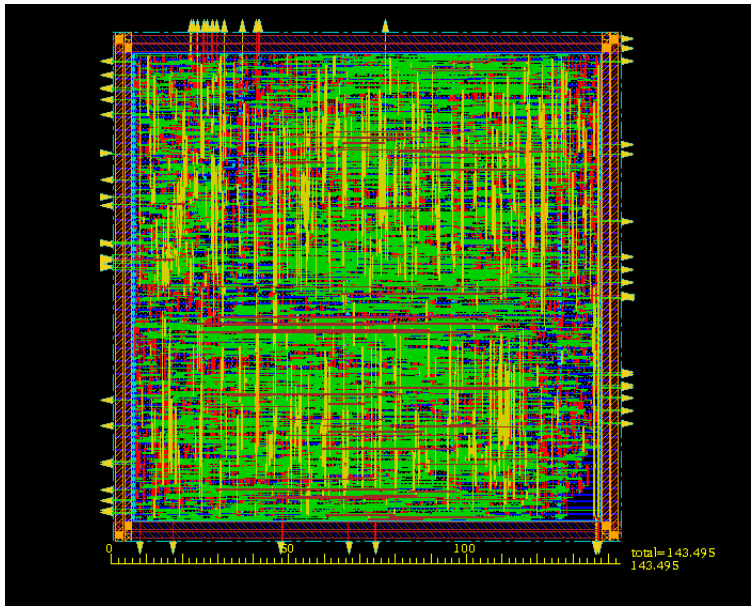
Implementation of an ADC, Controller and FIR Filter with Verilog for Pulse Oximetry in UMC65nm Technology



HDL Lab
Ho Thanh Tu Nguyen, 1804386
Nguyen Tien Dat Tran, 2871790

Supervisor: Dominik Kohrer
Integrated Electronic System Lab
Prof.Dr.-Ing. Klaus Hofmann
Sommer Semester 2020

Inhaltsverzeichnis

1. Introduction

The task of the HDL Lab project is to design the digital part of an ADC, controller and FIR Filter for pulse oximetry. In order to to this, we divided our report in five sections. The structure of our report is as follow:

In Sec.2, we present our design approach for each component. You may find some pseudocode in this section since we want to keep this section short and clean for quick understanding. The complete code be provided at the end in Appendix. In Sec.3, we present the result that we have submitted. In Sec.4, we present the result after the submission time, since we have some improvements after the submission time. Then we'll come to the conclusion in section 5. For all the code, we put them at the end in the Appendix section.

2. Design Approach

2.1. The System Overview

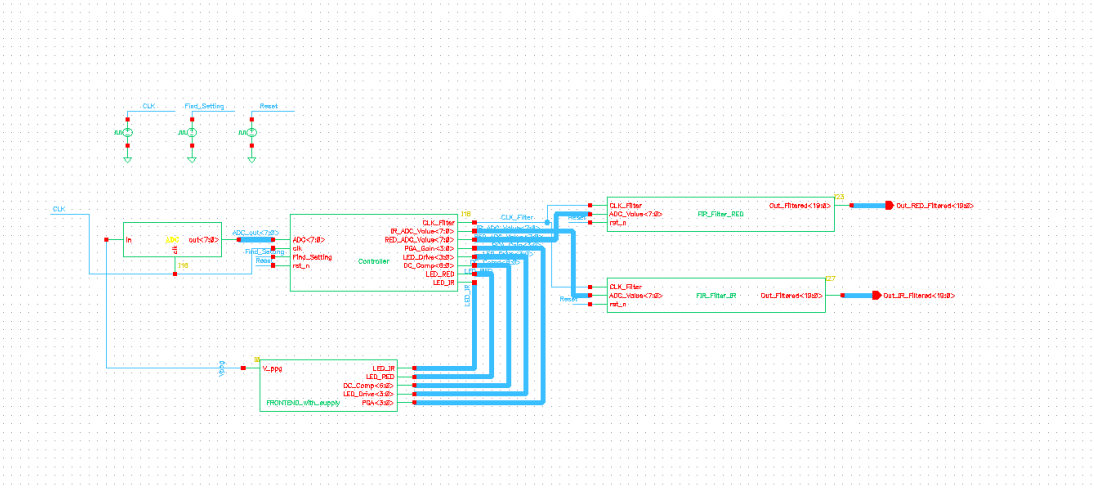


Abbildung 1: Schematic of the System

As we can see the figure 1, our system consists of 4 main parts: Finger Clip Model, ADC, Controller and the FIR Filter. We will cover each part individually in the following sections below.

2.2. Finger Clip Model

As we already noticed that the Finger Clip Model in ModelSim is a bit slow. It only reacts to the change of DC Compensation and PGA GAIN after 0.545s. Therefore, we extend the always block code of Finger Clip Model in ModelSim as follow:

```
`timescale 1ms/1us
38
module Fingerclip_Model(Vppg, DC_Comp, PGA_Gain);
40   ...
   always@(posedge clk or negedge clk or DC_Comp or PGA_Gain ) begin
42   ...
endmodule
```

Listing 1: Excerpt of Finger Clip Model

2.3. ADC Design

The analog to digital converter is a very important device since it takes the captured and amplified analog signal from the fingerclip model and convert it into a digital signal. In this task an 8-bit ADC, running at a sampling frequency of 1kHz, is needed to be implemented in Verilog-A. In the following we are going to explain our implementation in detail.

2.3.1. Verilog-A Code

All of our written Verilog and Verilog-A code, respectively, lie in the Appendix. So for the ADC, we only plot the most significant parts in Listing 2.

For this task, the analog-to-digital converter awaits an analog signal and converts it into an 8-bit bus string. To do this, we need to define two input variables *in* and *clk* as well as an 8-bit output variable *out*. Furthermore, we declare a threshold variable *vth* which has a value of 0.9. The choice of this value is do to the fact that this is exactly one MSB and facilitate our implementation of ADC, which will be explained in detail afterwards. After that, we can just skip into our excerpt of code in Listing 2, in which the main functionality of the ADC begins.

The cross-function in line 21 signalize that the operation commences after each time when the clock signal cross the threshold voltage, i.e. after each rising and falling edge. This event will trigger the comparison of the current input sample and the threshold. To convert the sample voltage into an 8-bit output signal, we can imagine the scenario if the sample voltage has already the right corresponding binary representation. E.g. for an input voltage of 1.35V, the corresponding binary code would be 1100 0000. The next step is to compare it with the binary representation of the threshold voltage. As it is the MSB, the representation is 1000 0000. Note that we just assume that we know the binary representation of our input voltage, but actually want to calculate it. So to do this, we start to compare the voltages. If the voltage is greater than 0.9V, the MSB of the input voltage is '1' and the sample voltage is reduced by the threshold voltage. If this is not the case, the bit representation is '0'. The next step is the most important one. So in both cases, either if the MSB is '0' or '1', the new resulting sample voltage must be doubled (line 30). We can imagine it as the 'predicted' binary representation is shifted to the left by one bit and the next bit element is compared with the MSB. This guarantees our correct calculation for the bit representation of the input sample.

All the bit elements are then stored in a temporary 8-bit register, namely *result*, and applied to the output register *out* afterwards (line 36). So the analog to digital conversion is done!

```
20 analog begin
     @(cross(V(clk) - vth, +1) ) begin
22     sample = V(in);
       for (i = bit - 1; i >= 0; i = i -1) begin
24       if( sample>vth ) begin
           result[i] = 1.8;
26         sample = sample - vth;
         end else begin
28         result[i] = 0.0;
         end
30     sample = 2.0 * sample;
       end
32   end


     for (i = 0; i < bit; i = i + 1 ) begin
36     V(out[i]) <+ transition(result[i], dly, ttime);
     end
38 end
```

Listing 2: Excerpt of Verilog-A ADC Code

2.3.2. ADC Testbench and Simulation

After writing successfully the Verilog-A code, we can now implement our testbench to test our code. To do this, we add two voltage sources: one to generate the clock and the other one to generate a sine wave. The parameters for the clock generator are chosen as described in the pdf, that means that we have a $t_{rise} = t_{fall} = 100$ ns and a pulse width of 500 μ. For the sine wave we should choose parameters which give us a better observation on our results. So we decide to choose a sine wave which is slower than the sampling frequency of the ADC, namely $f_{sine} = 20\ Hz$. The testbench is depicted in Fig.2 and the corresponding simulation result in Fig.3. According to Fig.3 we convert the output signal into a binary sequence and when comparing it with the input sine wave, we can see clearly that the Verilog-A implementation of our ADC works well.

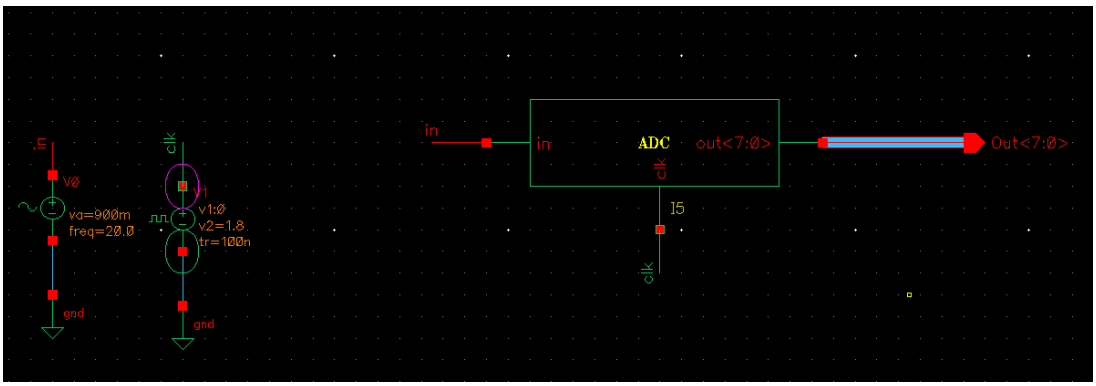


Abbildung 2: ADC testbench.

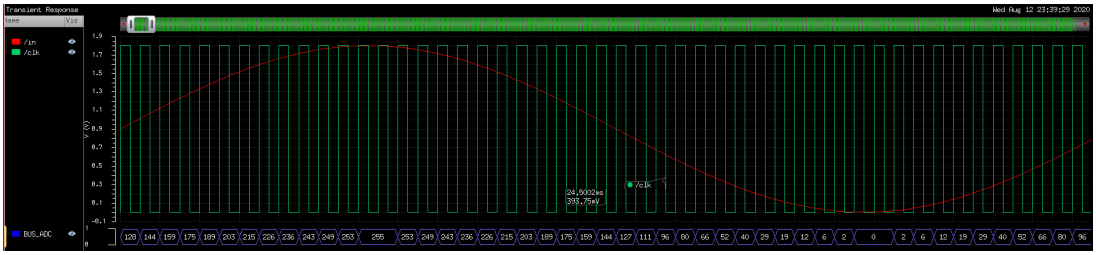


Abbildung 3: ADC testbench simulation.

2.4. Controller Design

2.4.1. State Machine of Controller

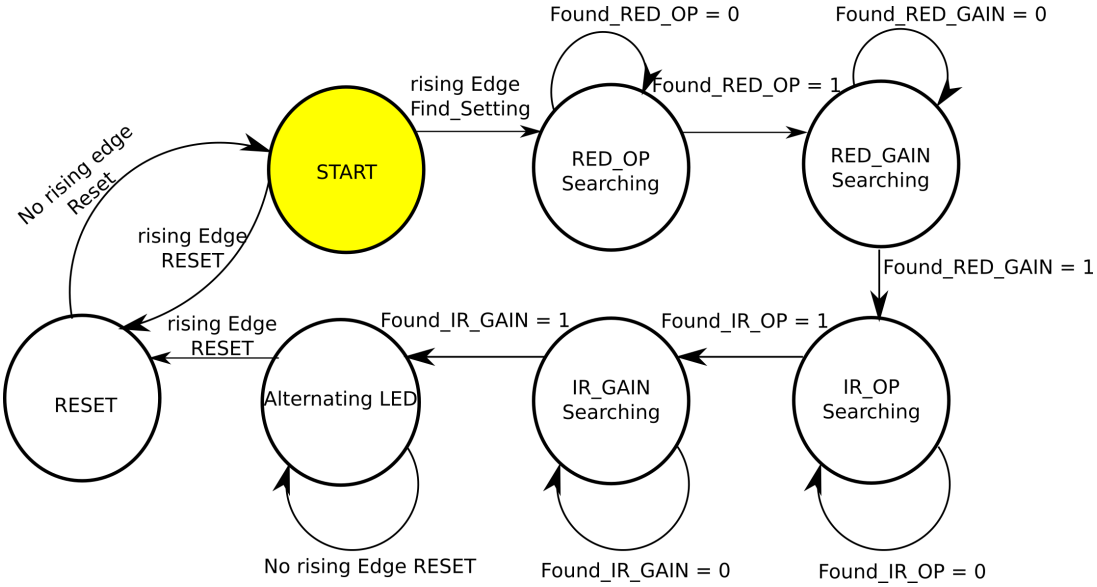


Abbildung 4: State Machine of the Controller

Our implementation of controller is structured as a State Machine. As we can see in the figure 4, we have $START$ state, $RESET$ state at the beginning of the code. Whenever there is a reset Signal(a rising edge reset signal), then we will go the $RESET$ state and set all the parameters to default value. If there is no reset signal, we jump to the $START$ state waiting for the $Find\_Setting$ signal.

A $Find\_Setting$ signal is detected, we start a series of states which are our main task:

- $RED\_OP\_Searching$: Finding the DC Compensation of RED LED
- $RED\_OP\_Searching$: Finding the PGA Gain of RED LED
- $IR\_OP\_Searching$: Finding the DC Compensation of IR LED
- $RED\_OP\_Searching$: Finding the PGA Gain of IR LED
- $Alternating\_LED$: Continuously switch the LEDs, split ADC stream to two ADCs stream and feed them to the FIR filter.

2.4.2. DC Compensation

The analog output of Finger Clip Model(Vppg) is fed directly to an ADC. Moreover, an ADC can only convert analog to digital signal, if the analog signal lies in the range of 0V to 1.8V. Therefore, if the Vppg doesn't not lie in range of 0V to 1.8V, the ADC would not be able to read this signal. Our goal is to adapt the DC compensation parameter(the input of Finger Clip Model) such that the middle

point of Vppg lies in the range of the threshold voltage of 0.9 as close as possible.

The approach to find the right DC compensation is very simple. First we set the PGA gain to 0 dB, so that it does not amplify at all. We notice that if we increase the DC Compensation, the signal will go down. In other word, the DC Compensation is anti-proportional to the Vppg. Therefore, if we detect that the DC middle of Vppg is larger than 0.9, we increase the DC Compensation. And vice versa, if the DC middle of Vppg is smaller than 0.9, we decrease the DC Compensation.

In order to find the DC middle, we have to obtain the maximum and minimum value of Vppg in a certain amount of time. The pseudocode is structured as follow:(**the verilog code can be completely found at the Apendix, since we want to keep this section short and clean**)

```
int interval = 10; //interval for searching min max is 10ms
int max, min, diff, dc_middle, acceptable_error=5;
find_Max_Min(interval);
diff = max-min;
dc_middle = min + diff/2; //find the middle point of Vppg signal
offset = dc_middle - 127;
//adjust the dc_compensation to get the middle point matching 0.9V
if(dc_middle > 127)
    dc_compensation = dc_compensation + 1;
else
    dc_compensation = dc_compensation + 1;

if(offset < acceptable_error)
    store(dc_compensation);
    stop_search();
    jump_next_state(); // stop searching DC Compensation and start searching for PGA Gain
```

Listing 3: Pseudo Code of finding DC compensation point

One thing to notice in the figure that the interval is adjustable variable. In our case, the Finger Clip Model responses very fast to the change of DC Compensation. Therefore, we set interval to 10ms. That means, after increasing the DC Compensation by 1, we wait 10ms for the response of Finger Clip Model. During this 10ms, we continuously sample and compare the ADC value and obtain the min, max. (In case of PGA Gain, we wait a lot longer, namely 1 second).

2.4.3. PGA Gain

After we find the PGA Compensation and store this value in the register, we now can find the PGA GAIN. The pseudocode is structured at follow:

```
int interval = 1000; //interval for searching min max is 1000ms
int max, min;
find_Max_Min(interval);
//adjust the PGA_Gain to get the middle point matching 0.9V
if(max < 240 and min > 15) //Vppg is still in range (0.1V,1.7V), we keep increasing PGA_Gain
    PGA_Gain = PGA_Gain + 1;
if(max > 240 or min < 15) //Vppg is out of range(reach lowest bound or highest bound)
    store(PGA_Gain);
    stop_search(); // stop searching PGA_Gain
Remark: In the verilog code in Cadence, the \textbf{lower bound} and \textbf{upper bound} in reality are slightly
     different as hand calculated case 240 and 15, namely they are set to 200 and 5 for RED LED and 240 and 40 for IR LED.
```

In opposite to the DC Compensation, in which we only wait 20ms, here we wait 1 second. The reason is that, the Finger Clip Model provides feedback of PGA Gain very slowly. Simularly to the DC-Compensation, that we find the max min value. we increase the PGA Gain by 1, then we want 1 second for the response behavior of Finger Clip Model. After 1 second, if the maximum of Vppg is not beyond 1.7 and minimum of Vppg is not below 0.1, then we continue increasing the PGA Gain by 1, until Vppg reachs 0.1 or 1.7.

For both RED led and IR led, we follow the same algorithm above. Once the DC Compensation and the PGA Gain are found, we can switch to other state which is the Alternating State as presented in the next section.

2.4.4. Clock Divider

Because the later tasks don't need 1kHz clock. but different clock frequencies. In particular, the **Alternating LED** task requires 100Hz clock and the **FIR Filter** task requires 500Hz. Therefore, we have to generate these two clock frequencies for later usage. We simply use a **counter** to divide the 1KHz down to 500Hz and 100Hz. For example, we generate 500Hz Clock Filter as follow:

```
always@(posedge clk) begin
    if(rst_n ==1)
      begin
```

```
      Counter_CLK_Filter=0;
      CLK_Filter = 0;
    end
  if(Counter_CLK_Filter==1)
    begin
    Counter_CLK_Filter = 0;
    if (CLK_Filter == 1)
      CLK_Filter=0;
    else
      CLK_Filter =1;
    end
  Counter_CLK_Filter = Counter_CLK_Filter+1;
end
```

The result is as follow:

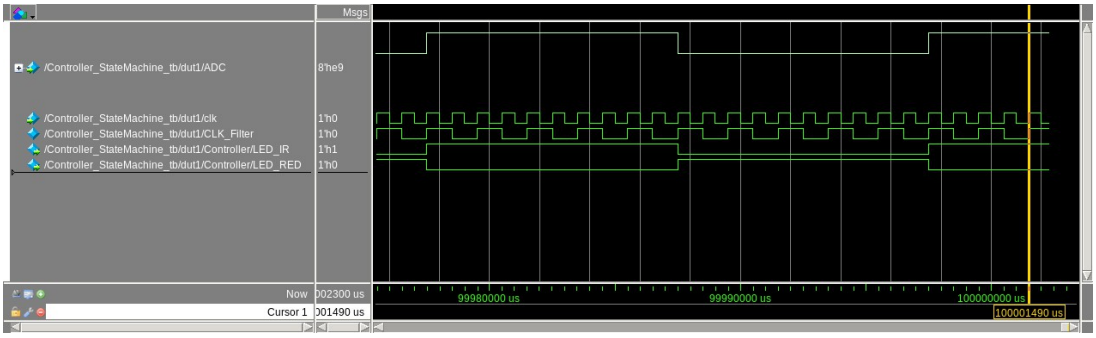


Abbildung 5: ModelSim CLK Divider for CLK Filter and Alternating LED

2.4.5. Alternating LED and split ADC

For the Alternating Task, we turn ON and OFF the IR LED and RED LED one after another with the frequency of 100Hz. In other word, each LED will stay on 10ms and off 10ms periodically as shown in the figure 5. During the 10ms ON, we get the input ADC value and put on the output of the corresponding LED. Here is the pseudocode:

```
input = ADC
output = RED_ADC_Value, IR_ADC_Value
duration = 10 //10ms
if(timer >= 10) //switch LED every 10ms
    switch_state();
    reset_timer();

LED_RED_Stream_State() // put input ADC on output RED_ADC_Value
    TURN_ON(RED_LED)
    TURN_OFF(IR_LED)
    RED_ADC_Value = ADC

IR_RED_Stream_State() // put input ADC on output IR_ADC_Value
    TURN_ON(IR_LED)
    TURN_OFF(RED_LED)
    IR_ADC_Value = ADC

switch_state():
    if(previous_state == LED_RED_Stream_State)
        next_State = IR_RED_Stream_State
    else
        next_State = LED_RED_Stream_State
```

2.5. FIR Filter

In order to design an FIR filter we should understand its function first. Here we make use of the description in the pdf as well as some literature.

2.5.1. Basics

Generally, an FIR filter is a filter which can create an impulse respone with finite length. Its characteristics are based on the choice of the order *N* and the filter coefficients. It is often realised as digital filters.

The mathematical function of an FIR filter is explained by using convolution of the input signal with its filter coefficient as it can be seen in the following equation, in which *x[n-k]* is the input sample, while *h[k]* determines the current filter coefficient and y[n] the resulting output.

$$y[n] = \sum_{k=1}^{N-1} h[k]x[n-k]$$

To realize this implementation, we can refer to Fig.6. It shows that the input sample is multiplied with the corresponding coefficient, depending on the timing 'position' of the input sample. According to the convolution operation input sample needs to be shifted after each clock trigger and therefore the implementation needs to integrate a delay element. After that, all the products are sum together which yield to the current output value.

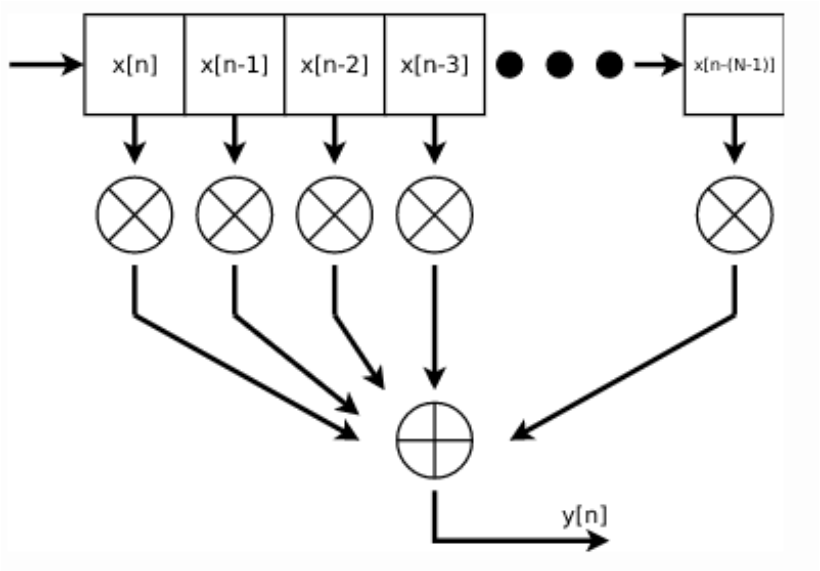


Abbildung 6: FIR Filter Implementation

2.5.2. Verilog Implementation

The complete Verilog code is applied in the Appendix. In general, we implement the behaviour of the FIR filter as described previously. Regarding to efficiency, this implementation is not strongly recommended as it uses 22 multipliers which captures a lot of area. But for the first approach, this is fine. We now explain detailly our implementation.

The Verilog code of our filter is called 'FIR_Filter_Optimized.v'. In this file, we await the signal *CLK_Filter*, which comes from the Controller module. It also gets the value from the ADC, which has a width of 8-bit. At last, we declare a 19-bit output signal, namely 'Out_Filtered'.

From here on, we prepare registers for the multiplication operations and delay elements. An excerpt of our code can be seen in Listing 4. The value before the variable name means the bit width of this variable, whereas the value after the variable declares the amount of values needed to be saved. Hence, we have the register variable *previous_Value* which is responsible for the shift of the elements after each clock cycle, and the register variable *product* which save the 22 products of the convolution operation. We also

have a variable *coeffs* which guards the coefficients of the FIR filter. In this term we can reduce the length of the array as the FIR filter is symmetric.

```
wire signed[8:0] coeffs[21:0];

reg [15:0] previous_Value[21:0];

reg [19:0] product[21:0];
```

Listing 4: Excerpt of Fir_Filtered_Optimized

The delay elements are implemented in that way that after each clock cycle, the next element in the *previous_value* register overtake the previous value.

```
...
previous_Value[1]     <= previous_Value[0];
previous_Value[0]     <= ADC_Value;
...
```

Listing 5: Verilog code to represent the shift and delay operation

As explained before, we use 22 multipliers in this implementation. Because of the symmetric characteristic of the FIR filter, we only need 11 coefficients to be saved.

```
product[10] <= coeffs[10] * previous_Value[10];
    product[11] <= coeffs[10] * previous_Value[11];
```

Listing 6: Verilog code to represent the shift and delay operation

Finally, all the products are added together which yield to the current output value.

```
Out_Filtered <= (product[0] + product[1] +
                 product[2] + product[3] + product[4] + product[5] +
                 product[6] + product[7] + product[8] + product[9] +
                 product[10] + product[11] + product[12] + product[13] +
                 product[14] + product[15] + product[16] + product[17] +
                 product[18] + product[19] + product[20] + product[21]);
```

Listing 7: Verilog code to represent the shift and delay operation

2.5.3. Optimization Ideas

In Sec.2.5.2 we deliver a functional Verilog implementation for our FIR filter. But as mentioned before, this implementation might not be the most efficient one since multipliers take a lot of area. To optimize our design, we use a completely new implementation idea. The full code lays in Appendix B and is named as 'FIR_Filter_Optimized.v'. There is also a testbench included, namely 'FIR_Filter_TB.v'. In the following, we describe our new idea.

Before we consider our new implementation, here is a short mention which parameters / registers we still use. The register *previous_value* needs to be maintained since it remembers past values and after a short period, some of the old values need to be replaced with new values. The coefficient register is increased again to 22 elements.

Now for the new implementation, the first step is to reduce the amount of multipliers to one. Therefore there exists two approaches: The first one is to use just one multiplication operation after each clock cycle of 'CLK_FILTER', but this would be very slow. The second approach is to use a second clock which triggers the multiplication operation. In this case we define a second clock *fast_clk* (line 5). This clock is set up to at least 22 times faster than the sampling rate of the FIR filter. We decide to set the clock speed to 12.5kHz.

The second step is that instead of taking all the product values simultaneously and calculate the sum, we calculate the product of the input sample and its corresponding value together and save it into an accumulator (*accu*). The principle is like a FIFO buffer in which we make use of pointers. Therefore, we use three pointers which significates the state of the FIFO. These pointers are named as *temp_Pointer*, *coeff_Pointer* and *nxt_Pointer*. The implementation is depicted in Listing 8. After each clock cycle of *fast_clk*, we take the next product and add it to our accumulator. To signalize that the calculation should be stopped, a register *full_flag* is declared which is set automatically if the condition in (line 58) is satisfied. In order to retrace this condition, we take a look into Fig.7. It says that the accumulator is full if the *coeff_Pointer* is one element behind of the *temp_Pointer*. For this, we should declare that the coefficient pointer always starts at the same element as the temporary pointer. This is implemented in Listing 9. During the convolution operation, we then just make use of the pointers *coeff_Pointer* and *nxt_pointer*. These pointers are responsible to fill in the right product element. Here *coeff_Pointer* is pointing to the current coefficient, while the corresponding input sample also taken from *previous_Value*. The product is directly added to the accumulator. Note that *temp_Pointer* is not taken into account during this

process. It acts as a starting point for taking all the sums for the convolution operation. This is explained later in detail. After all FIFO elements are filled, the register *full* is set to '1' as well as a flag, which signalize that no element can be filled into the accumulator anymore.

The final step is explained in Listing 9. On the next rising clock edge of *CLK_Filter*, the output value will overtake the value in the accumulator. After that, the accumulator value is reset as well as *full_flag*. Here, the use of *temp_Pointer* is more obvious and is used as the pointer from which the convolution operation should start. This implementation also guarantees that the buffer is also reset as the condition in line 58 is not satisfied anymore.

```
40 // define full_flag
   reg full_flag;
42
   // Accumulator
44 reg[19:0] accu;
46 // Define pointers
48 // Coefficient Pointer
   reg[4:0] coeff_Pointer;
50
   // Temporary pointers
52
   reg[4:0] temp_Pointer;
54
   // Next Pointer to read coefficient
56 reg[4:0] nxt_Pointer;
58 assign full = (temp_Pointer -1 == coeff_Pointer ) || (temp_Pointer == 0 && coeff_Pointer == 21);
60 always @(posedge fast_clk)
   begin
62   if(full)
64     full_flag <= 1;
     // If Pointer has done a cycle
66   if(full_flag)
68   accu = accu;      // do not change accu value
70   if(!full_flag)
       begin
72       accu = accu + coeffs[coeff_Pointer] * previous_Value[coeff_Pointer];
         coeff_Pointer <= nxt_Pointer;
74
         if(nxt_Pointer == 21)
76         nxt_Pointer <= 0;
         else
78         nxt_Pointer <= nxt_Pointer + 1;
       end
80 end
```

Listing 8: New implementation of the FIR filter using pointers and FIFO buffer (accumulator).

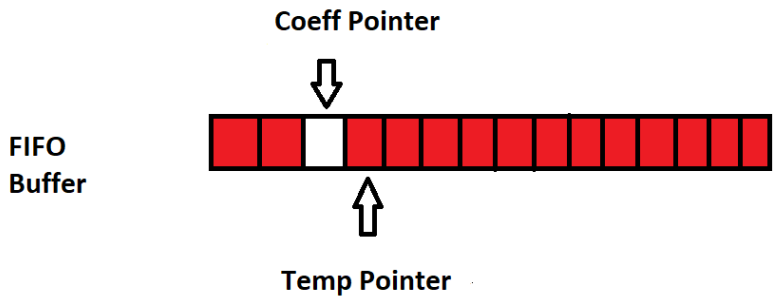


Abbildung 7: FIFO principle using pointers.

```
148    Out_Filtered <= accu;
   accu = 0;
```

```
150
    full_flag <= 0;
152
    if(!full_flag)
154 begin
      coeff_Pointer <= 0;
156   temp_Pointer <= 0;
      nxt_Pointer <= 1;
158 end

160 if(full_flag)
    begin
162           // increase all Pointer

164   if(temp_Pointer == 21)
      begin
166
        temp_Pointer <= 0;
168     coeff_Pointer <= 1;
        nxt_Pointer <= 2;
170   end

172   else if(temp_Pointer == 20)

174   begin

176     temp_Pointer <= temp_Pointer + 1;
        coeff_Pointer <= 0;
178     nxt_Pointer <= 1;
      end
180

182   else
      begin
184     temp_Pointer <= temp_Pointer + 1;
        coeff_Pointer <= temp_Pointer+1;
186     nxt_Pointer <= temp_Pointer + 2;
      end
188 end
```

Listing 9: Operations on rising CLK_Filter edge.

2.5.4. ModelSim Results

The simulation results is depicted in Fig.8 and Fig.9. Taking the testbench file 'FIR_Filter_TB.v' from Appendix B into account, we can verify that this filter functions very well.

Unfortunately, this implementation does not pass Design Vision as this implementation is not already syntheziseable. But we can remember it as future work!

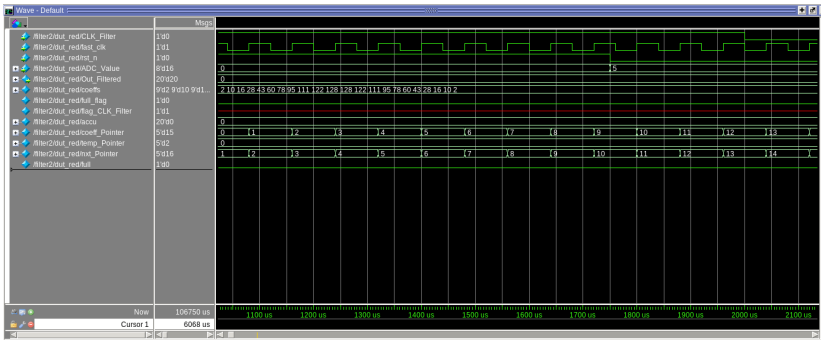


Abbildung 8: Simulation Results of the new FIR filter implementation (1st plot)

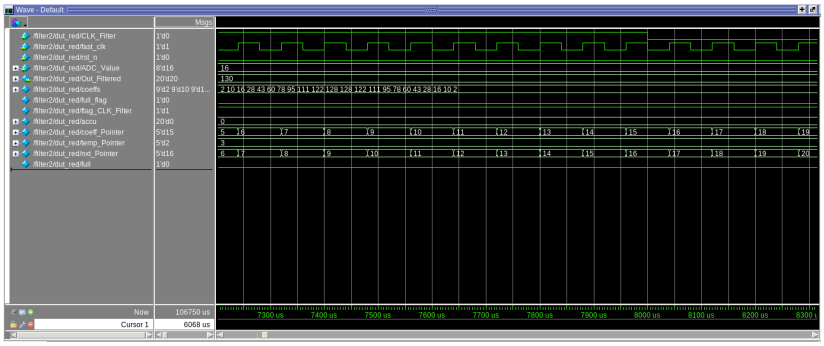


Abbildung 9: Simulation Results of the new FIR filter implementation (2nd plot)

3. Simulation Result

3.1. Modelsim

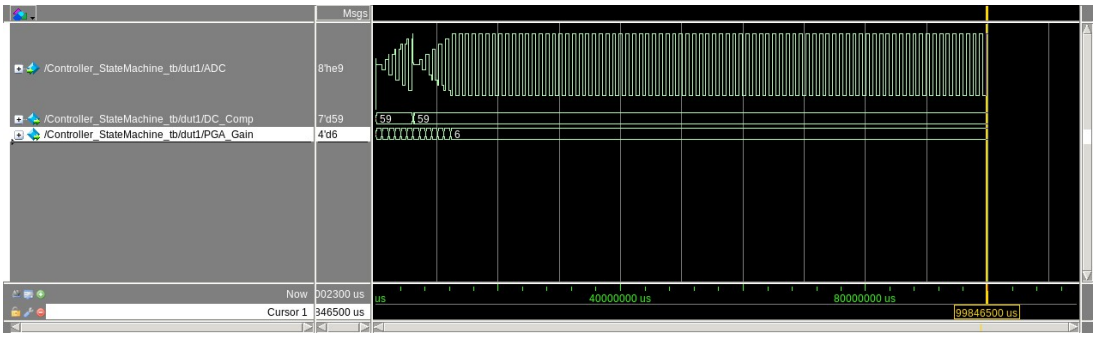


Abbildung 10: Vppg with respect to PGA GAIN and DC Compensation

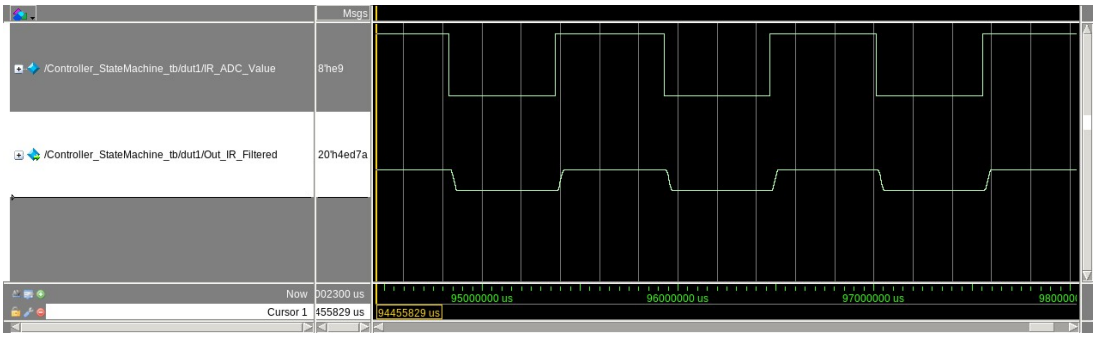


Abbildung 11: ADC stream of IR LED and its filtered stream

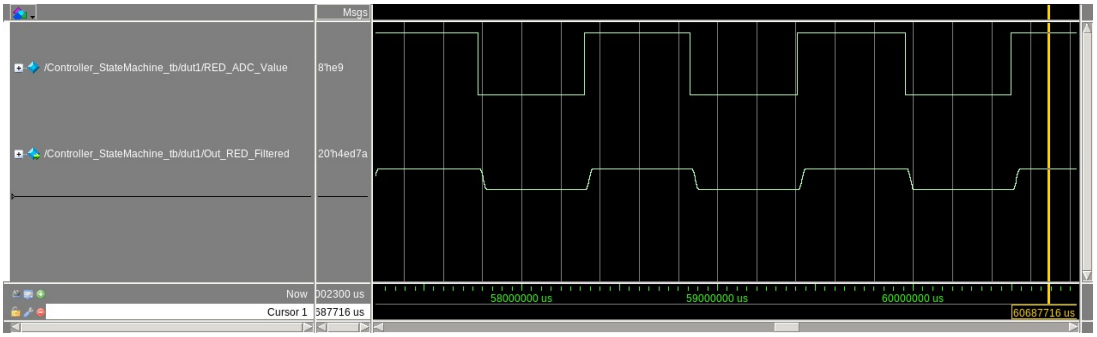


Abbildung 12: ADC stream of RED LED and its filtered stream

3.2. Cadence Virtuoso

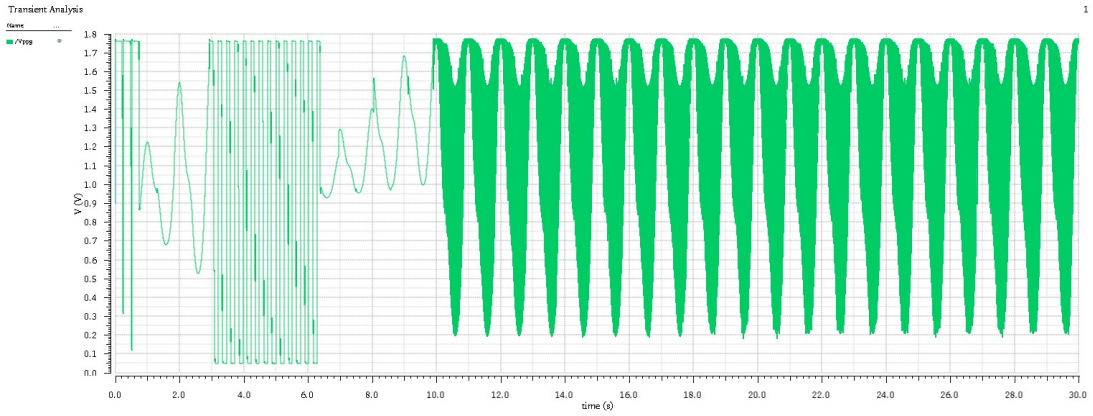


Abbildung 13: Vppg in Cadence

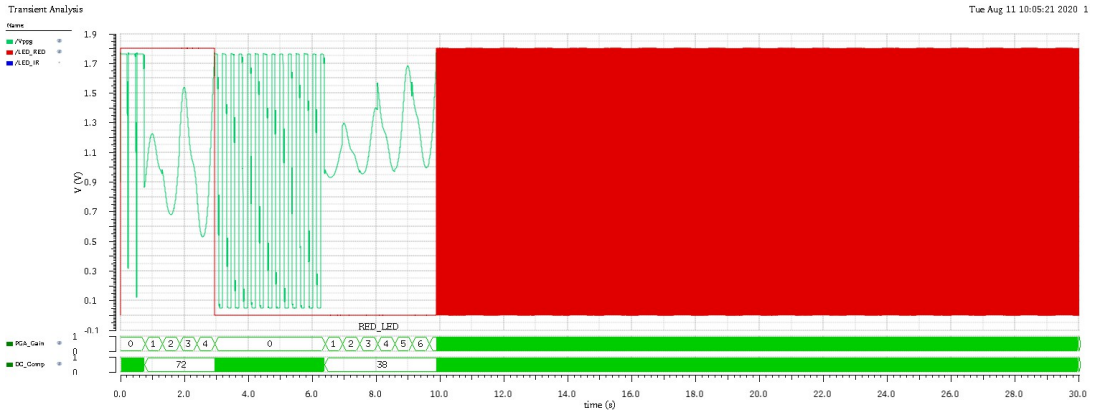


Abbildung 14: Vppg with respect to PGA GAIN and DC Compensation in Cadence

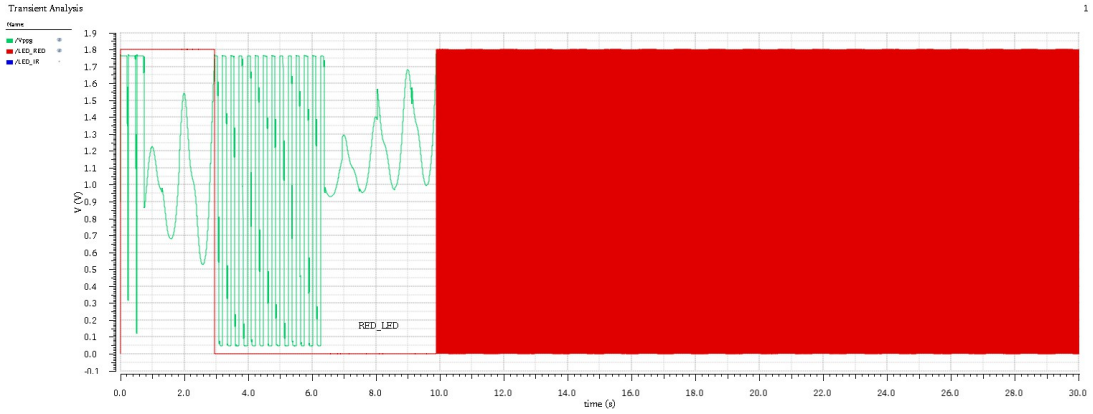


Abbildung 15: RED LED together with Vppg

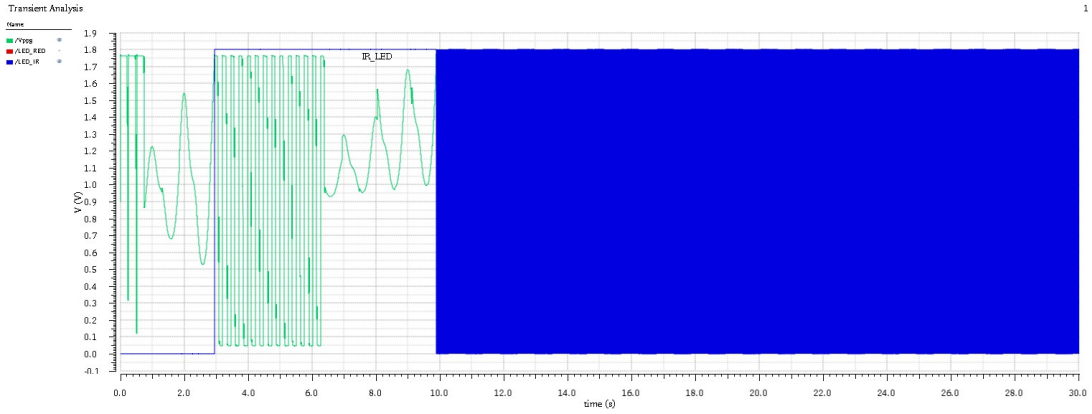


Abbildung 16: IR LED together with Vppg

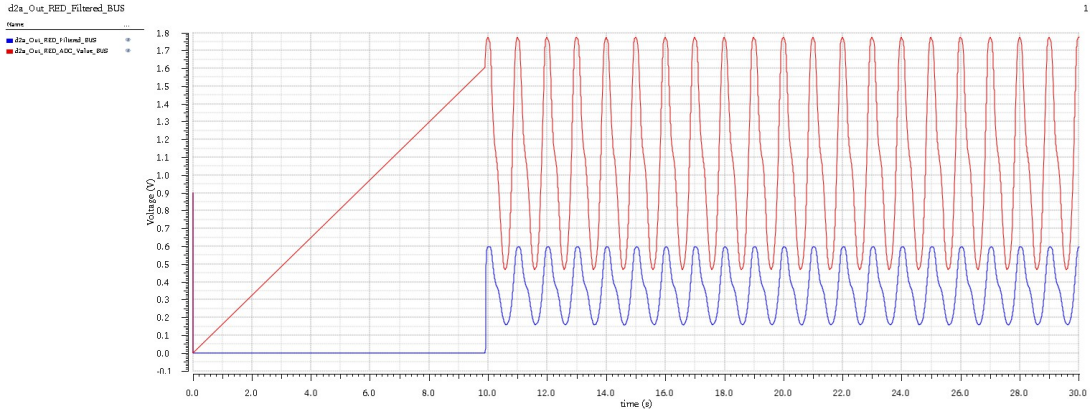


Abbildung 17: ADC stream of RED LED and its filtered stream

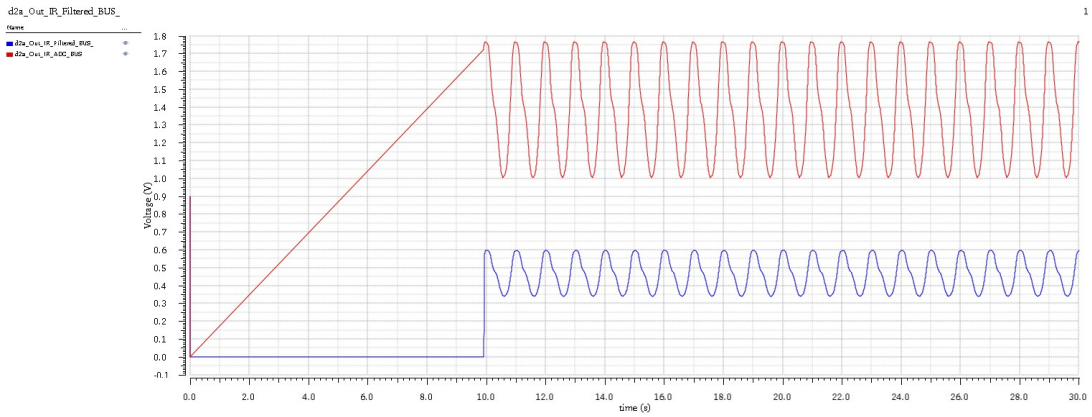


Abbildung 18: ADC stream of IR LED and its filtered stream

3.3. Synopsys

3.3.1. Timing Analysis

CLOCK (rise edge) 1000000.00 (1 KHz) Data arrival time: 11.68 Slack (MET): 999988.25

3.3.2. Area Analysis

Total Area: 33854.521255

3.3.3. Power Analysis

Total Power: 6.4379e-04 mW

3.4. Cadence Encounter

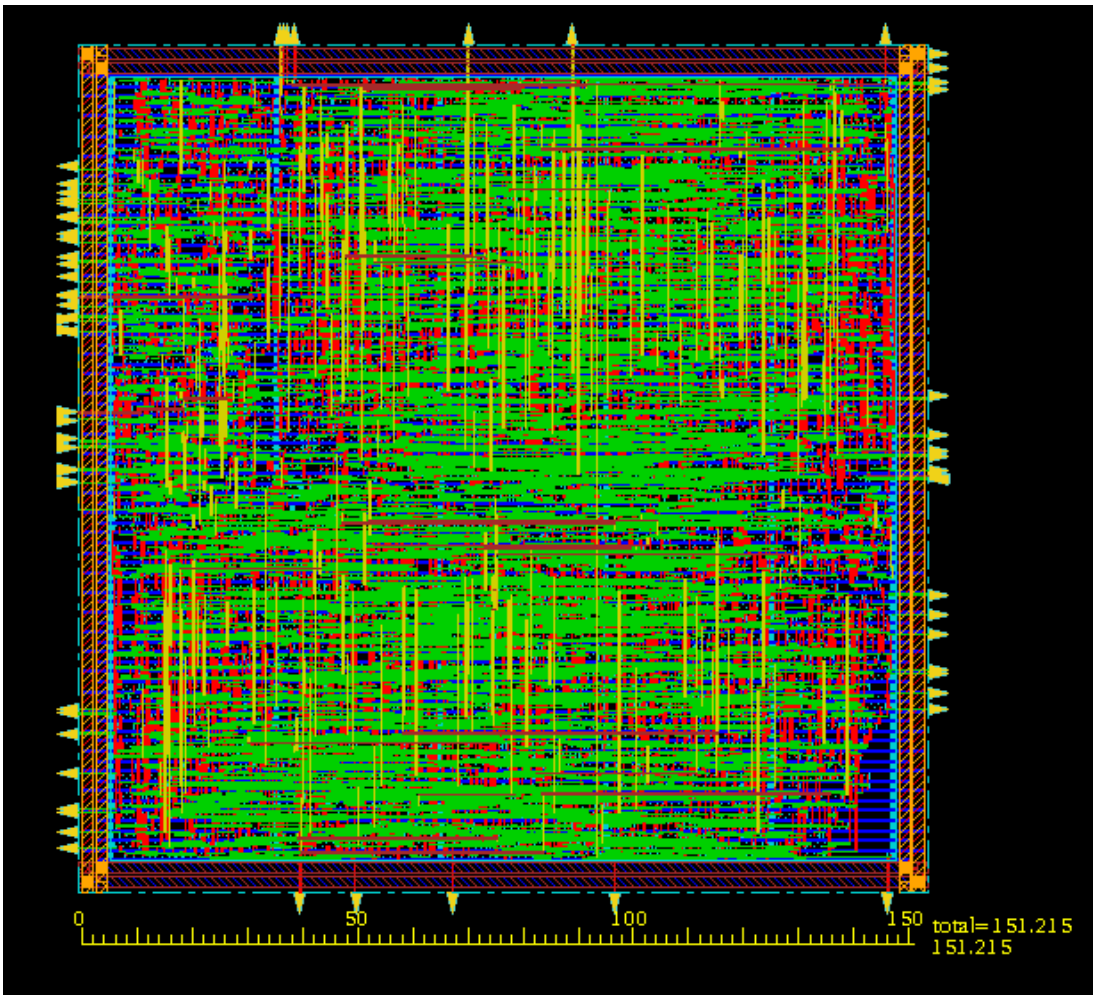


Abbildung 19: Layout generated by Cadence Encounter

We can achieve a smaller area as stated in below sections.

4. Simulation Result, Post Submission

We can not optimize our design before the submission. So we did that after the submission. The big difference is the Vppg range. Now it is not clipped like in the previous section but it can now lie exactly between 0.2V and 1.7V with maxium voltage swing, as it is shown in the figure. 23

Also, the area has also been optimized from 151.215 to 143.493.

We came up to a new idea and also the new optimized implementation for the FIR filter as stated in the section 2.5.

4.1. Modelsim

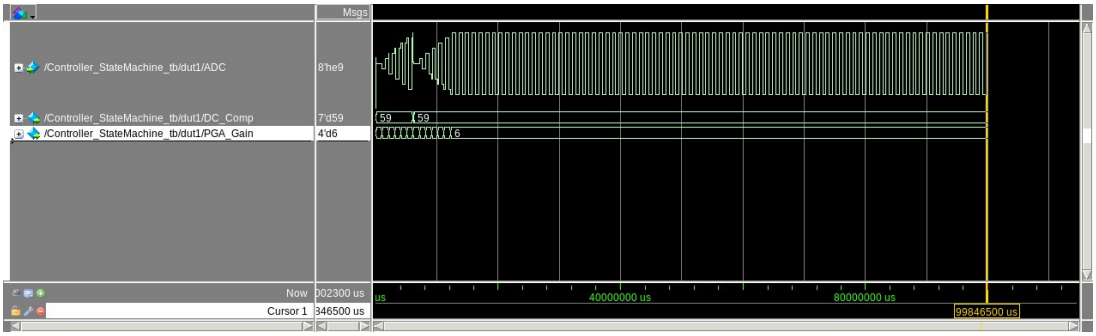


Abbildung 20: Vppg with respect to PGA GAIN and DC Compensation

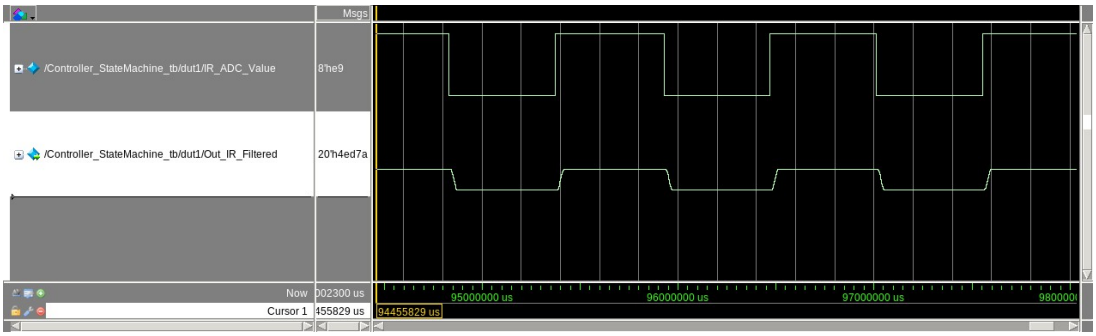


Abbildung 21: ADC stream of IR LED and its filtered stream

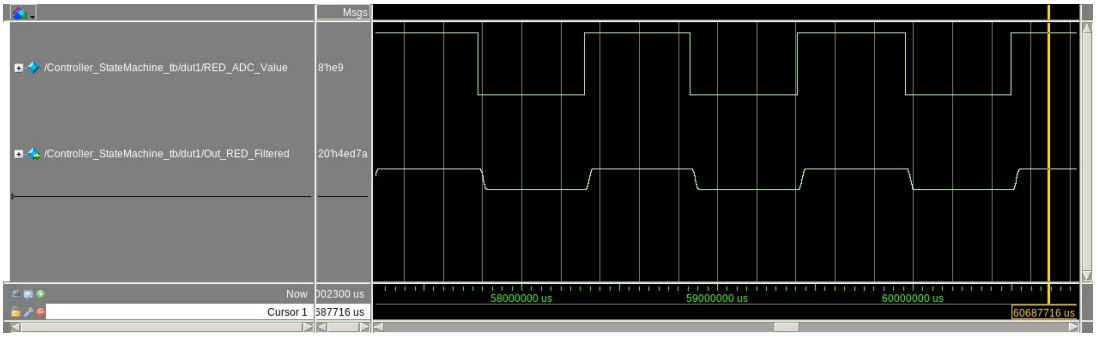


Abbildung 22: ADC stream of RED LED and its filtered stream

4.2. Cadence Virtuoso

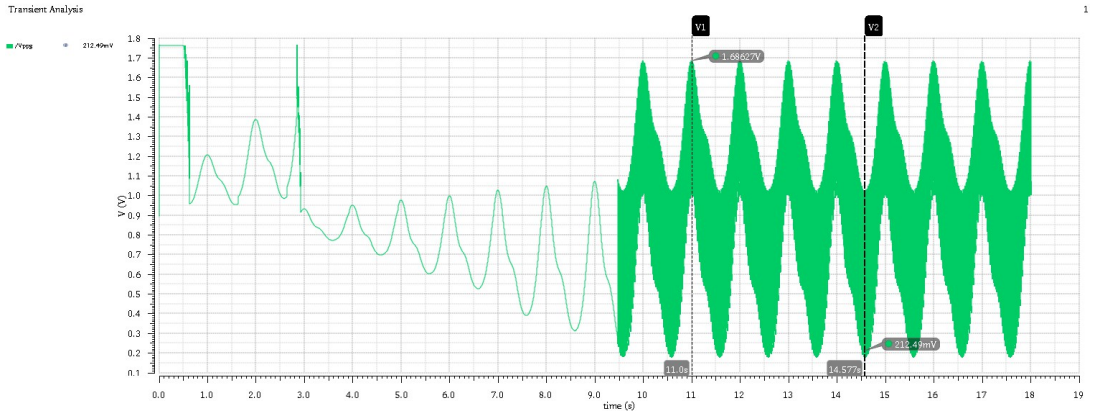


Abbildung 23: Vppg in Cadence

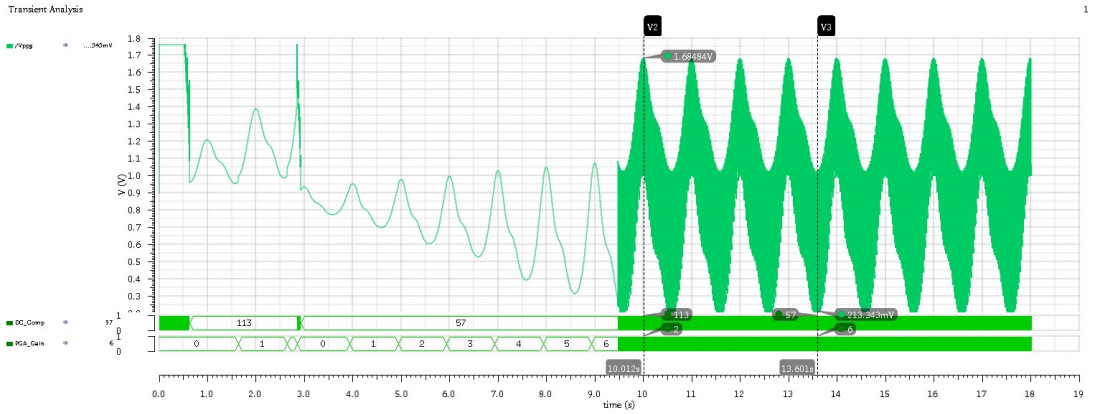


Abbildung 24: Vppg with respect to PGA GAIN and DC Compensation in Cadence

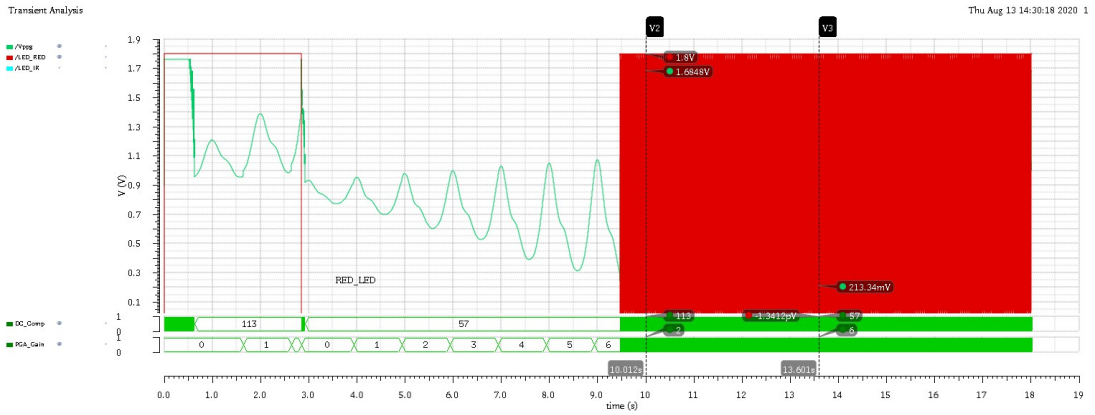


Abbildung 25: RED LED together with Vppg

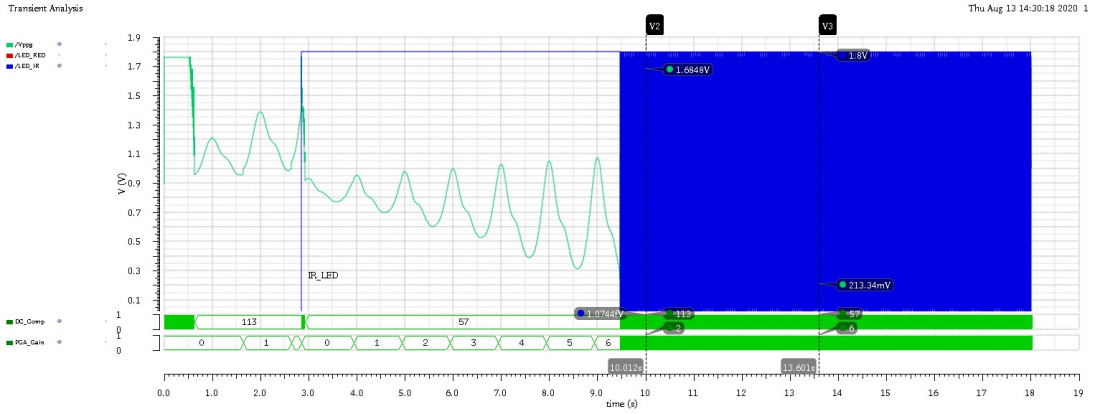


Abbildung 26: IR LED together with Vppg

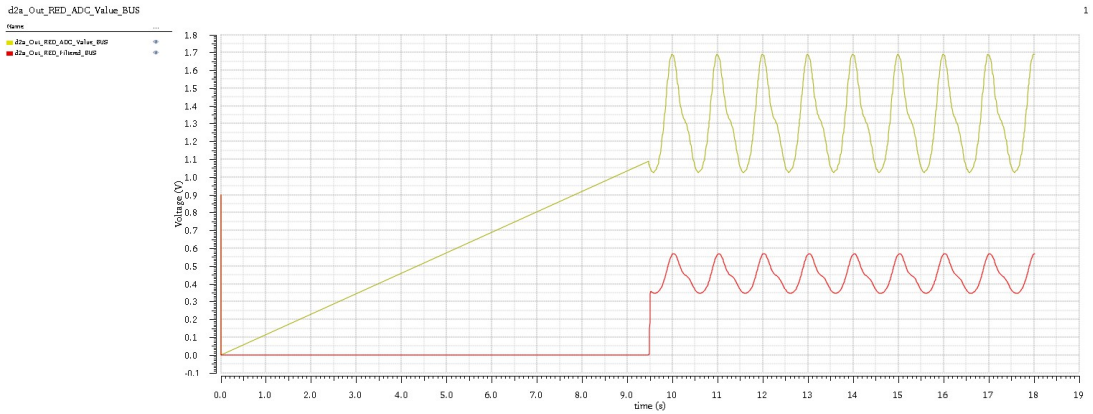


Abbildung 27: ADC stream of RED LED and its filtered stream

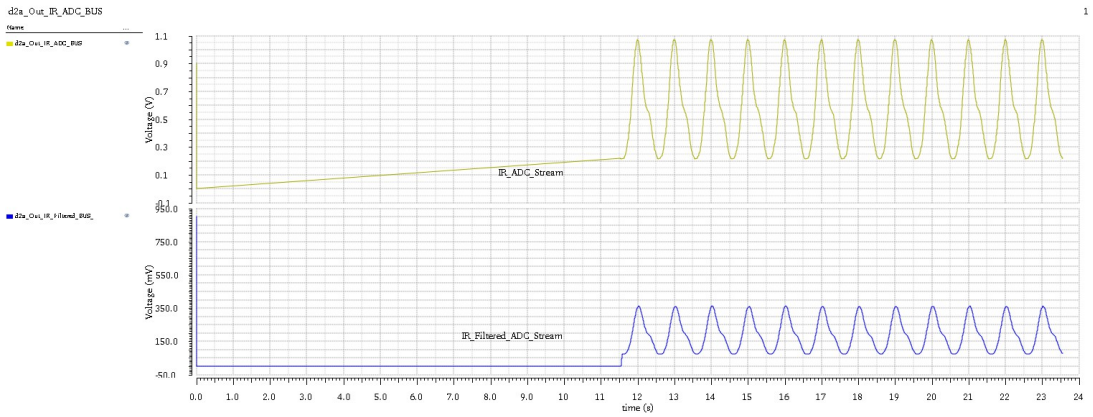


Abbildung 28: ADC stream of IR LED and its filtered stream

4.3. Synopsys

4.3.1. Version 1, Timing, Area and Power Analysis

In this configuration, we want to optimize the area. Therefore, we set the frequecy as low as possbile(1KHz)
CLOCK (rise edge) 1000000.00 (1 KHz) Data arrival time: 13.66 Slack (MET): 999986.19

Total Area: 34546.324300
Total Power: 6.5164e-04 mW

4.3.2. Version 2, Frequency Oriented, Timing, Area and Power Analysis

In this configuration, we want our design be able to operate with high frequency. Therefore, we set the frequecy as high as possible. Of course, if we set it too high, we will violate the **Slack**, or **Slack** will become negative. In other word, the critial path is very long compared to the operating clock frequency.

Therefore, we have to come up to some compromise where the **Slack** is still positive and also as small as possible. Then we came to the operating frequency about 71MHz or roughly 14ns.

CLOCK (rise edge) 14.00 (71 MHz) Data arrival time: 11.78 Slack (MET): 2.04
Total Area: 34467.484293
Total Power: 0.2359 mW

We can see very clearly that the power has been increased significantly when we increase the frequency(from 1KHz to 71 MHz). But the area did not increase that much.

4.4. Cadence Encounter

The configuration of floorPlan is floorPlan -site CORE -r 1 0.95 5.8 5.8 5.8 5.8

So as we can see, with the high concentration of 0.95 we still can generate the layout without any error.

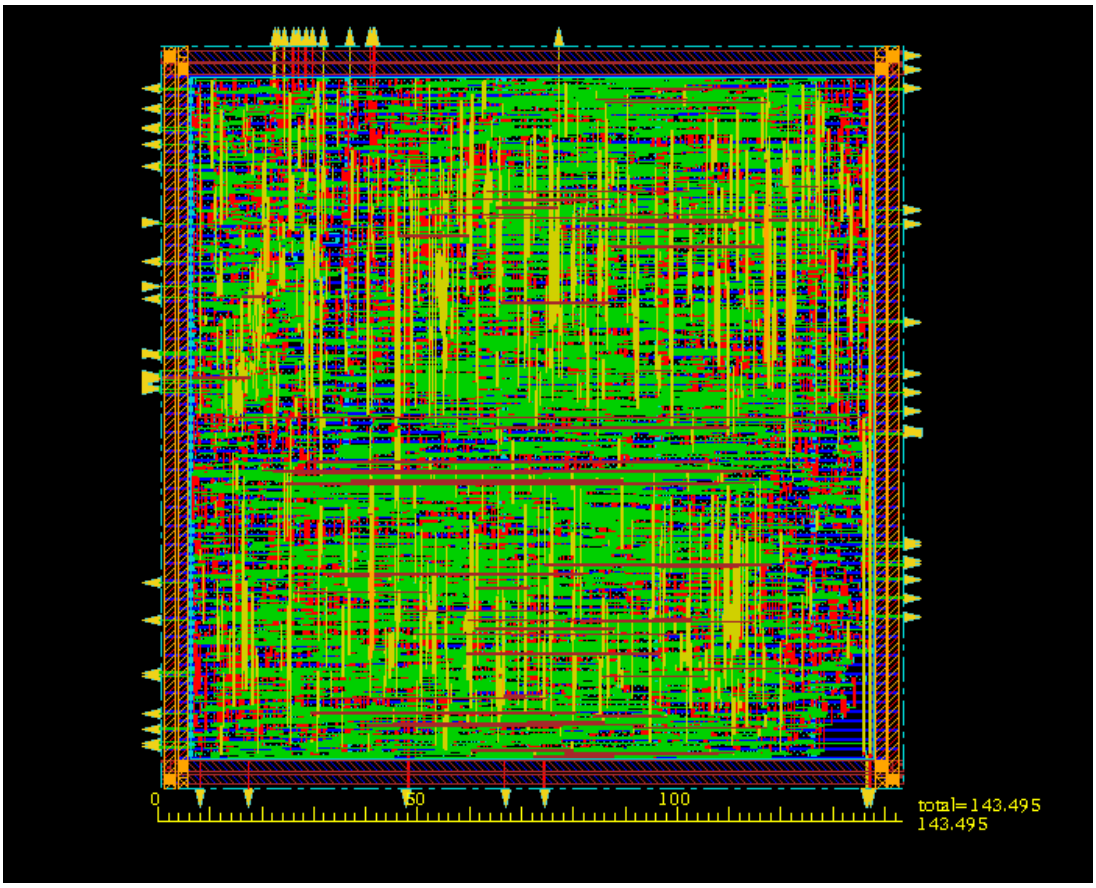


Abbildung 29: Layout for Version 1 generated by Cadence Encounter

5. Conclusion

We show that the principle goal to design the digital block for the pulse oximetry is fully reached. In Sec.2 we explained our design approach with Verilog in detail and depict our simulation results in ModelSim and Cadence Virtuoso in Sec. 3 as well.

As we can see in Fig.24, the signal is lying between 0.2V and 1.7V which is very symmetric signal without any clipping, so this signal can be fed into ADC without any problem. Moreover, the FIR Filter can also work correctly as we can see the figuer 27 and 28, the slope of these two ADC streams(RED and also IR LED) has decreased. So the FIR Filter is functioning as a Low Pass Filter.

With the use of Design Vision synthesis tool we could synthesize a Netlist which can be used for generating layout with Cadence Encounter later on. For the optimization of the floorplaning, we can set the concentration factor to 0.95 without any error and the area is about 143.495 x 143.495.

For the very limit time, we can not do all of the optimization. As described in Sec.2.5.3, we already write a successful Verilog Code for the new FIR filter which only uses only multiplier, but could not be synthesized. We are in the opinion that with small adjustments of the code, we can reduce the area drastically to improve our design. Moreover, the the PIN order, we can also put the PINs at the right position for tape-out.

Appendices

A. Verilog Code

ADC

```
// VerilogA for verilogA, ADC, veriloga

//------ Pipelined ADC------//
`include "discipline.h"
`include "constants.h"
module ADC (in, out, clk);
  parameter integer bit = 8;    // ADC resolution
  parameter real fullscale = 1.8, //supply voltage
  vth = 0.9,            //threshold
  dly = 10n,            // transition delay
  ttime = 1n;           // transition rising time
  input in;           // input analog voltage
  input clk;            // input clock
  output [7:0] out;     // digital vector output
  electrical in, clk;
  electrical [7:0] out;
  real sample;
  integer result[7:0];    // integer array
  genvar i;           // index loop
  analog begin
    @(cross(V(clk) - vth, +1) ) begin
      sample = V(in);
      for (i = bit - 1; i >= 0; i = i -1) begin
        if( sample>vth ) begin
          result[i] = 1.8;
          sample = sample - vth;
        end else begin
          result[i] = 0.0;
        end
      sample = 2.0 * sample;
      end
    end


    for (i = 0; i < bit; i = i + 1 ) begin
      V(out[i]) <+ transition(result[i], dly, ttime);
    end
  end
endmodule
```

Controller State Machine (Controller)

```
module Controller(ADC,clk,Find_Setting, rst_n, LED_Drive,DC_Comp,LED_IR,LED_RED,PGA_Gain, RED_ADC_Value, IR_ADC_Value,
    CLK_Filter);

  output reg [3:0] LED_Drive;
  output reg [6:0] DC_Comp;
  output reg LED_IR;
  output reg LED_RED;
  output reg [3:0] PGA_Gain;
  output reg [7:0] RED_ADC_Value;
  output reg [7:0] IR_ADC_Value;
  output reg CLK_Filter;


  //Input ADC of vppg
  input wire [7:0] ADC;
  //input clk;
  input wire clk;
  //input rst_n;
  input wire rst_n;
  //input Find_Setting;
  input wire Find_Setting;


  //RED OP searchingfi
  reg [6:0] RED_OP;

  //RED Gain searching
  reg [3:0] RED_Gain;

  //IR OP searching
  reg [6:0] IR_OP;
```

```
  //IR Gain searching
32 reg [3:0] IR_Gain;

34

  //Offset voltage compare to 0.9
36 reg [7:0] V_offset;
   reg [3:0] acceptable_offset;
38 reg [7:0] V_min_1s;
   reg [7:0] V_max_1s;
40

42 //wait time_counter
   reg [15:0] Gain_wait_time;
44 reg [15:0] Alternating_Wait_Time;
   reg [15:0] Counter_CLK_Filter;
46 reg [15:0] Min_Max_Wait_Counter;
   reg Min_Max_Found;
48 reg [7:0] Min_Max_Difference;
   reg [7:0] DC_Middle;
50
   //All States
52 reg idle_state;
   reg [3:0] currentState;
54 reg [3:0] previousState;
   reg [3:0] nextState;
56
   parameter Reset_State = 0;
58 parameter Find_New_Settings_State=1;
   parameter RED_LED_OP_Search_State =2;
60 parameter RED_LED_Gain_Search_State =3 ;
   parameter INRED_OP_Search_State =4;
62 parameter INRED_Gain_Search_State =5 ;

64 parameter Alternating_RED_LED_State = 6;
   parameter Alternating_IR_LED_State = 7;
66 parameter Alternating_LED_State = 8;

68 parameter FIR_Filter_State = 9;
   parameter Idle_State = 10;
70
   //Instantiate FIR_Filter Module
72 /*FIR_Filter FIR_Filter_RED(
   .CLK_Filter(CLK_Filter),
74 .rst_n(rst_n),
   .ADC_Value(RED_ADC_Value),
76 .Out_RED_Filtered(Out_RED_Filtered)
   );
78
   FIR_Filter FIR_Filter_IR(
80 .CLK_Filter(CLK_Filter),
   .rst_n(rst_n),
82 .ADC_Value(IR_ADC_Value),
   .Out_RED_Filtered(Out_IR_Filtered)
84 );*/

86 always@(posedge clk) begin
     if(rst_n ==1)
88     begin
       Counter_CLK_Filter=0;
90     CLK_Filter = 0;
       end
92   if(Counter_CLK_Filter==1)
       begin
94     Counter_CLK_Filter = 0;
       if (CLK_Filter == 1)
96       CLK_Filter=0;
       else
98       CLK_Filter =1;
       end
100  Counter_CLK_Filter = Counter_CLK_Filter+1;
   end
102

104 always@(posedge clk) begin        //MARKER: HAS BEEN MODIFIED only for SYNTHESIS, MUST ADJUST BACK
     if (Find_Setting==1)
106    begin
       currentState = Find_New_Settings_State;
108    end
     if (rst_n == 1)
110    begin
       currentState  = Reset_State;
112    end
```

```
case(currentState)
    Reset_State:
      begin
        LED_Drive = 10;
        DC_Comp = 50; // DC_Comp = 0
        LED_IR = 0; //Turn OFF IN_RED
        LED_RED = 1 ; //Turn ON RED_LED
        PGA_Gain = 0; // GAIN = 0
        V_offset = 0;
        acceptable_offset = 10;
        Gain_wait_time = 545;
        Min_Max_Wait_Counter=0;
        V_min_1s = 255;
        V_max_1s = 0;
        DC_Middle = 0;
        Min_Max_Found=0;
        Min_Max_Difference=0;
        Alternating_Wait_Time = 10;
        IR_ADC_Value = 0;
        RED_ADC_Value = 0;
        currentState = nextState; // go to where it was predefine, this Reset is just a middle state (bergang)

      end

    Find_New_Settings_State:
      begin
        currentState =  RED_LED_OP_Search_State;
      end

    RED_LED_OP_Search_State:
      begin
        if (Min_Max_Wait_Counter<10)
          begin
          if(ADC < V_min_1s)
            V_min_1s = ADC;
          if(ADC > V_max_1s)
            V_max_1s = ADC;
          end
        else
          begin
          Min_Max_Difference = V_max_1s-V_min_1s;
          DC_Middle =  (Min_Max_Difference >>2) + V_min_1s;
          if(DC_Middle < 127)
            V_offset = 127-DC_Middle;
          else
            V_offset = DC_Middle-127;
          V_max_1s = 0;
          V_min_1s = 255;
          Min_Max_Wait_Counter = 0;
          Min_Max_Found = 1;
          end
        //Stop searching, save for RED-LED DC point
        if(V_offset<= acceptable_offset & Min_Max_Found == 1)
          begin
          RED_OP = DC_Comp;
          Min_Max_Found=0;
          currentState = RED_LED_Gain_Search_State;
          end
        //Seaching for DC_Comp
         if ( Min_Max_Found == 1 & DC_Middle < 127)
          begin
          DC_Comp = DC_Comp-1;
          Min_Max_Found = 0;
          end
        else if (Min_Max_Found == 1 & DC_Middle > 127)
          begin
          DC_Comp = DC_Comp+1;
          Min_Max_Found = 0;
          end

        Min_Max_Wait_Counter = Min_Max_Wait_Counter+1;
      end

    RED_LED_Gain_Search_State:
      begin
      if (Min_Max_Wait_Counter<1000)
          begin
          if(ADC < V_min_1s)
            V_min_1s = ADC;
          if(ADC > V_max_1s)
            V_max_1s = ADC;
          end
```

```
          else
            begin
            Min_Max_Difference = V_max_1s-V_min_1s;
            DC_Middle =  (Min_Max_Difference >>2) + V_min_1s;
            if(DC_Middle < 127)
              V_offset = 127-DC_Middle;
            else
              V_offset = DC_Middle-127;
            V_max_1s = 0;
            V_min_1s = 255;
            Min_Max_Wait_Counter = 0;
            Min_Max_Found = 1;
            end



        if(V_max_1s < 200 & V_min_1s > 5 & Min_Max_Found == 1) // increase gain every 0.545(s)
          begin
          PGA_Gain = PGA_Gain + 1;
          Min_Max_Found = 0;
          end
        //stop searching, store RED-Gain
        if (V_max_1s > 200 | V_min_1s < 5)
          begin
          //if(PGA_Gain>=2)
            //PGA_Gain=2;
          RED_Gain = PGA_Gain;
          Min_Max_Found = 0;
          currentState = Reset_State; //Reset Register before going to INRED
          nextState = INRED_OP_Search_State;
          end
        Min_Max_Wait_Counter = Min_Max_Wait_Counter+1;
        end

      INRED_OP_Search_State:
        begin
          LED_IR = 1; //Turn ON IN_RED
          LED_RED = 0;

          if (Min_Max_Wait_Counter<10)
            begin
            if(ADC < V_min_1s)
              V_min_1s = ADC;
            if(ADC > V_max_1s)
              V_max_1s = ADC;
            end
          else
            begin
            Min_Max_Difference = V_max_1s-V_min_1s;
            DC_Middle =  (Min_Max_Difference >>2) + V_min_1s;
            if(DC_Middle < 127)
              V_offset = 127-DC_Middle;
            else
              V_offset = DC_Middle-127;
            V_max_1s = 0;
            V_min_1s = 255;
            Min_Max_Wait_Counter = 0;
            Min_Max_Found = 1;
            end


          //Stop searching, save for IR-LED DC point
          if(V_offset<= acceptable_offset & Min_Max_Found == 1)
            begin
            IR_OP = DC_Comp;
            Min_Max_Found=0;
            currentState = INRED_Gain_Search_State;
            end


          //Seaching for DC_Comp
           if ( Min_Max_Found == 1 & DC_Middle < 127)
            begin
            DC_Comp = DC_Comp-1;
            Min_Max_Found = 0;
            end
          else if (Min_Max_Found == 1 & DC_Middle > 127)
            begin
            DC_Comp = DC_Comp+1;
            Min_Max_Found = 0;
            end
```

```
278           Min_Max_Wait_Counter = Min_Max_Wait_Counter+1;
280         end
282       INRED_Gain_Search_State:
284         begin
286           if (Min_Max_Wait_Counter<1000)
              begin
288           if(ADC < V_min_1s)
                V_min_1s = ADC;
290           if(ADC > V_max_1s)
                V_max_1s = ADC;
              end
292           else
              begin
294           Min_Max_Difference = V_max_1s-V_min_1s;
              DC_Middle =  (Min_Max_Difference >>2) + V_min_1s;
296           if(DC_Middle < 127)
                V_offset = 127-DC_Middle;
298           else
                V_offset = DC_Middle-127;
              V_max_1s = 0;
300           V_min_1s = 255;
              Min_Max_Wait_Counter = 0;
302           Min_Max_Found = 1;
              end
304
306         if(V_max_1s < 240 & V_min_1s > 40 & Min_Max_Found == 1) //
              begin
308           PGA_Gain = PGA_Gain +1;
              Min_Max_Found = 0;
310           end
            //stop searching, store RED-Gain
312         if (V_max_1s > 240 | V_min_1s < 40)
              begin
314           IR_Gain = PGA_Gain;
              //if(IR_Gain>=6)
316             //IR_Gain=6;
              Min_Max_Found = 0;
318           currentState = Alternating_LED_State;
              end
320         Min_Max_Wait_Counter = Min_Max_Wait_Counter+1;
            end
322
324       Alternating_LED_State:
            begin
326         if(Alternating_Wait_Time >= 10) // 20 x 0.5 = 10ms, means we alternate every 10ms, means 100Hz
              begin
328           Alternating_Wait_Time = 0;
              if(previousState == Alternating_IR_LED_State)
330             currentState= Alternating_RED_LED_State;
              else if(previousState == Alternating_RED_LED_State)
332             currentState= Alternating_IR_LED_State;
              else
334             currentState= Alternating_RED_LED_State;
              end
            //Continuously out put the ADC based on which LED is currently on
336         if(previousState == Alternating_IR_LED_State)
              begin
338           IR_ADC_Value = ADC; //output ADC of IR_LED
              end
340         if(previousState == Alternating_RED_LED_State)
              begin
342           RED_ADC_Value = ADC;  //output RED-ADCoutput ADC of RED_LED
              end
344         //Increase timing-counter by 1
            Alternating_Wait_Time = Alternating_Wait_Time+1;
346         end
          Alternating_RED_LED_State:
348         begin
            LED_IR = 0; //Turn ON RED_LED
350         LED_RED = 1;
            DC_Comp = RED_OP;
352         PGA_Gain = RED_Gain;
            currentState = Alternating_LED_State;
354         previousState = Alternating_RED_LED_State;
            end
356
          Alternating_IR_LED_State:
358         begin
```

```
        LED_IR = 1; //Turn ON IR_LED
        LED_RED = 0;
        DC_Comp = IR_OP;
        PGA_Gain = IR_Gain;
        currentState = Alternating_LED_State;
        previousState = Alternating_IR_LED_State;
        end

      FIR_Filter_State:
        begin
        idle_state =1 ;
        end

    endcase
end
endmodule
```

FIR Filter Optimized

```
//Verilog HDL for "HDL_Lab_10", "FIR_Filter" "functional"
module FIR_Filter_Optimized(
  input CLK_Filter,
  input rst_n,
  input wire[7:0] ADC_Value,
  output reg[19:0] Out_Filtered);

// parameter N = 4;
wire signed[8:0] coeffs[21:0];

//
reg [15:0] previous_Value[21:0];

// define multiplier

// FIFO
reg [19:0] product[21:0];

assign coeffs[0]=2;
assign coeffs[1]=10;
assign coeffs[2]=16;
assign coeffs[3]=28;
assign coeffs[4]=43;
assign coeffs[5]=60;
assign coeffs[6]=78;
assign coeffs[7]=95;
assign coeffs[8]=111;
assign coeffs[9]=122;
assign coeffs[10]= 128;


always @(posedge CLK_Filter or posedge rst_n)
begin
    if(rst_n)
        begin

  product[0] <= 0;
  product[1] <= 0;
  product[2] <= 0;
  product[3] <= 0;
  product[4] <= 0;
  product[5] <= 0;
  product[6] <= 0;
  product[7] <= 0;
  product[8] <= 0;
  product[9] <= 0;
  product[10] <= 0;
  product[11] <= 0;
  product[12] <= 0;
  product[13] <= 0;
  product[14] <= 0;
  product[15] <= 0;
  product[16] <= 0;
  product[17] <= 0;
  product[18] <= 0;
  product[19] <= 0;
  product[20] <= 0;
  product[21] <= 0;

  previous_Value[21]     <= 0;
        previous_Value[20]     <= 0;
  previous_Value[19]     <= 0;
        previous_Value[18]     <= 0;
  previous_Value[17]     <= 0;
        previous_Value[16]     <= 0;
  previous_Value[15]     <= 0;
        previous_Value[14]     <= 0;
        previous_Value[13]     <= 0;
        previous_Value[12]     <= 0;
  previous_Value[11]     <= 0;
        previous_Value[10]     <= 0;
        previous_Value[9]     <= 0;
        previous_Value[8]     <= 0;
  previous_Value[7]     <= 0;
        previous_Value[6]     <= 0;
        previous_Value[5]     <= 0;
        previous_Value[4]     <= 0;
  previous_Value[3]     <= 0;
        previous_Value[2]     <= 0;
        previous_Value[1]     <= 0;
```

```
        previous_Value[0]     <= 0;

        Out_Filtered    <= 0;
        end

    else
        begin


  previous_Value[21]     <= previous_Value[20];
        previous_Value[20]     <= previous_Value[19];
  previous_Value[19]     <= previous_Value[18];
        previous_Value[18]     <= previous_Value[17];
        previous_Value[17]     <= previous_Value[16];
        previous_Value[16]     <= previous_Value[15];
  previous_Value[15]     <= previous_Value[14];
        previous_Value[14]     <= previous_Value[13];
        previous_Value[13]     <= previous_Value[12];
        previous_Value[12]     <= previous_Value[11];
  previous_Value[11]     <= previous_Value[10];
        previous_Value[10]     <= previous_Value[9];
        previous_Value[9]     <= previous_Value[8];
        previous_Value[8]     <= previous_Value[7];
  previous_Value[7]     <= previous_Value[6];
        previous_Value[6]     <= previous_Value[5];
        previous_Value[5]     <= previous_Value[4];
        previous_Value[4]     <= previous_Value[3];
        previous_Value[3]     <= previous_Value[2];
        previous_Value[2]     <= previous_Value[1];
        previous_Value[1]     <= previous_Value[0];
        previous_Value[0]     <= ADC_Value;

  product[0] <= coeffs[0] * previous_Value[0];
    product[1] <= coeffs[1] * previous_Value[1];
    product[2] <= coeffs[2] * previous_Value[2];
    product[3] <= coeffs[3] * previous_Value[3];
    product[4] <= coeffs[4] * previous_Value[4];
    product[5] <= coeffs[5] * previous_Value[5];
    product[6] <= coeffs[6] * previous_Value[6];
    product[7] <= coeffs[7] * previous_Value[7];
    product[8] <= coeffs[8] * previous_Value[8];
    product[9] <= coeffs[9] * previous_Value[9];
    product[10] <= coeffs[10] * previous_Value[10];
    product[11] <= coeffs[10] * previous_Value[11];
    product[12] <= coeffs[9] * previous_Value[12];
    product[13] <= coeffs[8] * previous_Value[13];
    product[14] <= coeffs[7] * previous_Value[14];
    product[15] <= coeffs[6] * previous_Value[15];
    product[16] <= coeffs[5] * previous_Value[16];
    product[17] <= coeffs[4] * previous_Value[17];
    product[18] <= coeffs[3] * previous_Value[18];
    product[19] <= coeffs[2] * previous_Value[19];
    product[20] <= coeffs[1] * previous_Value[20];
    product[21] <= coeffs[0] * previous_Value[21];

   Out_Filtered <= (product[0] + product[1] +
                            product[2] + product[3] + product[4] + product[5] +
                            product[6] + product[7] + product[8] + product[9] +
                            product[10] + product[11] + product[12] + product[13] +
                            product[14] + product[15] + product[16] + product[17] +
                            product[18] + product[19] + product[20] + product[21]);


  end

end
endmodule
```

Top Module Controller and FIR

```
module Top_Module_Controller_and_FIR(ADC, clk, Find_Setting, rst_n, LED_Drive, DC_Comp, LED_IR, LED_RED, PGA_Gain,
    Out_IR_Filtered, Out_RED_Filtered);

  //IN-OUT of CONTROLLER
  output wire [3:0] LED_Drive;
  output wire [6:0] DC_Comp;
  output wire LED_IR;
  output wire LED_RED;
  output wire [3:0] PGA_Gain;
  wire [7:0] RED_ADC_Value;
  wire [7:0] IR_ADC_Value;
  wire CLK_Filter;

  //input wire fast_clk;
  input wire [7:0] ADC;
  input wire clk;
  input wire rst_n;
  input wire Find_Setting;


  //IN-OUT OF FIR
  output wire[19:0] Out_IR_Filtered;
  output wire[19:0] Out_RED_Filtered;



  Controller_StateMachine Controller(
  .LED_Drive(LED_Drive),
  .DC_Comp(DC_Comp),
  .LED_IR(LED_IR),
  .LED_RED(LED_RED),
  .PGA_Gain(PGA_Gain),
  .RED_ADC_Value(RED_ADC_Value),
  .IR_ADC_Value(IR_ADC_Value),
  .CLK_Filter(CLK_Filter),
  .ADC(ADC),
  .clk(clk),
  .rst_n(rst_n),
  .Find_Setting(Find_Setting));


  FIR_Filter_Optimized FIR_Filter_RED(
  .CLK_Filter(CLK_Filter),
  .rst_n(rst_n),
  .ADC_Value(RED_ADC_Value),
  .Out_Filtered(Out_RED_Filtered)
  );


  FIR_Filter_Optimized FIR_Filter_IR(
  .CLK_Filter(CLK_Filter),
  .rst_n(rst_n),
  .ADC_Value(IR_ADC_Value),
  .Out_Filtered(Out_IR_Filtered)
  );

endmodule
```

Top Module Testbench(Testbench for the entire system

```
`timescale 1us/1us
module Controller_StateMachine_tb();

  //Input controller
  reg  clk;
  reg fast_clk;
  wire [7:0] Vppg;
  reg rst_n;
  reg Find_Setting;



  //output Controller
  wire [3:0] LED_Drive;
  wire [6:0] DC_Comp;
  wire LED_IR;
  wire LED_RED;
  wire [3:0] PGA_Gain;
  wire [19:0] Out_RED_Filtered;
  wire [19:0] Out_IR_Filtered;
  wire CLK_Filter;
  //output Fingerclip




  //Controller_Dat dut1( .RED_ADC_Value(RED_ADC_Value), .LED_Drive(LED_Drive), .DC_Comp(DC_Comp), .LED_IR(LED_IR), .
     LED_RED(LED_RED), .PGA_Gain(PGA_Gain), .ADC(Vppg), .clk(clk));
  Fingerclip_Model dut2(.Vppg(Vppg), .DC_Comp(DC_Comp), .PGA_Gain(PGA_Gain));
  Top_Module_Controller_and_FIR dut1(.LED_Drive(LED_Drive), .DC_Comp(DC_Comp), .LED_IR(LED_IR), .LED_RED(LED_RED),
.PGA_Gain(PGA_Gain), .Out_IR_Filtered(Out_IR_Filtered), .Out_RED_Filtered(Out_RED_Filtered), .ADC(Vppg), .clk(clk),.rst_n
     (rst_n),.Find_Setting(Find_Setting));
// Instantiate the Unit Under Test (UUT)




  initial begin
    Find_Setting = 0;
    rst_n = 1;
    clk = 0;


    #550 rst_n = 0;
    #600 Find_Setting = 1;
    #1150 Find_Setting = 0;


    #100000000 $stop;
  end




  always #500 clk = !clk;  //Clock 1khz controller
  always #40 fast_clk = !fast_clk; //Clock of Filter, 22 times faster than Sampling Frequency(500Hz)

endmodule
```

B. Optimization Verilog Code

FIR Filter Optimization Code (less Multipliers, using Pointers

```
//Verilog HDL for "HDL_Lab_10", "FIR_Filter" "functional"
module FIR_Filter_Optimized(
  input CLK_Filter,
  input fast_clk,
  input rst_n,
  input wire[7:0] ADC_Value,
  output reg[19:0] Out_Filtered);

// parameter N = 4;
wire signed[8:0] coeffs[21:0];

//
reg [15:0] previous_Value[21:0];

reg full_flag;

reg flag_CLK_Filter;

// define coefficients


assign coeffs[0]=2;
assign coeffs[1]=10;
assign coeffs[2]=16;
assign coeffs[3]=28;
assign coeffs[4]=43;
assign coeffs[5]=60;
assign coeffs[6]=78;
assign coeffs[7]=95;
assign coeffs[8]=111;
assign coeffs[9]=122;
assign coeffs[10]= 128;
assign coeffs[11]=128;
assign coeffs[12]=122;
assign coeffs[13]=111;
assign coeffs[14]=95;
assign coeffs[15]=78;
assign coeffs[16]=60;
assign coeffs[17]=43;
assign coeffs[18]=28;
assign coeffs[19]=16;
assign coeffs[20]=10;
assign coeffs[21]= 2;


// Accumulator
reg[19:0] accu;

// Define pointers

// Coefficient Pointer
reg[4:0] coeff_Pointer;

// Temporary pointers

reg[4:0] temp_Pointer;

// Next Pointer to read coefficient
reg[4:0] nxt_Pointer;

assign full = (temp_Pointer -1 == coeff_Pointer ) || (temp_Pointer == 0 && coeff_Pointer == 21);

always @(posedge fast_clk)
begin
  if(full)

    full_flag <= 1;
  // If Pointer has done a cycle
  if(full_flag)

  accu = accu;      // do not change accu value

  if(!full_flag)
    begin
      accu = accu + coeffs[coeff_Pointer] * previous_Value[coeff_Pointer];
```

```
76        coeff_Pointer <= nxt_Pointer;
77
78        if(nxt_Pointer == 21)
79          nxt_Pointer <= 0;
80        else
81          nxt_Pointer <= nxt_Pointer + 1;
82      end
83  end
84
85
86  always @(posedge CLK_Filter or posedge rst_n)
87  begin
88      if(rst_n)
89      begin
90
91    accu = 0;
92
93    coeff_Pointer <= 0;
94    temp_Pointer <= 0;
95    nxt_Pointer <= 1;
96
97    previous_Value[21]     <= 0;
98          previous_Value[20]     <= 0;
99    previous_Value[19]     <= 0;
100         previous_Value[18]     <= 0;
101   previous_Value[17]     <= 0;
102         previous_Value[16]     <= 0;
103   previous_Value[15]     <= 0;
104         previous_Value[14]     <= 0;
105         previous_Value[13]     <= 0;
106         previous_Value[12]     <= 0;
107   previous_Value[11]     <= 0;
108         previous_Value[10]     <= 0;
109         previous_Value[9]      <= 0;
110         previous_Value[8]      <= 0;
111   previous_Value[7]      <= 0;
112         previous_Value[6]      <= 0;
113         previous_Value[5]      <= 0;
114         previous_Value[4]      <= 0;
115   previous_Value[3]      <= 0;
116         previous_Value[2]      <= 0;
117         previous_Value[1]      <= 0;
118         previous_Value[0]      <= 0;
119
120         Out_Filtered     <= 0;
121   full_flag <= 0;
122
123     end
124
125     else
126     begin
127   // Shift every value to the next register place
128   previous_Value[21]     <= previous_Value[20];
129         previous_Value[20]     <= previous_Value[19];
130   previous_Value[19]     <= previous_Value[18];
131         previous_Value[18]     <= previous_Value[17];
132         previous_Value[17]     <= previous_Value[16];
133         previous_Value[16]     <= previous_Value[15];
134   previous_Value[15]     <= previous_Value[14];
135         previous_Value[14]     <= previous_Value[13];
136         previous_Value[13]     <= previous_Value[12];
137         previous_Value[12]     <= previous_Value[11];
138   previous_Value[11]     <= previous_Value[10];
139         previous_Value[10]     <= previous_Value[9];
140         previous_Value[9]      <= previous_Value[8];
141         previous_Value[8]      <= previous_Value[7];
142   previous_Value[7]      <= previous_Value[6];
143         previous_Value[6]      <= previous_Value[5];
144         previous_Value[5]      <= previous_Value[4];
145         previous_Value[4]      <= previous_Value[3];
146         previous_Value[3]      <= previous_Value[2];
147         previous_Value[2]      <= previous_Value[1];
148         previous_Value[1]      <= previous_Value[0];
149         previous_Value[0]      <= ADC_Value;
150
151   Out_Filtered <= accu;
152   accu = 0;
153
154   full_flag <= 0;
155   flag_CLK_Filter <= 1;
156
157   if(!full_flag)
```

```
  begin
    coeff_Pointer <= 0;
    temp_Pointer <= 0;
    nxt_Pointer <= 1;
  end

  if(full_flag)
  begin
          // increase all Pointer

    if(temp_Pointer == 21)
    begin

      temp_Pointer <= 0;
      coeff_Pointer <= 1;
      nxt_Pointer <= 2;
    end

    else if(temp_Pointer == 20)

    begin

      temp_Pointer <= temp_Pointer + 1;
      coeff_Pointer <= 0;
      nxt_Pointer <= 1;
    end


    else
    begin
      temp_Pointer <= temp_Pointer + 1;
      coeff_Pointer <= temp_Pointer+1;
      nxt_Pointer <= temp_Pointer + 2;
    end
  end


       // Set flags


   end
end
endmodule
```

FIR Filter Testbench

```
`timescale 1us / 1us

module Filter_tst;

// Inputs
reg CLK_Filter;
reg rst_n;
reg [7:0] RED_ADC_Value;


// Outputs
wire [19:0] Out_RED_Filtered;

// Instantiate the Unit Under Test (UUT)
FIR_Filter_RED dut_red (
.CLK_Filter(CLK_Filter),
.rst_n(rst_n),
.ADC_Value(RED_ADC_Value),
.Out_RED_Filtered(Out_RED_Filtered)
);

FIR_Filter_IR dut_ir (
.CLK_Filter(CLK_Filter),
.rst_n(rst_n),
.ADC_Value(IR_ADC_Value),
.Out_IR_Filtered(Out_IR_Filtered)
);


initial begin
// Initialize Inputs


CLK_Filter = 0;
```

```
rst_n = 0;
RED_ADC_Value = 0;
#1000;

rst_n = 1;
#800;

rst_n = 0;
RED_ADC_Value = 8'd5;
#1000;
RED_ADC_Value = 8'd10;
#1000;
RED_ADC_Value  = 8'd12;
#1000;
RED_ADC_Value  = 8'd15;
#1000;
RED_ADC_Value = 8'd16;
#1000;

#100000 $stop;


end
always begin #500 CLK_Filter=~CLK_Filter; end
endmodule
```