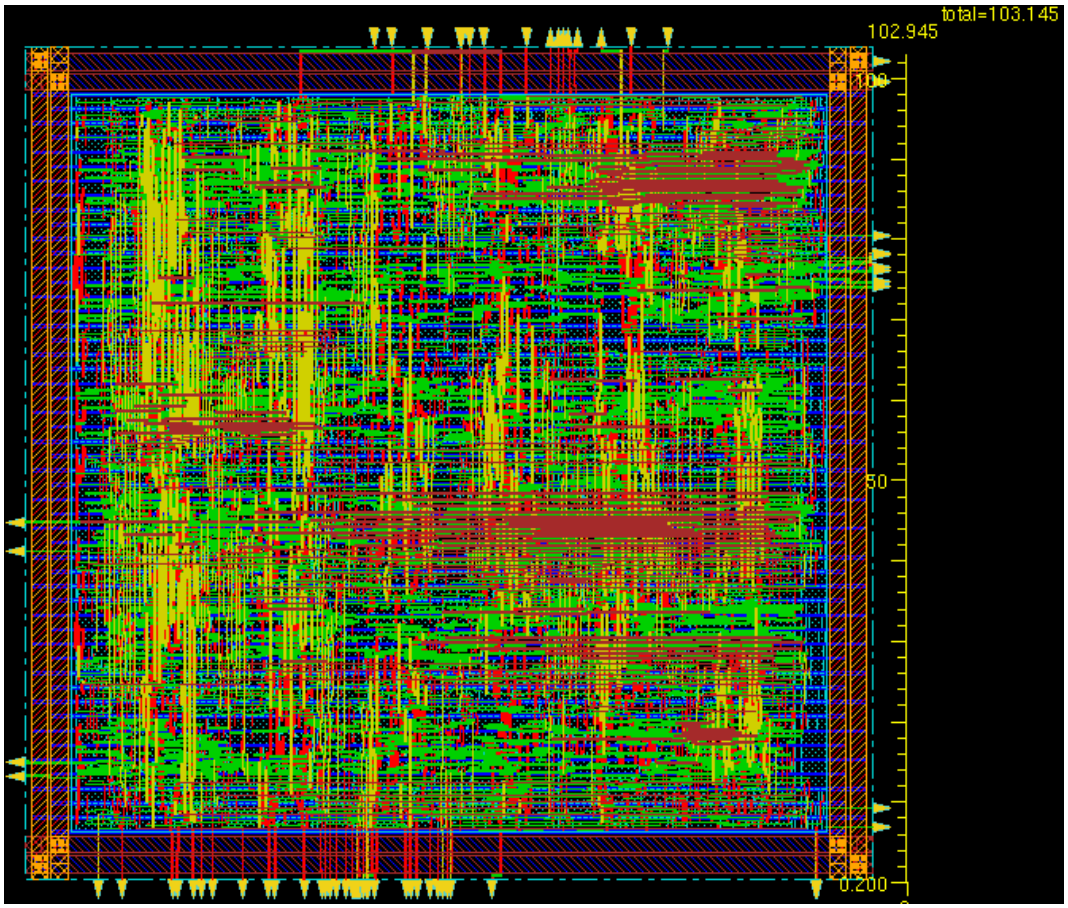

Figure 24: result of Encounter

```
==============================
Floorplan/Placement Information
==============================
Total area of Standard cells: 8610.840 um^2
Total area of Standard cells(Subtracting Physical Cells): 8116.560 um^2
Total area of Macros: 0.000 um^2
Total area of Blockages: 0.000 um^2
Total area of Pad cells: 0.000 um^2
Total area of Core: 8612.676 um^2
Total area of Chip: 10900.428 um^2
Effective Utilization: 1.0000e+00
Number of Cell Rows: 51
% Pure Gate Density #1 (Subtracting BLOCKAGES): 99.979%
% Pure Gate Density #2 (Subtracting BLOCKAGES and Physical Cells): 94.240%
% Pure Gate Density #3 (Subtracting MACROS): 99.979%
% Pure Gate Density #4 (Subtracting MACROS and Physical Cells): 94.240%
% Pure Gate Density #5 (Subtracting MACROS and BLOCKAGES): 99.979%
% Pure Gate Density #6 (Subtracting MACROS and BLOCKAGES and Physical Cells): 94.240%
% Core Density (Counting Std Cells and MACROs): 99.979%
% Core Density #2(Subtracting Physical Cells): 94.240%
% Chip Density (Counting Std Cells and MACROs and IOs): 78.995%
% Chip Density #2(Subtracting Physical Cells): 74.461%
# Macros within 5 sites of IO pad: No
```

Figure 25: report of Encounter

5 Conclusion

During this lab we have learnt a lot of things. It provide us a best chance for us to use our knowledge in practices and solve real problems.

We have already known that, the Verilog is a language for the description of hardware. But when we wrote codes, we often forgot this concept and wrote codes just like C language. The result for that is we spent much time on finding bugs and correct the codes. It is a valuable experience for us, so that we will form a better coding habit in the feature.

Because both of us don't have experience on Verilog projects, it took lots of time for us to get familiar with the software. Thanks to our tutor, when we had any kind of questions, he always answered us in time and gave us friendly instruction with patience.

Finally this lab is also a good way to train our ability of teamwork. Though we all experience the hard time of epidemic, we still met every day and work together for the whole day. We camp up with many new ideas and solved many problem we met.

References

[1] Klaus hofmann, dominic korner, ferdinand keil. cad4soc 2019 script complete, 433.

[2] Klaus hofmann, dominic korner. hdl lab manual 2020, 2.

Appendix

1. Controller

In this Controller, S0 is DC_Comp approxiamtion state, S1 is DC_Comp adjustment state, S2 is PGA_Gain find state and S3 is LED toggle state.

```
module Controller (clk, Find_Setting, rst_n, V_ADC, DC_Comp, LED_Drive, LED_IR, LED_RED, Out_IR, Out_RED,
    PGA_Gain, clk_filter);
  input clk, Find_Setting, rst_n, clk_filter; //clk_filter is another clk for filter, which is faster
    than 500Hz
  input [7:0] V_ADC;
  output reg [6:0] DC_Comp;
  output reg [3:0] LED_Drive, PGA_Gain;
  output reg LED_IR, LED_RED;
  output wire [19:0] Out_IR, Out_RED;

  //boundary for finding DC_Comp
  parameter ADC_HALF_MAX = 8'b10000111, ADC_HALF_MIN = 8'b01110000;

  //boundary for finding PGA_Gain
    parameter ADC_MAX_1 = 8'b11001100, ADC_MAX_2 = 8'b11111110, ADC_MIN_1 = 8'b00011001, ADC_MIN_2 = 8'
    b00000001;
    parameter COUNT_100HZ = 10, COUNT_1HZ = 1000; //how many times we need to count for a certain clk
    frequency

  //used for state: S0: search for a suitable DC_Comp for LEDs; S1: search for DC_Comp exactly; S2:
    search for a suitable PGA_Gain for LEDs;  S3: change LEDs for a certain frequency after we have
    found right settings for both of LEDs
  parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011;
  //used for find_state: IR_DC_FINDING: find DC_Comp for LED IR;  IR_GAIN_FINDING: find PGA_Gain for LED
    IR;
            // RED_DC_FINDING: find DC_Comp for LED RED; RED_GAIN_FINDING: find PGA_Gain for LED RED
  parameter IR_DC_FINDING = 3'b000, IR_GAIN_FINDING = 3'b001, RED_DC_FINDING = 3'b010, RED_GAIN_FINDING =
     3'b011;
  reg [2:0] state, find_state;
  reg [9:0] count;  //We need a long period of counting when we search for PGA_Gain of LED

  reg [3:0] PGA_red_max, PGA_infrared_max;  //PGA_Gain of red and infrared LED which we have already
    found
  reg [6:0] DC_Comp_red, DC_Comp_infrared;  //DC_Comp of red and infrared LED which we have already found
  reg [7:0] ADC_max, ADC_min; //find the max and min value of V_ADC, it helps us to find the right
    setting of PGA_Gain
  wire [7:0] ADC_RED, ADC_IR; //output of ADC signal of RED_LED and IR_LED
  wire [2:0] dc_pos, pga_pos; //position of bit "1" in the setting of DC_Comp and PGA_Gain (using a
    funtion from other file)
  //maxmin_init == 1: begin to find the max and min value of V_ADC; maxmin_init == 0: stop finding;
  //clock_sampling: the frequency of FIR_Filter, which is 500Hz
  reg maxmin_init, clock_sampling;
  reg find_setting_previous;  //save the find_setting signal of the last clk period


  always @(V_ADC or maxmin_init) begin
    if (!maxmin_init) begin //initial values to a middel value, 0.9V
      ADC_max = 8'b10000000;
      ADC_min = 8'b10000000;
    end
    else begin    //save max and min value of one cycle
      if(V_ADC > ADC_max)
        ADC_max = V_ADC;
      if(V_ADC < ADC_min)
        ADC_min = V_ADC;
    end
  end

  always @(posedge clk or negedge rst_n) begin
    //reset: clean all the setting and keep swtiching the LED without DC_Comp and PGA_Gain
    if(!rst_n) begin
```

```
      DC_Comp <= 0;
      PGA_Gain <= 0;
      state <= S3;
      PGA_red_max <= 0;
      DC_Comp_red <= 0;
      PGA_infrared_max <= 0;
      DC_Comp_infrared <= 0;
      LED_Drive <= 10;
      count <= 0;
      LED_IR <= 0;
      LED_RED <= 1;
      maxmin_init <= 0;
    end
    //when there is a Find_Setting signal, begin to search settings for both LEDs
    else if(find_setting_previous == 0 && Find_Setting == 1) begin
      state <= S0;
      find_state <= RED_DC_FINDING;
      DC_Comp <= 7'b1000000;
      PGA_Gain <= 0;
      LED_IR <= 0;
      LED_RED <= 1;
      maxmin_init <= 0; //don't find max and min value, this is only for PGA_Gain
      count <= 0;
      find_setting_previous <= Find_Setting;  //save the state of Find_Setting signal for every clk cycle
    end
    else begin
      find_setting_previous <= Find_Setting;  //save the state of Find_Setting signal for every clk cycle
      case (state)  //begin the statemachine
        S0: begin
          if(V_ADC > ADC_HALF_MAX)  //V_ADC is too large, increase the DC_Comp to reduce V_ADC
            case(dc_pos)
                1: DC_Comp <= DC_Comp | 7'b0000001;
              2: DC_Comp <= DC_Comp | 7'b0000010;
                3: DC_Comp <= DC_Comp | 7'b0000100;
              4: DC_Comp <= DC_Comp | 7'b0001000;
                5: DC_Comp <= DC_Comp | 7'b0010000;
                6: DC_Comp <= DC_Comp | 7'b0100000;
            endcase
             else if(V_ADC < ADC_HALF_MIN)  //V_ADC is too small, decrease the DC_Comp to increase V_ADC
              case(dc_pos)
                0: DC_Comp <= DC_Comp & 7'b1111110;
                1: DC_Comp <= (DC_Comp & 7'b1111101)|7'b0000001;
                  2: DC_Comp <= (DC_Comp & 7'b1111011)|7'b0000010;
                3: DC_Comp <= (DC_Comp & 7'b1110111)|7'b0000100;
                4: DC_Comp <= (DC_Comp & 7'b1101111)|7'b0001000;
                5: DC_Comp <= (DC_Comp & 7'b1011111)|7'b0010000;
              6: DC_Comp <= (DC_Comp & 7'b0111111)|7'b0100000;
            endcase
           else begin
                if(find_state == RED_DC_FINDING) begin  //save setting for RED LED
              DC_Comp_red <= DC_Comp;
              find_state <= RED_GAIN_FINDING;
            end
            else if(find_state == IR_DC_FINDING) begin  ///save setting for IR LED
              DC_Comp_infrared <= DC_Comp;
              find_state <= IR_GAIN_FINDING;
            end
            state <= S1;
                maxmin_init <= 0;
          end
          end
        S1: begin
             maxmin_init <= 1; //start to find max and min value
             if(count < 500) //collect data for 0.5s
                 count <= count + 1;
             else begin
               count <= 0;

                       if( (ADC_max + ADC_min)/2 > ADC_HALF_MAX)
                 DC_Comp <= DC_Comp + 1;
```

```
                    else if( (ADC_max + ADC_min)/2 < ADC_HALF_MIN)
122               DC_Comp <= DC_Comp - 1;
                  else begin
124               state <= S2;
                  PGA_Gain <= 4'b0100;
126               maxmin_init <= 0;
                end
128
                    if(find_state == RED_GAIN_FINDING)
130                     DC_Comp_red <= DC_Comp;
                    else if(find_state == IR_GAIN_FINDING)
132                     DC_Comp_infrared <= DC_Comp;

134             maxmin_init <= 0;
              end
136         end
          S2: begin
138         maxmin_init <= 1; //start to find max and min value
            if(count < COUNT_1HZ) //collect data for 1s
140             count <= count + 1;
            else begin
142           count <= 0;
              if((ADC_max < ADC_MAX_1) && (ADC_min > ADC_MIN_1))  //V_ADC is too small, increase PGA_Gain
    to increase V_ADC
144             case(pga_pos)
              //  0: PGA_Gain <= PGA_Gain + 1;
146               1: PGA_Gain <= PGA_Gain | 4'b0001;
                  2: PGA_Gain <= PGA_Gain | 4'b0010;
148               3: PGA_Gain <= PGA_Gain | 4'b0100;
                endcase
150           else if((ADC_max > ADC_MAX_2) || (ADC_min < ADC_MIN_2)) //V_ADC is too large, decrease
    PGA_Gain to decrease V_ADC
                case(pga_pos)
152               0: PGA_Gain <= PGA_Gain & 4'b1110;
                  1: PGA_Gain <= (PGA_Gain & 4'b1101) | 4'b0001;
154               2: PGA_Gain <= (PGA_Gain & 4'b1011) | 4'b0010;
                  3: PGA_Gain <= (PGA_Gain & 4'b0111) | 4'b0100;
156             endcase
              else begin
158             if(find_state == RED_GAIN_FINDING) begin  //save setting for RED LED
                      PGA_red_max <= PGA_Gain;
160               DC_Comp <= 7'b1000000;
                  PGA_Gain <= 4'b0000;
162               LED_RED <= 0;
                  LED_IR <= 1;
164               find_state <= IR_DC_FINDING;
                  state <= S0;
166             end
                else if(find_state == IR_GAIN_FINDING) begin  //save setting for IR LED
168               PGA_infrared_max <= PGA_Gain;
                  DC_Comp_infrared <= DC_Comp;
170               state <= S3;
                end
172           end
              maxmin_init <= 0;
174         end
          end
176       S3: begin
              if(count < COUNT_100HZ) //frequency of switching LEDs
178             count <= count + 1;
            else begin
180           count <= 0;
              LED_IR <= !LED_IR;
182           LED_RED <= !LED_RED;
              if(LED_IR == 1) begin //use the corresponding setting for LED_IR
184               PGA_Gain <= PGA_red_max;
                  DC_Comp <= DC_Comp_red;
186           end
              else if(LED_RED == 1) begin //use the corresponding setting for LED_RED
188               PGA_Gain <= PGA_infrared_max;
```

```
                DC_Comp <= DC_Comp_infrared;
            end
          end
          end
      endcase
    end
  end

  always @(posedge clk)
    if(!rst_n)
        clock_sampling <= 0;
    else
        clock_sampling <= !clock_sampling;  //The frequency of filter of the half of the frequency of
    Controller

  Demux_1to2 demux_bitstream(.in(V_ADC), .s(LED_RED), .out0(ADC_IR), .out1(ADC_RED), .rst(rst_n));
  Position_check pos_check(.dc_pos(dc_pos), .DC_Comp(DC_Comp), .PGA_Gain(PGA_Gain), .pga_pos(pga_pos));
  FIR_filter red_fir_filter(.clk(clk_filter), .rst(rst_n), .filter_in(ADC_RED), .filter_out(Out_RED), .
    clk_sampling(clock_sampling));
  FIR_filter ir_fir_filter(.clk(clk_filter), .rst(rst_n), .filter_in(ADC_IR), .filter_out(Out_IR), .
    clk_sampling(clock_sampling));
endmodule
```

Listing 3: Controller

2. Position_check

```
module Position_check(DC_Comp, PGA_Gain, dc_pos, pga_pos);

  input [6:0] DC_Comp;
  input [3:0] PGA_Gain;
  output reg [2:0] dc_pos;
  output reg [2:0] pga_pos;

  always @(DC_Comp)
      if(DC_Comp & 7'b0000001)
        dc_pos = 0;
      else if(DC_Comp & 7'b0000010)
        dc_pos = 1;
      else if(DC_Comp & 7'b0000100)
        dc_pos = 2;
      else if(DC_Comp & 7'b0001000)
        dc_pos = 3;
      else if(DC_Comp & 7'b0010000)
        dc_pos = 4;
      else if(DC_Comp & 7'b0100000)
        dc_pos = 5;
      else if(DC_Comp & 7'b1000000)
        dc_pos = 6;


  always @(PGA_Gain)
      if(PGA_Gain & 4'b0001)
        pga_pos = 0;
      else if(PGA_Gain & 4'b0010)
        pga_pos = 1;
      else if(PGA_Gain & 4'b0100)
        pga_pos = 2;
        else if(PGA_Gain & 4'b1000)
        pga_pos = 3;

endmodule
```

Listing 4: Position_check

3. Demux_1to2

```
module Position_check(DC_Comp, PGA_Gain, dc_pos, pga_pos);
module Demux_1to2(out0, out1, in, s, rst);
  input s, rst;
```

```
  input [7:0] in;
  output reg [7:0] out0, out1;

  always @(rst or in or s) begin
      if(!rst) begin
        out0 <= 0;
        out1 <= 0;
      end
      else begin
        if(s==0) out0 <= in;
        if(s==1) out1 <= in;
      end
  end
endmodule
```

Listing 5: Demux_1to2

4. FIR_filter

```
module FIR_filter(clk_sampling, clk, rst, filter_in, filter_out);

  input clk_sampling, clk, rst;
    input wire signed [7:0] filter_in;
    output reg signed [19:0] filter_out;

  parameter word_width = 8;
    parameter order = 21;
    parameter out_width = 20;
    parameter order_half = 10;

    reg [word_width-1:0] L_coef;                    //register to save the current used coefficient
    reg [word_width-1:0] L_tap;                     //register to save the current used tap
    reg [out_width- 1:0] L_product;                 //register to save the current product
    reg [5:0] tap_counter;                          //counter to indentify the current used index for
    tap and coefficient
    reg clk_sampling_previous, clk_sampling_current;

    reg signed [word_width-1:0] delay_tap[order:0];

    //define coef
    wire signed [word_width-1:0] coef[order:0];
    assign coef[0] = 2;
    assign coef[1] = 10;
    assign coef[2] = 16;
    assign coef[3] = 28;
    assign coef[4] = 43;
    assign coef[5] = 60;
    assign coef[6] = 78;
    assign coef[7] = 95;
    assign coef[8] = 111;
    assign coef[9] = 122;
    assign coef[10] = 128;
    assign coef[11] = 128;
    assign coef[12] = 122;
    assign coef[13] = 111;
    assign coef[14] = 95;
    assign coef[15] = 78;
    assign coef[16] = 60;
    assign coef[17] = 43;
    assign coef[18] = 28;
    assign coef[19] = 16;
    assign coef[20] = 10;
    assign coef[21] = 2;

    //define multiplier
    reg signed [out_width-1:0] product[order_half:0];

    //define sum_buf
    reg signed [out_width-1:0] sum_buf;

```

```
    //defined input data buffer
52  reg signed [word_width-1:0] data_in_buf;

54  //get new data and save in data_in_buf
    always @(posedge clk_sampling or negedge rst) begin
56  if (!rst) begin
      data_in_buf <= 0;
58  end
    else begin
60    data_in_buf <= filter_in;
    end
62  end

64  //update delay_tap
    always @(posedge clk_sampling or negedge rst) begin
66    if(!rst) begin
          delay_tap[0] <= 0;
68        delay_tap[1] <= 0;
          delay_tap[2] <= 0;
70        delay_tap[3] <= 0;
          delay_tap[4] <= 0;
72        delay_tap[5] <= 0;
          delay_tap[6] <= 0;
74        delay_tap[7] <= 0;
          delay_tap[8] <= 0;
76        delay_tap[9] <= 0;
          delay_tap[10] <= 0;
78        delay_tap[11] <= 0;
          delay_tap[12] <= 0;
80        delay_tap[13] <= 0;
          delay_tap[14] <= 0;
82        delay_tap[15] <= 0;
          delay_tap[16] <= 0;
84        delay_tap[17] <= 0;
        delay_tap[18] <= 0;
86        delay_tap[19] <= 0;
          delay_tap[20] <= 0;
88        delay_tap[21] <= 0;
      end
90    else begin
          delay_tap[0] <= data_in_buf;
92        delay_tap[1] <= delay_tap[0];
          delay_tap[2] <= delay_tap[1];
94      delay_tap[3] <= delay_tap[2];
          delay_tap[4] <= delay_tap[3];
96        delay_tap[5] <= delay_tap[4];
          delay_tap[6] <= delay_tap[5];
98        delay_tap[7] <= delay_tap[6];
        delay_tap[8] <= delay_tap[7];
100       delay_tap[9] <= delay_tap[8];
          delay_tap[10] <= delay_tap[9];
102       delay_tap[11] <= delay_tap[10];
          delay_tap[12] <= delay_tap[11];
104       delay_tap[13] <= delay_tap[12];
          delay_tap[14] <= delay_tap[13];
106       delay_tap[15] <= delay_tap[14];
          delay_tap[16] <= delay_tap[15];
108     delay_tap[17] <= delay_tap[16];
          delay_tap[18] <= delay_tap[17];
110       delay_tap[19] <= delay_tap[18];
          delay_tap[20] <= delay_tap[19];
112       delay_tap[21] <= delay_tap[20];
      end
114 end

116   /*
       *  This always block will run every time new data arrive with much faster clock frequency.
118    *  By using registers to save the current used coefficient, tap and calculating product
       *  multiplier and adder are reused to reduce used hardward and make the critical path shorter.
120    *  When the posedge of clock sigle is detected, next coefficient and delay tap are loaded to
```

```
     *  resgister L_coef and L_tap. The calculated result is then saved in register L_product.
     */
    always @(posedge clk or negedge rst) begin
      if (!rst) begin
          tap_counter <= 0;
          clk_sampling_previous <= 0;
          clk_sampling_current <= 0;
          filter_out <= 0;
          sum_buf <= 0;
          L_product <= 0;
          L_tap <= 0;
          L_coef <= 0;
      end
      else begin
          clk_sampling_previous <= clk_sampling_current;
          clk_sampling_current <= clk_sampling;
          if(clk_sampling_previous == 0 && clk_sampling_current == 1) begin
        tap_counter <= 0;
            sum_buf <= 0;
            L_product <= 0;
            L_tap <= 0;
            L_coef <= 0;
          end
          if(tap_counter == order + 1)
            filter_out <= sum_buf;
          else begin
            if(tap_counter > 21) begin
              sum_buf <= sum_buf + L_product;
              tap_counter <= tap_counter + 1;
            end
            else begin
              L_tap <= delay_tap[tap_counter];
              L_coef <= coef[tap_counter];
              L_product <= L_coef * L_tap;
              sum_buf <= sum_buf + L_product;
              tap_counter <= tap_counter + 1;
            end
      end
    end
  end
endmodule
```

Listing 6: FIR_filter

5. Controller_TB

```
`timescale 1us/1ps
module Controller_TB();
 reg clk;
 reg Find_Setting, rst_n, clk_filter;
 wire [6:0] DC_Comp;
 wire [3:0] LED_Drive, PGA_Gain;
 wire LED_IR, LED_RED;
 wire [19:0] Out_IR, Out_RED;
 wire [7:0] Vppg;
Controller DUT(.clk(clk),
    .Find_Setting(Find_Setting),
    .rst_n(rst_n),
    .V_ADC(Vppg),
    .DC_Comp(DC_Comp),
    .LED_Drive(LED_Drive),
    .PGA_Gain(PGA_Gain),
    .LED_IR(LED_IR),
    .LED_RED(LED_RED),
    .Out_IR(Out_IR),
    .Out_RED(Out_RED),
    .clk_filter(clk_filter));

Fingerclip_Model Fingerclip(.Vppg(Vppg), .DC_Comp(DC_Comp), .PGA_Gain(PGA_Gain));

```

```
always #20 clk_filter = !clk_filter; //clock_filter 20000Hz, faster 25 times as clk

always #500 clk = !clk; //Clock 1000Hz

initial
  begin
    clk = 1'b0;
    clk_filter = 1'b0;
    Find_Setting = 0;
    rst_n = 1;

    #2000 rst_n = 0;
    #2000 rst_n = 1;
    #2000 Find_Setting = 1;
    #1000000000 $stop;
  end
endmodule
```

Listing 7: Controller_TB

6. FIR_filter_TB

```
`timescale 1ms/1us
module FIR_filter_TB();
  reg clk;
  reg clk_sampling;
  reg rst;
  reg signed [7:0] filter_in;

  wire signed [19:0] filter_out;

  FIR_filter DUT(.clk(clk),
                 .rst(rst),
                 .filter_in(filter_in),
                 .filter_out(filter_out),
                 .clk_sampling(clk_sampling));

  initial begin
    clk = 0;
    clk_sampling = 0;
    rst = 0;
    #24;
    rst = 1;
    #10000 $stop;
  end

  always #1 clk = !clk;       // This is fast clock used for calculating
  always #25 clk_sampling = !clk_sampling; // This is slow clock use for sampling



  initial begin //#120.2
    #23 filter_in =5;
    repeat(2) begin
        #50 filter_in = 0;
        #50 filter_in = 25;
        #50 filter_in = 50;
        #50 filter_in = 25;
        #50 filter_in = 20;
        #50 filter_in = 10;
        #50 filter_in = 0;
    end
  end

  integer file;
  initial begin
    file = $fopen("dataout.txt", "w");
  end

  always @(posedge clk_sampling) begin
```

```
    $fdisplay(file, filter_out);
  end
endmodule
```

Listing 8: FIR_filter_TB

7.Fingerclip_Model_TB

```
`timescale 1ms/1us
module Fingerclip_Model_TB ();



  reg [0:6] DC_Comp;
  reg [0:3] PGA_Gain;
  wire [0:7] Vppg;
  reg clk;

  Fingerclip_Model dut(.Vppg(Vppg), .DC_Comp(DC_Comp), .PGA_Gain(PGA_Gain));

  initial begin
    DC_Comp = 0;
    PGA_Gain = 0;
    clk = 0;
    #1000000 $stop;
  end

  always #500 clk = !clk;

  always@(posedge clk) begin
    if (DC_Comp < 40) DC_Comp = DC_Comp +1;
    else PGA_Gain = PGA_Gain + 1;

  end
endmodule
```

Listing 9: Fingerclip_Model_TB