HDL Lab Report

Group 03
Course Semester: SoSe 2023
Supervisor: Viktor Weinelt
Department: Integrierte Elektronische Systeme (IES) | Prof. Dr.-Ing. Klaus Hofmann

HDL Lab Report by
Xue Wenxuan,2500838
Jingang Zhang, 2705413
Abdenassir El Amin, 2773160

Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Electrical Engineering and Information Technology Department

Contents

1 Introduction

The process to accelerate digital designs in Integrated Circuits (ICs) heavily relies on high-level Hardware Description Language (HDL) such as Verilog or VHSIC(Very High-Speed Integrated Circuits) Hardware Description Language (VHDL). In this lab, we used HDL step by step to realize the digital design part of a pulse oximetry method. This involved creating three main components: an Analog-to-Digital Converter (ADC), a controller, and a Finite Impulse Response-Filter (FIR-Filter).

To begin, we used Verilog-A to build an ADC capable of converting analog signals into 8-bit digital signals. Subsequently, a testbench was created in Cadence Virtuoso[1] to verify its operational accuracy.

Following this, Verilog code was written to separately implement the controller and FIR-Filter. The controller's function involves adjusting the values of DC Compensation and increasing the Programmable-gain amplifier (PGA) Gain to control the inputs of the analog frontend. It produces two separate 8-bit streams for the red LED and the infrared LED outputs. The role of the FIR Filter is to remove high-frequency noise present in the system. For both of these components, testbenches were written using ModelSim[2] to ensure logical correctness. Further validation and algorithm optimization were performed by creating and utilizing testbenches in Cadence Virtuoso to observe waveform signals. After optimizing the algorithm, we enabled the controller to determine appropriate DC Compensation and PGA Gain values for the LEDs within 3 seconds.

Following that, Synopsis[3] was used to validate and modify the code for both components until they were successfully synthesized.

The synthesized code was then subjected to functional logic verification using a testbench in Cadence Virtuoso. A Top Module was developed to instantiate and interconnect the controller and two FIR-Filters, resulting in an integrated block. This block was likewise synthesized and verified.

However, merely simulating the block for ideal timing isn't enough to guarantee the correct operation of the gate-level circuit. Hence, Standard Delay Format (SDF) was employed to further simulate the block under realistic timing conditions. Finally, Cadence Innovus and related scripts were utilized to position all input and output pins and optimize the size of chip area. This resulted in the final design layout of the block.

This report will treat each component as a separate chapter, and within each chapter we will discuss its functions, design, simulations, verifications, and various aspects.

2 Implementation of ADC

2.1 Discussion and design

As referenced from the CAD4SOC Lab Exercise 2 manual, the ideal ADC converter consists of two important components: the Sample&Hold circuit and the Quantizer. The former is used to sample the the analog input signal every millisecond, while the latter converts the continuous analog signal values sampled and held in the holding circuit into discrete digital values and outputs them. The sampling frequency for this ADC converter design is $1kHz$, and its resolution is 8 bits. The following sections will discuss and show the code implementation and observed waveform results for these two components.

2.2 Implementation of Sample & Hold

According to the working principle of the Sample & Hold circuit, we need to capture the current input analog signal value on every rising edge of the clock signal and store it. This stored value is then passed to the Quantizer until the next sampling event occurs. The specific code implementation is shown in Appendix.

2.3 Implementation of Quantizer

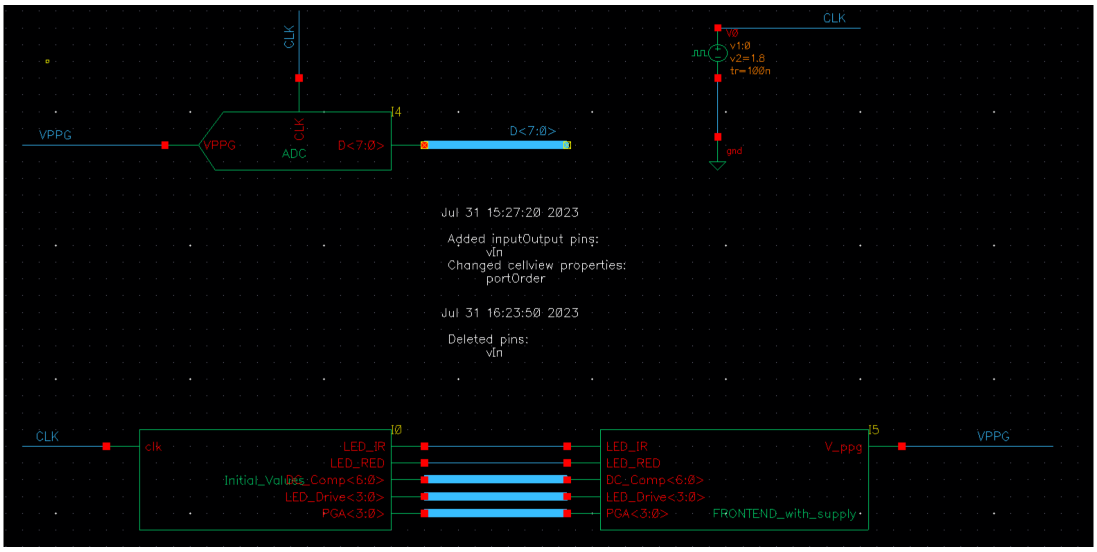
Based on the design requirements to quantize continuous analog input signals into an 8-bit digital output signal, it is determined that the number of quantization levels required is $2^8 = 256$. The output value $result$ is obtained using the formula 2.1:

$$result = 256 \times \frac{V_{in} - v_{RefL}}{v_{RefH} - v_{RefL}} + 0.5 \tag{2.1}$$

where v_{RefL} represents the lower reference voltage of $0V$ and v_{RefH} is the upper reference voltage of $1.8V$. Addition of 0.5 ensures that the result is rounded to the nearest quantization level. Additionally, to prevent overflow and underflow, the result is bounded within the range of 0 to 255. Then, the integer result does AND operation with each bit. If the result is 1, it means that the corresponding output bit is set to a high voltage of 1.8V (logic 1), and vice versa. The code is shown in Appendix. A significant portion of our code was based on the code discussed in the CAD4SOC Lab Exercise 2.

2.4 Testbench and Waveform

In the process of creating a testbench for the ADC, we utilize the analog output signal `Vppg` generated by `Frontend_TB` as the input to the ADC. By observing the waveforms of `Vppg` and the generated 8-bit digital output signals `D<7:0>` in below figure 2.2, we can confidently determine that the ADC is functioning properly.



```
Jul 31 15:27:20 2023

 Added inputOutput pins:
      vIn
 Changed cellview properties:
      portOrder

Jul 31 16:23:50 2023

 Deleted pins:
      vIn
```

Figure 2.1: Testbench for ADC

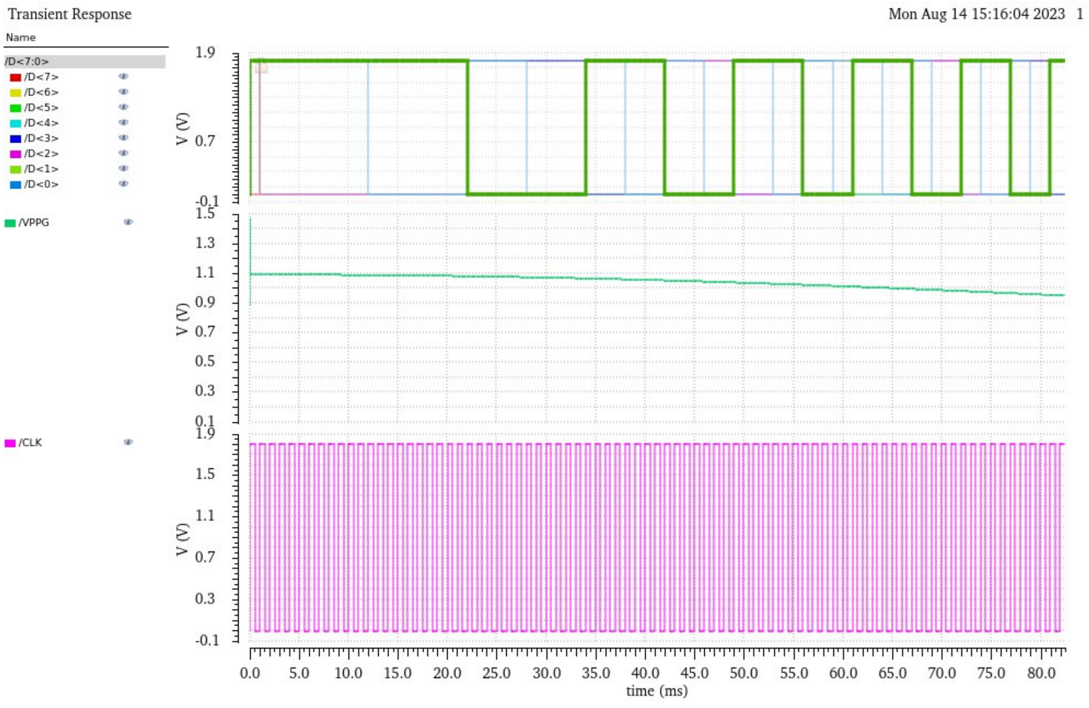


Figure 2.2: Waveform of input and outputs in ADC

3 Implementation of Controller

3.1 Discussion and design

The function of this controller is to find the corresponding DC compensation and maximum PGA gain for the red LED and infrared LED as soon as possible and accurately, so that in the case of these two LEDs being turned on respectively, the average value of the analog signal `Vppg` with PGA gain of 0 will be close to 0.9V and with this maximum PGA gain the signal will not be chipped. Then turn on both LEDs alternately for $10ms$ when the corresponding settings are found for the red LED and the infrared LED respectively. At the same time, an 8-bit digital signal is output for the red LED and the infrared LED respectively.
The following sections discusses the algorithm used to implement the controller, the use of state machines, and the waveform simulated by testbench.

3.2 Binary search algorithm

3.2.1 Principle and Prerequisites

Binary search is an efficient algorithm to find a valid `DC_Comp`,adhering to a logarithmic time complexity of $O(\log(n))$. To use this algorithm in this implementation of controller we must first check the monotony between the `DC_Comp` and the value of `Vppg`. Because binary search can only be used to deal with sorted list of items, which means the need of a continuous monotone function. By adjusting the value of `DC_Comp` in `Initial_Values.v` and examining the waveform of `Vppg` in `Frontend_TB`, it can be observed that for both LEDs, as `DC_Comp` is increased, the value of `Vppg` exhibits a monotonic decrease. The results of checking are shown in the following table 3.1.

Table 3.1: Monotony between `DC_Comp` and `Vppg`

LED	DC_Comp	Waveform of `Vppg`
red/infrared	0	line near 1.75v
infrared	64	wave near 0.9v
red	104	wave near 0.9v
red/infrared	127	line near 0v

3.2.2 First binary search

In the beginning of our program we will do a rough binary search of DC_Comp using the first sample value of Vppg. This step is to make sure that there is an obvious wave in order to help us to find the index of Vppg_max and Vppg_min in one cycle.

3.2.3 Index of Vppg_max and Vppg_min

After performing 1000 samples on the waveform of Vppg over one complete cycle, we obtained the index values corresponding to Vppg_max and Vppg_min within that cycle. Initially, we used the method of summing and averaging 1000 samples to determine if Vppg is close to 0.9 V. However, this method is slow (1000 samples per cycle). Since the waveform of Vppg is similar to sinus signal, we later used to sample only 1 cycle to get the index value of Vppg_max and the index value of Vppg_min. With these two index values, we can get the wave's Vppg_max, Vppg_min, and Vppg_median for each subsequent cycle. By selecting Vppg_median, we can determine whether Vppg is close to 0.9v or not.

3.2.4 Second binary search

After calculating the index of Vppg_median, we will sample the Vppg value around this index 15 times, and compare it with 0.9 V. Finally we can get a relatively accurate valid DC_Comp value by binary search again.

3.3 Implementation of state machine

3.3.1 INITIAL

Upon reception of an asynchronous reset, the system will initiate by entering the INITIAL state, where it will appropriately assign initial values to each register. In this we will assign the nextState to FIND_SETTING as well as almost assigning every register the value 0. This is important because it allows us to have a logic understanding of what our current values are and in case of errors, which happened a lot during synthesizing, we could debug it.

3.3.2 FIND_SETTING

Upon receiving the Find_Setting signal, we will start go into IDLE state and initiate the process to find the DC_Comp and PGA_Gain settings for both LEDs.

3.3.3 IDLE

This state was implemented to set the initial settings for the red LED and infrared LED before finding the correct settings for DC_Comp and PGA_Gain. We assign the register DC_left the value 0 and the register DC_right the value of our parameter MITTEL_VOLTAGE, which happens to be 127, so we would have these values before we would enter our first binary search in order to find our DC compensation(DC_Comp) value.

3.3.4 FIRST_DC_COMP

Upon finishing the initial settings for the red LED and infrared LED, we did the first DC compensation sweep. The aim of this first initial DC compensation sweep was to generate a wave of `Vppg` instead of a line so we could extract the index of `Vppg_max` and `Vppg_min`. For the generation of a wave we use binary search algorithm. How the algorithm works was described in the section `Binary search algorithm`.

3.3.5 SWEEP_DC_COMP

In Sweep DC Comp we use binary search to find the value nearest to 0.9 V or 127 as digital value. In the first step we calculate the median index value as

```
min_index + max_index >>1;
```

and we will at most need 15 clock frequencies to finish the binary search. In the next step we check if the current digital voltage from the Fingerclip is lower or higher than `MITTEL_VOLTAGE`, which is 0.9 V as an analog value or as a digital value 127. If its smaller than 127, then we save the old DC Comp value inside the register `DC_right`, which was found in the previous state `FIRST_DC_Comp` and calculate the new `DC_Comp` value by taking the middle value in between the previous `DC_Comp` and `DC_left` value as.

```
DC_Comp + DC_left >>1;
```

Otherwise, if the value is higher than 0.9 V, then we would save the old `DC_Comp` value in the register `DC_left` and calculate the new `DC_Comp` as.

```
DC_Comp + DC_right >>1;
```

After 15 clock frequencies or if `Vppg` is equal to 127, we have found an optimal value for the DC compensation and we assign the register `nextState` the state `INCREASE_PGA_GAIN` and shift into the next state `INCREASE_PGA_GAIN`.

3.3.6 INCREASE PGA GAIN

Once we have found our DC comp settings we start to adjust the PGA Gain. The idea on how we approach to increase the PGA Gain is by finding the signals index for its maximum value and increase the PGA Gain every clock cycle, as the signals amplitude isn't changing that much within 15 clock periods. In the first statement

```
(counter >= max_index) && counter <= (max_index + 15);
```

we check if our counter is higher than the maximum index and equal or less than 15. If that is the case and if `Vppg < MAX_RAND_VOLTAGE`, then we increase the PGA Gain by one step. Otherwise, we already have found our maximum PGA Gain and temporarily save it in `gaintemp_max`

```
gaintemp_max <= PGA_Gain;
```

If the counter reaches 15, then we reset the PGA Gain value for the next PGA Gain calculation and start storing the PGA Gain and DC Comp settings for the red and infrared LED respectively.

```
PGA_Gain <= 4'd0;
start <= 1;
```

Now we check whether the red LED(`LED_RED`) or the infrared LED(`LED_IR`) is on. If the red LED is on, then we would save the DC_Comp value and `gaintemp_max` in registers as our DC comp and PGA gain settings. We would also reset `gaintemp_max` value to 0 and assign nextState to IDLE, as we have to find a DC compensation and PGA gain value for the infrared LED.

```
DC_Comp_RED <= DC_Comp;
PGA_Gain_RED <= gaintemp_max;
nextState <= IDLE;
```

Otherwise if infrared LED is on, we would reset the same settings and save the DC compensation value and PGA gain value in registers aswell as change the nextState to `Oscillate`.

```
DC_Comp_IR <= DC_Comp;
PGA_Gain_IR <= gaintemp_max;
nextState <= OSCILLATE;
```

3.3.7 OSCILLATE

Once we have established both settings, they will be saved in registers and we will start to oscillate between the two LED's with a frequency of 100 Hz. We managed to create the oscillation between the two LED's with a frequency of 100 Hz by having a counter that goes up to 10 which divides the controller frequency of 1 kHz by 10. We'll provide more detailed explanations of how the code operates in appendix.

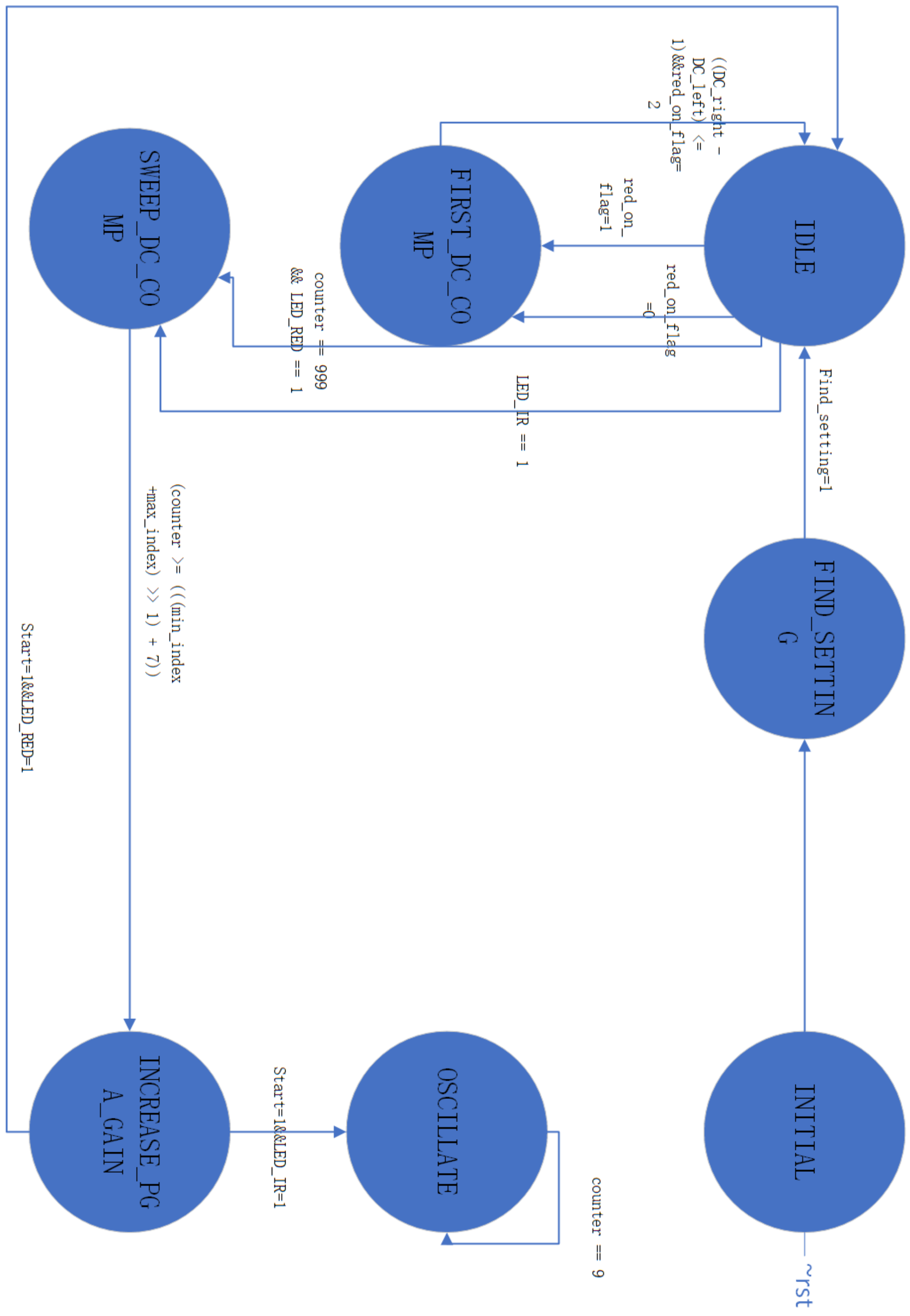
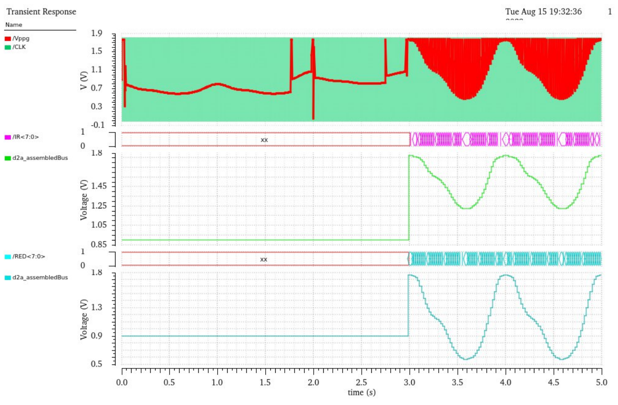
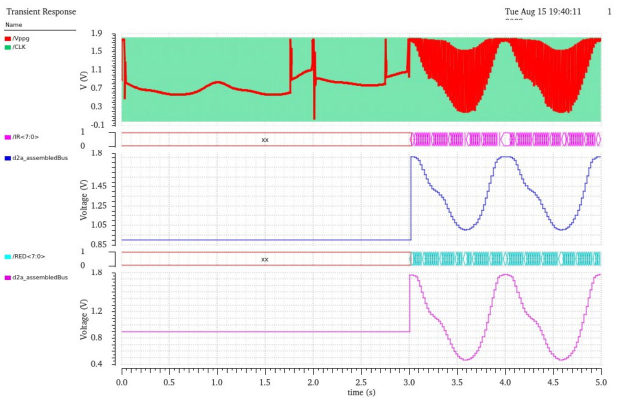


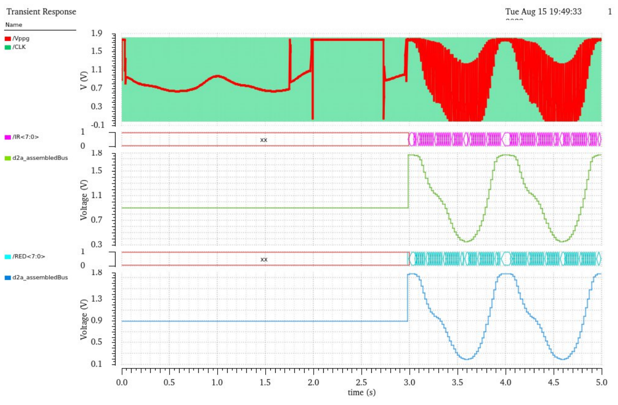
Figure 3.1: State machine of Controller

3.4 Testbench and Waveform

For the controller's testbench, we mainly used the given `Frontend_HDL_Lab_TB` to directly connect our designed ideal ADC and controller to the modules `LED_Drive_Gen` and `FRONTEND_with_supply`. By changing the value of `Drive_Strength` between 8 to 12 in `LED_Drive_Gen`, we get the waveform shown in below Figure 3.2.
By observing the waveforms, we can see that when `Drive_Strength` is 10, the controller we designed can accurately find the `DC_Comp` and `PGA_Gain` for both LEDs within 3s. However, when `Drive_Strength` is changed, even though the value of `DC_Comp` is found correctly, the `Vppg` is chipped due to the inaccurate `PGA_Gain`. Unfortunately, due to time constraints, we weren't able to find the optimization for this problem.


(a) Waveform of Controller with `Drive_Strength`:= 8


(b) Waveform of Controller with `Drive_Strength`:= 10


(c) Waveform of Controller with `Drive_Strength`:= 11

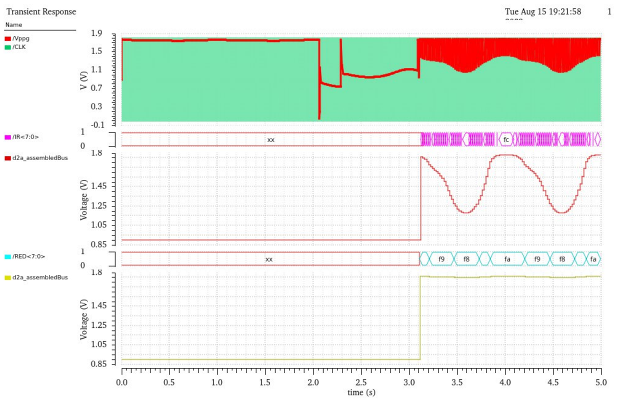

(d) Waveform of Controller with `Drive_Strength`:= 12

Figure 3.2: Different `Drive_Strength` for controller

4 Implementation of FIR-Filter

4.1 Discussion and Design

In order to reduce high-frequency noise in the system, two identical low-pass FIR-Filters with 8-bit digital signal inputs from controllers for red LED and infrared LED have to be implemented. Since FIR-Filter is purely a digital circuit component, it can be developed entirely using Verilog instead of Verilog-A. According to research, the working principle of a FIR-Filter involves multiplying the input digital signal $x[n]$ with a set of coefficients (b_i), and then summing the products to obtain the output signal $y[n]$. The formula 4.1[4] is as follows:

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N] = \sum_{i=0}^{N} b_i \cdot x[n-i] \tag{4.1}$$

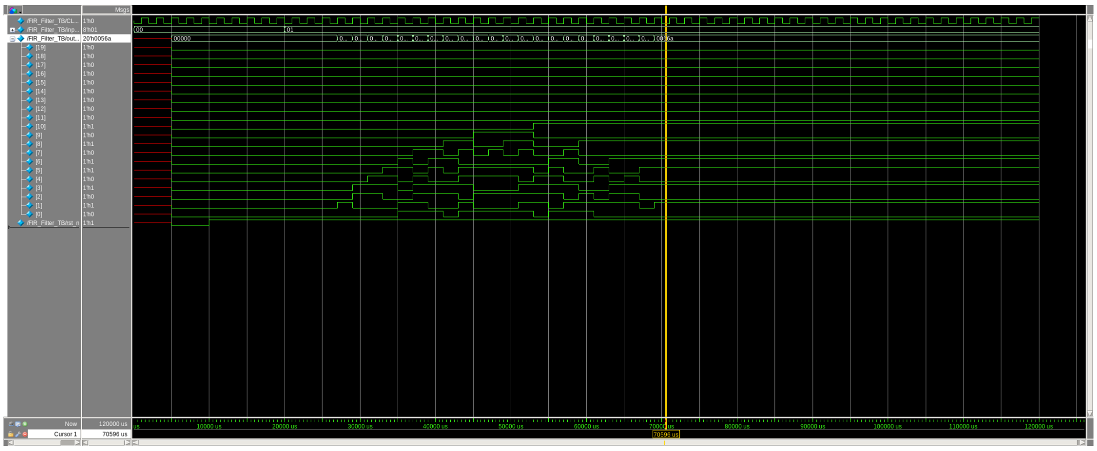
For this design, the FIR-Filter's sampling frequency is $500Hz$, and with the given 22 coefficients, the output results in a 20-bit filtered digital signal output. We provided specific code implementation section in Appendix.

4.2 Implementation of Verilog code

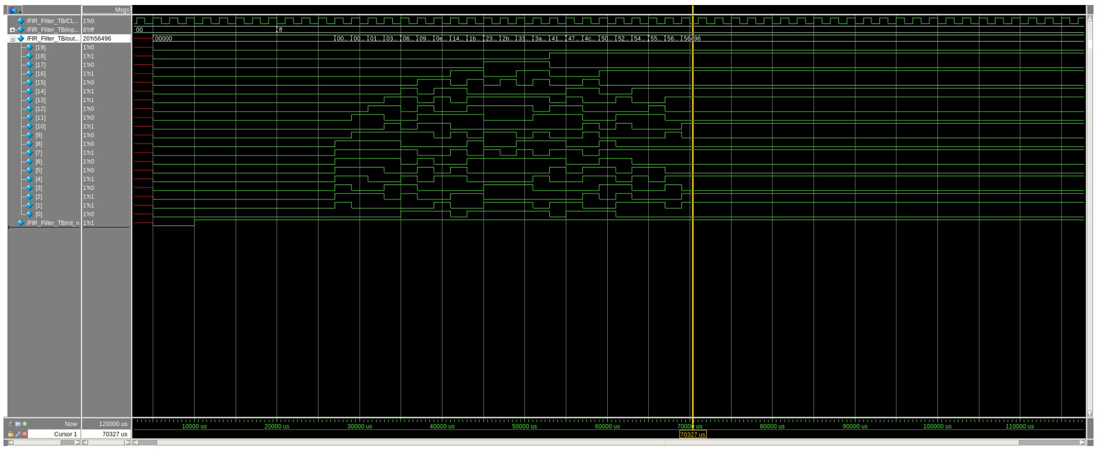
From the equation 4.1, we can see that to build the FIR-Filter, we need 22 shift registers acting as buffer. Each of these can hold an 8-bit input signal. Whenever the clock signal rises, the `input_data<7:0>` gets stored in `shift_regs[0]`, and the prior content of `shift_regs[0]` moves to `shift_regs[1]`, and so forth. Due to the symmetrical nature of the FIR-Filter coefficients, we also need 11 add registers. For instance, `add_regs[0]` holds the sum of values from `shift_regs[0]` and `shift_regs[21]`, while `add_regs[1]` holds the sum of values from `shift_regs[1]` and `shift_regs[20]`, and so on. Afterward, 11 order registers `order_regs` are used to keep the product of multiplying each add register's value by the corresponding coefficients. Finally, each of the products is added together, resulting in the value that becomes the 20-bit output. This algorithm utilizes only 11 multipliers instead of 22, which helps to save chip area probably. In Appendix shows the specific implementation code block.

4.3 Testbench and Waveform

To simulate and verify the correctness of this code logic, we initially created a testbench in ModelSim. By observing the following figure4.1, we noted that when the input was set to 1, the output was the sum of all coefficients (decimal: 1386) or (hexadecimal: $056A$). Similarly, when the input was set to 255, the output was the product of 255 and the sum of coefficients (decimal: 353430) or (hexadecimal: 56496). This observation provides evidence that the logic is correct.



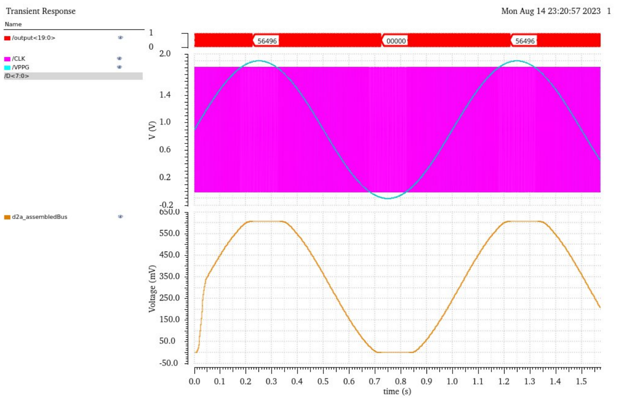
(a) The waveform of the input $==1$



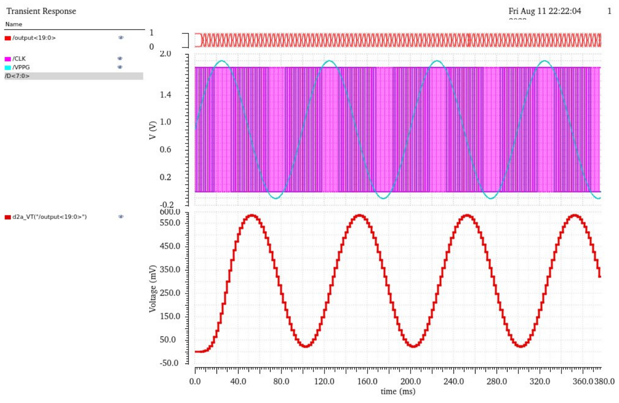
(b) The waveform of the input $==255$

Figure 4.1: Comparison of the waveform between the input := 1 and input := 255

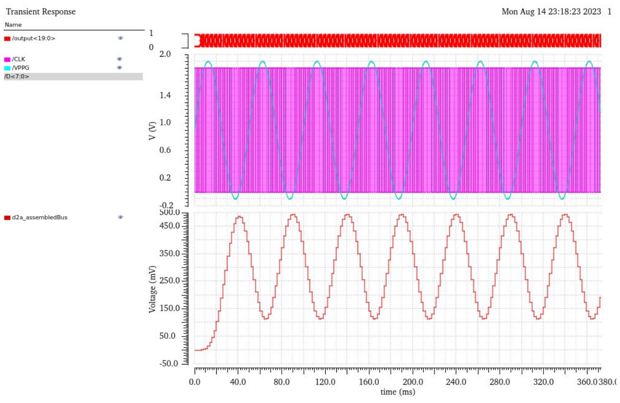
Similarly, a testbench is created in Cadence Virtuoso, and a sinus signal with frequencies of $1Hz$, $10Hz$, $20Hz$, and $100Hz$ and an amplitude of $900mv$ is used as the analog input signal, `Vppg`, which is converted by an ideal ADC to an 8-bit digital signal, `D<7:0>`, as the input signal to the FIR-Filter. As shown in following figure 4.2, as the input signal of the FIR-Filter, the amplitude of the output signal decreases to 0 as the frequency increases, thus realizing the characteristic of passing low frequency and blocking high frequency.



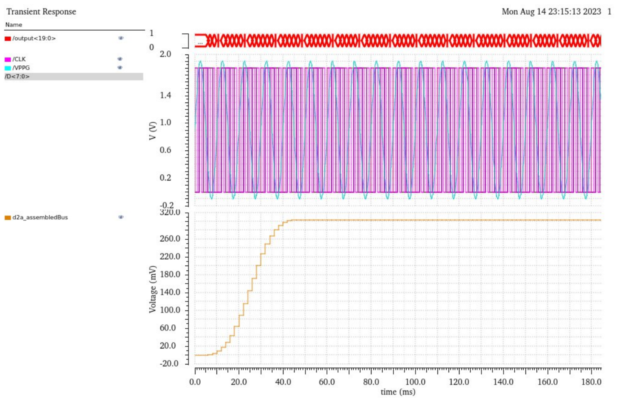
(a) Waveform of input frequency: $1Hz$



(b) Waveform of input frequency: $10Hz$



(c) Waveform of input frequency: $20Hz$



(d) Waveform of input frequency: $100Hz$

Figure 4.2: Different input frequencies of inputs in FIR

5 Top Module of Controller & FIR-Filter

5.1 Discussion and Design

After designing the controller and the FIR-Filter, we need to combine these two sub-modules by Verilog code to form the final top module. This top module should include a controller and two identical FIR-Filters, each filtering the output signals of the red LED and the infrared LED, respectively. The implementation code is shown in the Appendix.

5.2 Testbench and Waveform

After putting the symbol generated from the code of this top module into the given `Frontend_HDL_Lab_TB`, the waveform in the figure 5.1 below is obtained by using the testbench simulation. Observing the waveform, we can see that the top module works normally with a value of 10 for `Drive_Strength`.

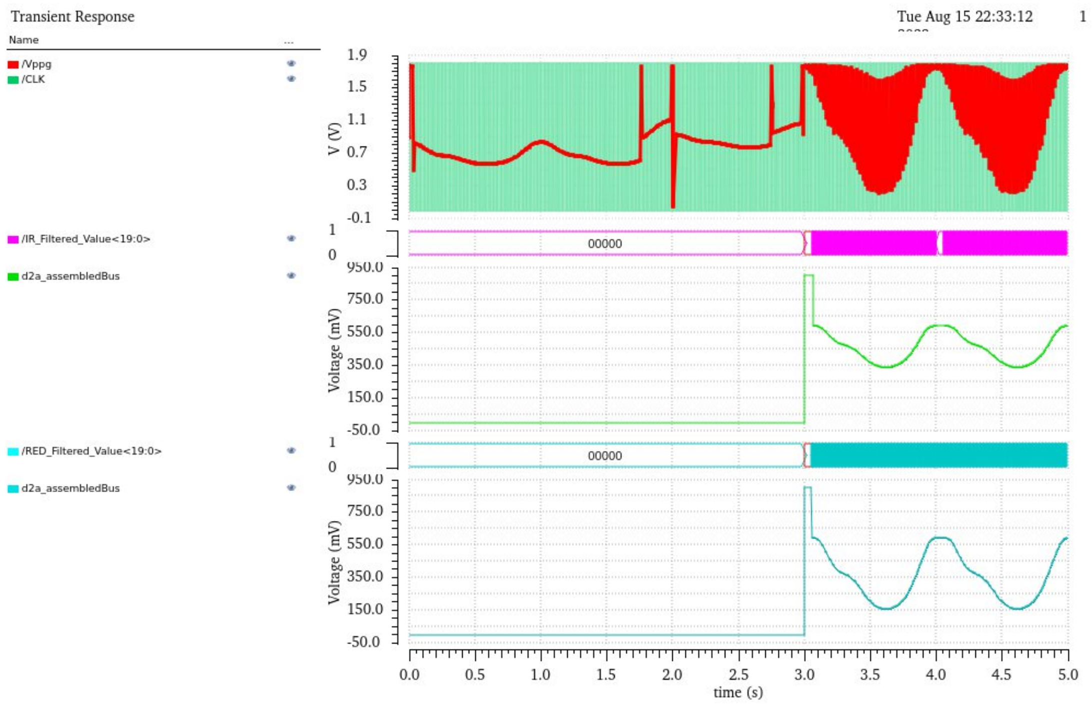


Figure 5.1: Waveform of top module with `Drive_Strength`:= 10

6 Synopsis Design Version and SDF Back Annotation

6.1 Synopsis Design Vision

Synopsis Design Vision was the chosen logic synthesis tool for the conversion of HDL designs into gate-level HDL netlists. The process began with the synthesis of the Filter and Controller components. These components were translated from high-level descriptions into gate-level representations suitable for hardware implementation.
For our Clock, we chose a period of 1 ms. We have faced several problems by synthesis and therefore we adapted our code towards making it synthesized. First it is about initial value. Neither initial command or assigning initial values during declaration can not be synthesised. So we must add an initial state in our state machine. We remove every task and function and write them out in the respective state. Additionally, we removed all blocking assignments and transferred them into non-blocking assignments and encountered many errors when using more than one always block.

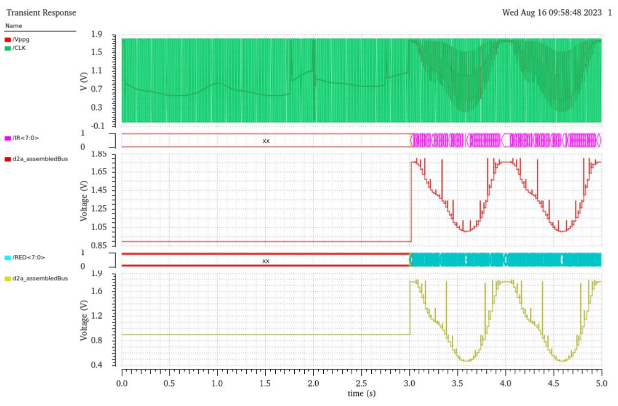
Following successful individual synthesis, the Controller and FIR_Filter were combined into a single Top Module. The outcome of these efforts was summarized in following Table 6.1. This table held results, documenting aspects such as the area of each module, power consumption, and so on.

Table 6.1: The reports of each module

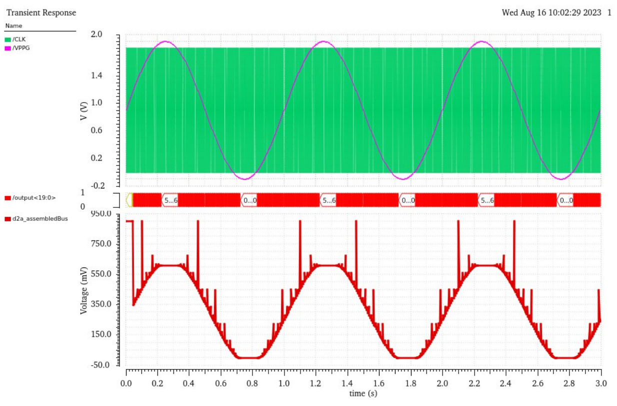
module	Area/μm^2	power consumption/nW	slack time/ns
controller	5487.68	88.35	999987.50
FIR-Filter	11372.87	227.88	999991.56
controller & FIR-Filter	28311.07	538.35	999987.69

6.2 SDF Back Annotation

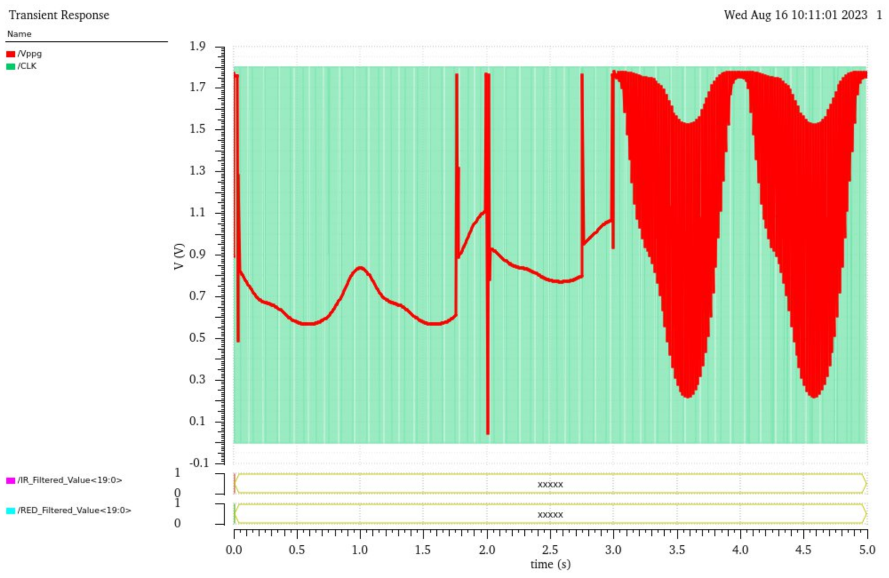
After we had synthesized all of the modules, we proceeded to perform a gate-level simulation using SDF Back Annotation.The SDF Back Annotation involve simulations of the synthesized Filter and Controller. This step ensured accurate performance by considering realistic timing and standards. The waveforms of the AMS Simulation for the controller testbench, FIR-Filter testbench and the top module testbench were displayed in following Figure 6.1. During the AMS Simulation of our controller testbench and FIR-Filter testbench, we observed the intentional presence of noise in the digital output stream. Unfortunately, we did not observe the 20-bit noise-filtered output in the top module, but only the input signal Vppg. We guessed that the FIR-Filter in the top module was not working properly due to not holding realistic time constraints and we were not able to solve this problem unfortunately.



(a) AMS Simulation of controller



(b) AMS Simulation of FIR-Filter



(c) AMS Simulation of the top module

Figure 6.1: Waveform of each module with AMS Simulation

7 Layout of Top Module

7.1 Discussion and Design

First we set all the pins equably on the both sides of chip, which is different with frequency in our design. Second it is common to leave a blank space in the floorplan where no standard cell exists. it is not possible to have every standard cell close to each other in the floorplan, as this would lead to routing congestion problems and would not be conducive to timing. So we adjust the size ratio,which we choose 0.7, because in out design the value higher than it will cause shortcuts. This is to minimize the chip area while ensuring that the circuit works correctly.

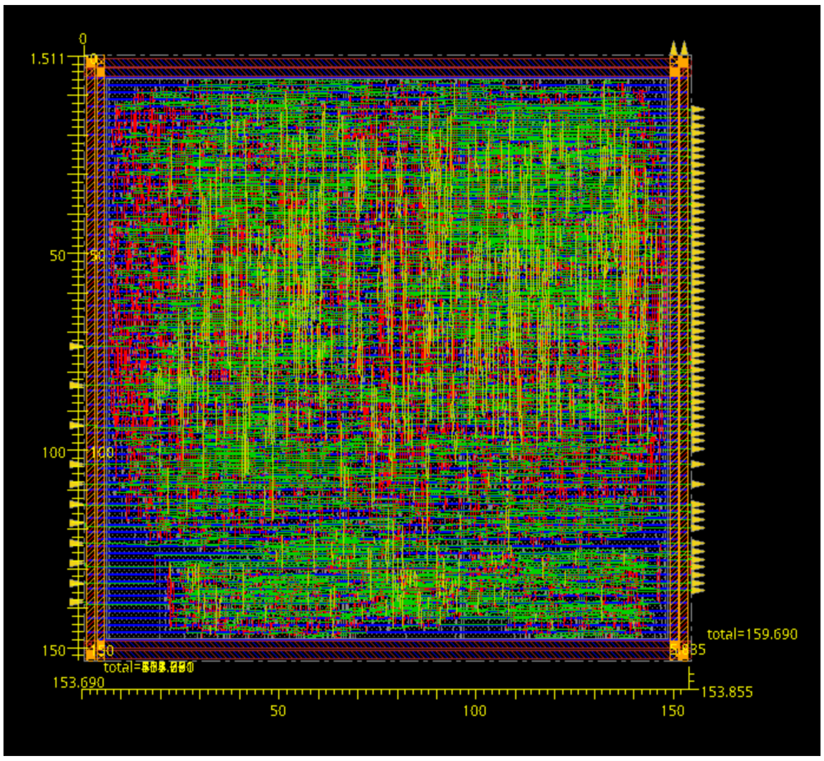


Figure 7.1: Layout of top module with clock frequency 1kHz

8 Summary

8.1 Final Result

After we finished all necessary tests, we collect important information of parameters of our design, that shows the quality of our design.

Table 8.1: Achieved results

Frequency	1 kHz	100 MHz	Max.Frequency(1 GHz)
time to find valid setting in controller	3 s	X	X
Area Synthesis	28311.07 μm^2	28292.58 μm^2	34187.89μm^2
Area Layout	23685.2 μm^2	23715.960 μm^2	29308 μm^2
Power consumption in Synthesis	538.35 nW	98988 nW	1.4744 mW
Power consumption in Layout	474.5 nW	501.7 nW	3853 nW

As the result shows, we achieve a very fast find-setting time, but our design consumes a little bit larger area and power consumption.

Table 8.2: Filename table

Exercise	Path and Modulename	Path and Testbenchname
Verilog-A ADC	HDL_LAB_Gruppe03/ADC	HDL_LAB_Gruppe03/ADC_TB
Verilog Controller	src/Controller.v/Controller	HDL_LAB_Gruppe03/Controller_TB
Verilog Filter	src/FIR_Filter.v/FIR_Filter	HDL_LAB_Gruppe03/FIR_Filter_TB
Verilog Controller&FIR	src/Controller_FIR.v/Controller_FIR	HDL_LAB_Gruppe03/Controller_FIR_TB

8.2 The issues we resolved

8.2.1 Sample frequency

In the beginning of the creation of the ADC, we got no digital signal at all. There are lots of high frequency part in the output. Then we realize, that the Nyquist Sampling Theorem is violated, where the sampling frequency is more than twice the highest frequency of the signal in order for the signal to be recovered without distortion.

8.2.2 Blocking vs Non-Blocking

In ModelSim we use blocking assignment in the second part of state machine as the script shows. But during the simulation in Cadence we found, that blocking assignment can not give a correct output because of timing sequence. So we decided to change all assignments to non-blocking. Meanwhile we have to change the conditional statements to original value minus one because every value will always be assigned after the end of a block.

8.3 Outlook and Challenges

During our first SDF Back Annotation we separately do the simulation for controller and the filter, which works well. But when we combine them together, the find-setting part still works properly but the oscillation part appeared to have lots of cut-off. We will discuss the possible reasons in following section.

8.3.1 Frequency problem

Due to the nature of FIR Filters, the input signal at 100 Hz will be converted into max value. We think the reason of that problem might be, that when red and infrared oscillate with 100Hz, the frequency of input signal increases and is eliminated by FIR.

8.3.2 Burr phenomena

Second when signals in digital circuits go through the logic unit connection, there must be a delay. In addition, the signal of the high to low level conversion also requires a certain transition time, due to the existence of these two factors, the level of multi-channel signal value changes, in the instant of the signal changes, the output of the combined logic has a sequential order, not the same time changes, often appear some incorrect spike signal, which are competing adventures, that will generate burrs.

8.3.3 DC_Drive issues

The controller we designed is fast and correctly executes when DC_Drive is 10, but once its value is changed to 8 or 12, `Vppg` is chipped. This means we have to optimize the code of INCREASE_PGA_Gain state, towards making it not leading to clipping once the DC_Drive value is changed to 8 or 12.

9 Appendix

9.1 ADC converter

```
// VerilogA for HDL_LAB_Gruppe03, SimpleHold4ADC, veriloga

/**
This module is a sample and hold block for ADC
**/
'include "constants.vams"
'include "disciplines.vams"

module SimpleHold4ADC(AGND, clk, vIn, vOut);

    // four inout ports: AGND, clk, vIn, and vOut.
    inout AGND;
    electrical AGND; // declared as electrical, indicating electrical nature.
    inout clk;
    electrical clk;
    inout vIn;
    electrical vIn;
    inout vOut;
    electrical vOut;

    // the clock threshold voltage is 0.9 V
    parameter real clkThreshold = 0.9 from [0:inf];
    real vOutTemp;

    analog begin
        // waits for a rising edge of the clock (clk) that crosses the clock threshold voltage
        @(cross(V(clk,AGND) - clkThreshold,1))begin
            vOutTemp = V(vIn,AGND); // rising edge is detected, the input voltage vIn stored
    in vOutTemp.
        end

        V(vOut,AGND) <+ vOutTemp; // vOutTemp is assigned to the output voltage vOut

    end

endmodule
```

Verilog-A code block of Sample & Hold circuit

```
// VerilogA for HDL_LAB_Gruppe03, Quantizer4ADC, veriloga
/**
This module is quantizer for analog-to-digital converter (ADC)
**/
```

```
'include "constants.vams"
'include "disciplines.vams"

module Quantizer4ADC(AGND, D, vIn);

    // three inout ports: AGND, D, and vIn.
    inout AGND;
    electrical AGND; //declared as electrical, indicating electrical nature
    inout[7:0] D;
    electrical[7:0] D;
    inout vIn;
    electrical vIn;

    // end of automatically generated header
    // the logic levels are 0 to 1.8. V
    parameter real vLow = 0 from [0:inf];
    parameter real vHigh = 1.8 from [0:inf];

    // the reference voltage is 1.8. V
    parameter real vRefL = 0 from [0:inf];
    parameter real vRefH = 1.8 from [0:inf];

    // the number of quantization levels
    localparam integer levels = 256;

    integer result;

analog begin
    // the output value is calculated arithmetically
    result = levels*(V(vIn,AGND) - vRefL)/(vRefH - vRefL) + 0.5;

    // overflow and underflow are caught here
    if (result > levels-1)
        result = levels-1;
    else if (result < 0)
        result = 0;

    // the integer result is converted into bits
    V(D[0]) <+ (result & 1)   ? vHigh : vLow;
    V(D[1]) <+ (result & 2)   ? vHigh : vLow;
    V(D[2]) <+ (result & 4)   ? vHigh : vLow;
    V(D[3]) <+ (result & 8)   ? vHigh : vLow;
    V(D[4]) <+ (result & 16)  ? vHigh : vLow;
    V(D[5]) <+ (result & 32)  ? vHigh : vLow;
    V(D[6]) <+ (result & 64)  ? vHigh : vLow;
    V(D[7]) <+ (result & 128) ? vHigh : vLow;
end

// end of module
endmodule
```

Verilog-A code block of Quantizer

9.2 Controller

```verilog
INITIAL: begin
    nextState           <= FIND_SETTING;
    counter             <= 4'd0;
    PGA_Gain_RED        <= 0;
    PGA_Gain_IR         <= 0;
    DC_Comp_temp        <= 0;
    DC_Comp_IR          <= 0;
    DC_Comp_RED         <= 0;
    red_on_flag         <= 1;
    temp_min            <= 255;
    temp_max            <= 0;
    min_index           <= 0;
    max_index           <= 0;
    LED_IR              <= 0;
    LED_RED             <= 1;
    PGA_Gain            <= 4'd0;
    gaintemp            <= 15;
    LED_Drive           <= 8;
    CLK_Filter          <= 0;
    gaintemp_max        <= 0;
    gaintemp_min        <= 0;
    start <= 0;
end
```

Verilog code block of INITIAL state

```verilog
FIND_SETTING: begin
    if(Find_Setting) begin
        nextState <= IDLE;
    end
end
```

Verilog code block of FIND SETTING state

```verilog
IDLE: begin
    // reset
    DC_left   <= 0;
    DC_right  <= MITTEL_VOLTAGE;
    case(red_on_flag)  //Here we check whether we have do DC_Sweep with RED LED or IR LED.
    0: begin  //Here we do 1 LED_IR DC_Sweep in order to get a waveform so we can determine
    min and max index of a wave.
        nextState <= FIRST_DC_COMP;
        LED_IR    <= 1;
        LED_RED   <= 0;
        DC_Comp   <= 0;
        temp_min  <= 255;
        temp_max  <= 0;
    end
    1: begin  //Here we do 1  DC_Sweep for LED_RED in order to get a waveform so we can
    determine min and max index of a wave.
        nextState <= FIRST_DC_COMP;
        LED_IR    <= 0;
        LED_RED   <= 1;
        DC_Comp   <= 0;
        temp_min  <= 255;
        temp_max  <= 0;
    end
    2: begin  //In previous states we have calculated min and max index and here we do multiple DC_Comp changes around the median value until it matches 0.9 Voltage.

        if (Vppg < temp_min && LED_RED == 1) begin
            temp_min  <= Vppg;
            min_index <= counter; //index_counter
            nextState <= IDLE;
        end
        if (Vppg > temp_max && LED_RED == 1) begin
            temp_max  <= Vppg;
            max_index <= counter; //index_counter
            nextState <= IDLE;
        end

        if (counter == 999 && LED_RED == 1)begin //index_counter
            nextState <= SWEEP_DC_COMP; // set led red on
        end
        if(LED_IR == 1) begin
            nextState <= SWEEP_DC_COMP;
        end
    end

    default:;
    endcase
end
```

Verilog code block of IDLE state

```verilog
FIRST_DC_COMP:begin
    if ((DC_right - DC_left) > 1)begin
        if(Vppg < MITTEL_VOLTAGE)begin
            DC_right        <=  DC_Comp;
            DC_Comp_temp    <=  DC_Comp + DC_left;
            DC_Comp         <=  DC_Comp_temp >>1;
        end else if(Vppg > MITTEL_VOLTAGE)begin
            DC_left         <=  DC_Comp;
            DC_Comp_temp    <=  DC_Comp + DC_right;
            DC_Comp         <=  DC_Comp_temp >> 1;
        end else begin
            DC_left         <=  DC_Comp;
            DC_right        <=  DC_Comp;
        end
    end else begin
        red_on_flag         <=   2;
        nextState           <=  IDLE;
    end
end
```

Verilog code block of FIRST DC COMP state

```verilog
SWEEP_DC_COMP: begin
    if (counter < (((min_index +max_index) >> 1) + 7) && counter >= (((min_index + max_index) >> 1) - 7))begin
        if(Vppg < MITTEL_VOLTAGE)begin
            DC_right        <=  DC_Comp;
            DC_Comp_temp    <=  DC_Comp + DC_left;
            DC_Comp         <=  DC_Comp_temp >>1;

        end else if(Vppg > MITTEL_VOLTAGE)begin
            DC_left          <=  DC_Comp;
            DC_Comp_temp     <=  DC_Comp + DC_right;
            DC_Comp          <=  DC_Comp_temp >> 1;

        end else begin
            DC_left          <=  DC_Comp;
            DC_right         <=  DC_Comp;
            nextState        <= INCREASE_PGA_GAIN;
        end
    end else if (counter >= (((min_index +max_index) >> 1) + 7)) begin
        nextState        <= INCREASE_PGA_GAIN;
    end
end
```

Verilog code block of SWEEP DC COMP state

```verilog
INCREASE_PGA_GAIN: begin
    if((counter >= max_index) && counter <= (max_index + 4'd15)) begin
        if(Vppg < MAX_RAND_VOLTAGE) begin
            PGA_Gain <= PGA_Gain + 4'd1;
        end
        else begin
            gaintemp_max <= PGA_Gain;  //Once it would start to clip the PGA_Gain will be saved in a register.
        end
        if(counter == (max_index + 4'd15)) begin  //Here we make sure to store the biggest PGA_Gain value and reset the PGA_Gain value once we go through all 15 PGA_Gains
            PGA_Gain <= 4'd0;
            start <= 1;
        end
    end
    if(start) begin
        start <= 0;
        if(2000 < gaintemp_max) begin
            red_on_flag    <= 0 ;
            counter        <= 0; //reset

            if(LED_IR) begin                    // store actual DC_Comp setting of LED IR in register DC_Comp_IR
                DC_Comp_IR    <= DC_Comp;       //store the PGA_Gain setting of LED IR in register PGA_Gain_IR
                PGA_Gain_IR   <= gaintemp_min;  //start switching between LED RED UND LED IR with a frequency of 100 Hz
                nextState     <= OSCILLATE;
                PGA_Gain      <= 4'd0; //reset
                gaintemp      <= 15;
                gaintemp_max  <= 0;
                gaintemp_min  <= 0;
            end

            if(LED_RED) begin
                DC_Comp_RED   <= DC_Comp;       // store actual DC_Comp setting of LED RED in register DC_Comp_RED
                PGA_Gain_RED  <= gaintemp_min;  //store the PGA_Gain setting of LED RED in register PGA_Gain_RED
                nextState     <= IDLE;          //We have to start with LED_RED and switch to LED IR
                PGA_Gain      <= 4'd0;          //reset of PGA_Gain
                gaintemp      <= 15;
                gaintemp_max  <= 0;
                gaintemp_min  <= 0;
                LED_RED       <= 0;
                LED_IR        <= 1;
            end

        end else begin
            red_on_flag       <= 0 ;
            counter           <= 0; //reset

            if(LED_IR) begin
                DC_Comp_IR    <= DC_Comp;           // store actual DC_Comp setting of LED IR in register DC_Comp_IR
                PGA_Gain_IR   <= gaintemp_max;      //store the PGA_Gain setting of LED IR in register PGA_Gain_IR
                nextState     <= OSCILLATE;         //start switching between LED RED UND LED IR with a frequency of 100 Hz
                PGA_Gain      <= 4'd0; //reset
                gaintemp      <= 15;
                gaintemp_max  <= 0;
                gaintemp_min  <= 0;
                LED_RED       <= 1;
                LED_IR        <= 0;
            end

            if(LED_RED) begin
                DC_Comp_RED   <= DC_Comp;           // store actual DC_Comp setting of LED RED in register DC_Comp_RED
                PGA_Gain_RED  <= gaintemp_max;      //store the PGA_Gain setting of LED RED in register PGA_Gain_RED
                nextState     <= IDLE;              //We have to start with LED_RED and switch to LED IR
                PGA_Gain      <= 4'd0;              //reset of PGA_Gain
                gaintemp      <= 15;
                gaintemp_max  <= 0;
                gaintemp_min  <= 0;
                LED_RED       <= 0;
                LED_IR        <= 1;
            end
        end
    end
end
```

Verilog code block of INCREASE PGA GAIN state

```
OSCILLATE: begin
    CLK_Filter <=~CLK_Filter; //Create Clock for FIR_Fitler
    if(counter == 9) begin   //We count up to 10 to create periods of 10 ms
        LED_RED <= ~LED_RED;
        LED_IR  <= ~LED_IR;
        counter <= 0; //reset

        // set dc and pga parameter for next LED RED
        if(LED_IR) begin
            DC_Comp       <= DC_Comp_RED;
            PGA_Gain      <= PGA_Gain_RED;
            IR_ADC_Value  <= Vppg; //output
        end

        // set dc and pga parameter for next LED IR
        if(LED_RED) begin
            DC_Comp       <= DC_Comp_IR;
            PGA_Gain      <= PGA_Gain_IR;
            RED_ADC_Value <= Vppg; //output
        end

    end
end
```

Verilog code block of OSCILLATE state

9.3 FIR-Filter

```
/**
This module is FIR Filter, that can remove high frequency noise in system
**/
module FIR_Filter(
    input wire CLK_Filter,          // Clock for the filter
    input wire [7:0] input_data,    // Input data from ADC (0 to 255)
    input wire rst_n,               // Reset signal (active-low)
    output reg [19:0] output_data   // Filtered output (20-bit)
);

parameter NUM_SHI_REGS = 22;    // Number of shift registers that as buffer
parameter NUM_ADD_REGS = 11;    // Number of add registers

reg     [7:0]   shift_regs   [NUM_SHI_REGS-1:0];  // 22 shift registers as buffer to store
    input data
reg     [8:0]   add_regs     [NUM_ADD_REGS-1:0];  // to store the summed data of first and
    last shift register
// Each order multiplies with the according coefficient
reg     [16:0]  order[NUM_ADD_REGS-1:0];
wire    [7:0]   COEFFICIENTS [NUM_ADD_REGS-1:0];  // to store FIR Filter coefficients value

// 11 FIR filter coefficients
// Only 11 multipliers needed because of the symmetry of FIR filter coefficients
assign COEFFICIENTS[0]  =    8'd2;
assign COEFFICIENTS[1]  =    8'd10;
assign COEFFICIENTS[2]  =    8'd16;
assign COEFFICIENTS[3]  =    8'd28;
assign COEFFICIENTS[4]  =    8'd43;
assign COEFFICIENTS[5]  =    8'd60;
assign COEFFICIENTS[6]  =    8'd78;
assign COEFFICIENTS[7]  =    8'd95;
assign COEFFICIENTS[8]  =    8'd111;
assign COEFFICIENTS[9]  =    8'd122;
assign COEFFICIENTS[10] =    8'd128;

always @(posedge CLK_Filter or negedge rst_n) begin
    if (!rst_n) begin
        // reset the value in all registers to 0
        shift_regs[0]  <= 8'b0;
        shift_regs[1]  <= 8'b0;
        shift_regs[2]  <= 8'b0;
        shift_regs[3]  <= 8'b0;
        shift_regs[4]  <= 8'b0;
        shift_regs[5]  <= 8'b0;
        shift_regs[6]  <= 8'b0;
        shift_regs[7]  <= 8'b0;
        shift_regs[8]  <= 8'b0;
        shift_regs[9]  <= 8'b0;
        shift_regs[10] <= 8'b0;
        shift_regs[11] <= 8'b0;
        shift_regs[12] <= 8'b0;
        shift_regs[13] <= 8'b0;
        shift_regs[14] <= 8'b0;
        shift_regs[15] <= 8'b0;
        shift_regs[16] <= 8'b0;
        shift_regs[17] <= 8'b0;
        shift_regs[18] <= 8'b0;
        shift_regs[19] <= 8'b0;
        shift_regs[20] <= 8'b0;
        shift_regs[21] <= 8'b0;

        add_regs[0]  <= 9'b0;
        add_regs[1]  <= 9'b0;
        add_regs[2]  <= 9'b0;
        add_regs[3]  <= 9'b0;
        add_regs[4]  <= 9'b0;
        add_regs[5]  <= 9'b0;
        add_regs[6]  <= 9'b0;
        add_regs[7]  <= 9'b0;
        add_regs[8]  <= 9'b0;
        add_regs[9]  <= 9'b0;
        add_regs[10] <= 9'b0;

        order[0]  <= 17'b0;
        order[1]  <= 17'b0;
        order[2]  <= 17'b0;
        order[3]  <= 17'b0;
        order[4]  <= 17'b0;
        order[5]  <= 17'b0;
        order[6]  <= 17'b0;
        order[7]  <= 17'b0;
        order[8]  <= 17'b0;
        order[9]  <= 17'b0;
        order[10] <= 17'b0;

        output_data <= 20'd0;

    end else begin
        // all shift registers to store input data
        shift_regs[0]  <= input_data;
        shift_regs[1]  <= shift_regs[0];
        shift_regs[2]  <= shift_regs[1];
        shift_regs[3]  <= shift_regs[2];
        shift_regs[4]  <= shift_regs[3];
        shift_regs[5]  <= shift_regs[4];
        shift_regs[6]  <= shift_regs[5];
```

```verilog
        shift_regs[7]  <= shift_regs[6];
        shift_regs[8]  <= shift_regs[7];
        shift_regs[9]  <= shift_regs[8];
        shift_regs[10] <= shift_regs[9];
        shift_regs[11] <= shift_regs[10];
        shift_regs[12] <= shift_regs[11];
        shift_regs[13] <= shift_regs[12];
        shift_regs[14] <= shift_regs[13];
        shift_regs[15] <= shift_regs[14];
        shift_regs[16] <= shift_regs[15];
        shift_regs[17] <= shift_regs[16];
        shift_regs[18] <= shift_regs[17];
        shift_regs[19] <= shift_regs[18];
        shift_regs[20] <= shift_regs[19];
        shift_regs[21] <= shift_regs[20];

        // register data are summed first and last
        add_regs[0]     <= shift_regs[0] + shift_regs[21];
        add_regs[1]     <= shift_regs[1] + shift_regs[20];
        add_regs[2]     <= shift_regs[2] + shift_regs[19];
        add_regs[3]     <= shift_regs[3] + shift_regs[18];
        add_regs[4]     <= shift_regs[4] + shift_regs[17];
        add_regs[5]     <= shift_regs[5] + shift_regs[16];
        add_regs[6]     <= shift_regs[6] + shift_regs[15];
        add_regs[7]     <= shift_regs[7] + shift_regs[14];
        add_regs[8]     <= shift_regs[8] + shift_regs[13];
        add_regs[9]     <= shift_regs[9] + shift_regs[12];
        add_regs[10]    <= shift_regs[10] + shift_regs[11];

        // Each order multiplies with the according coefficient
        order[0]    <= COEFFICIENTS[0] * add_regs[0];
        order[1]    <= COEFFICIENTS[1] * add_regs[1];
        order[2]    <= COEFFICIENTS[2] * add_regs[2];
        order[3]    <= COEFFICIENTS[3] * add_regs[3];
        order[4]    <= COEFFICIENTS[4] * add_regs[4];
        order[5]    <= COEFFICIENTS[5] * add_regs[5];
        order[6]    <= COEFFICIENTS[6] * add_regs[6];
        order[7]    <= COEFFICIENTS[7] * add_regs[7];
        order[8]    <= COEFFICIENTS[8] * add_regs[8];
        order[9]    <= COEFFICIENTS[9] * add_regs[9];
        order[10]   <= COEFFICIENTS[10] * add_regs[10];

        // output
        output_data <=   order[0] + order[1] + order[2] + order[3]
                       + order[4] + order[5] + order[6] + order[7]
                       + order[8] + order[9] + order[10];
    end
end
endmodule
```

Verilog code block of FIR-Filter

```verilog
/**
This FIR_Filter Testbench is used to simulate Waveform of outputs of FIR in ModelSim
**/
`timescale 1ms/1us
module FIR_Filter_TB;

```

```verilog
    reg         CLK_Filter;    //The clock signal of Filter
    reg  [7:0]  input_data;
    wire [19:0] output_data;
    reg         rst_n;         //An asychronous positive edge reset signal

    //Instantiate FIR Filter Module
    FIR_Filter DUT (
        .CLK_Filter(CLK_Filter),
        .input_data(input_data),
        .output_data(output_data),
        .rst_n(rst_n)
    );

    initial begin
        CLK_Filter = 0;
        input_data = 0;

        // Reset the filter
        #5 rst_n = 0;
        #5 rst_n = 1;

        // Testbench stimulus
        #10 input_data = 1;     // Test input data

        //#10 input_data = 255; // Test input data
        #100 $stop;             // End simulation after some time
    end

    always begin
        #1 CLK_Filter = ~CLK_Filter; // Toggle clock every 1 time units
    end

endmodule
```

Verilog code block of FIR-Filter testbench

9.4 Top-module of Controller & FIR-Filter

```verilog
/**
This module is the top module of controller and FIR filter
**/
module Controller_FIR (
    input   [7:0]   VPPG,             //Binary output Signal of the ADC
    input           Find_Setting,     //Reset to start find Settings
    input           CLK,              //The clock signal
    input           rst_n,            //An asychronous positive edge reset signal

    output  [6:0]   DC_Comp,          //Used to change the DC_Offset
    output  [3:0]   pga_Gain,         //Used to increase the Gain
    output          LED_IR,           //Controlls Infrared LED
    output          LED_RED,          //Controlls RED LED
    output  [19:0]  Out_RED_Filtered,//The filtered Bit stream output of RED LED
    output  [19:0]  Out_IR_Filtered, //The filtered Bit stream output of RED IR
    output  [3:0]   LED_Drive         //Sets the LED brightness
);
```

```

    // the wires to connect Contoller module and 2 FIR Filters moudule
    wire [7:0] IR_ADC_Value, RED_ADC_Value;
    wire       CLK_Filter;

    //Instantiate Controller_Model
    Controller ct(  .Vppg(VPPG),
                    .Find_Setting(Find_Setting),
                    .CLK(CLK),
                    .rst_n(rst_n),
                    .DC_Comp(DC_Comp),
                    .PGA_Gain(pga_Gain),
                    .CLK_Filter(CLK_Filter),
                    .LED_IR(LED_IR),
                    .LED_RED(LED_RED),
                    .IR_ADC_Value(IR_ADC_Value),
                    .RED_ADC_Value(RED_ADC_Value),
                    .LED_Drive(LED_Drive));

    //Instantiate the FIR_Filter Module for RED_ADC_Value
    FIR_Filter red( .CLK_Filter(CLK_Filter),
                    .input_data(RED_ADC_Value),
                    .rst_n(rst_n),
                    .output_data(Out_RED_Filtered));

    //Instantiate the FIR_Filter Module for IR_ADC_Value
    FIR_Filter ir(  .CLK_Filter(CLK_Filter),
                    .input_data(IR_ADC_Value),
                    .rst_n(rst_n),
                    .output_data(Out_IR_Filtered));

endmodule
```

top_module

List of Abbreviation

ICs	Integrated Circuits
HDL	Hardware Description Language
VHDL	VHSIC(Very High-Speed Integrated Circuits) Hardware Description Language
ADC	Analog-to-Digital Converter
FIR-Filter	Finite Impulse Response-Filter
PGA	Programmable-gain amplifier
SDF	Standard Delay Format

Bibliography

[1] *Cadence Virtuoso*. `https://www.cadence.com/`.

[2] *ModelSim*. `https://eda.sw.siemens.com/en-US/ic/modelsim/`.

[3] *Synopsys*. `https://www.synopsys.com/`.

[4] *FIR-Filter equation*. `https://en.wikipedia.org/wiki/Finite_impulse_response`.

List of Figures

List of Tables