Learning Portfolio by

NAME	Yang Hu
MATRICULATION NUMBER	2085761
EMAIL	udifb@student.kit.edu

_____

Date, Signature

Content

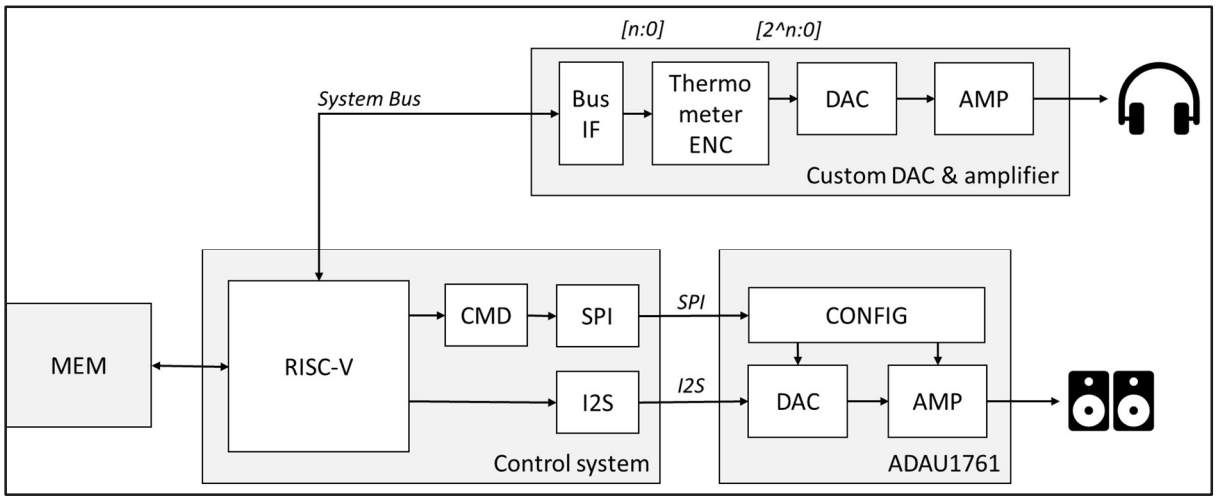
1. Introduction & Motivation

With the abundance of digital audio sources in the modern world, be it streaming music & podcasts or watching videos, there is a particular demand for solutions that are able to process and play the audio files. With the requirement that these audio files need to be replayable on a wide range of end devices in different sizes, like smartphones, Bluetooth headphones or wearables creates the need for highly integrated circuits. These circuits have to include a system that reads data from a memory, a DAC to convert the digital audio data to an analog signal and finally amplify that signal so it can be played on speakers or headphones.
The realization of that task requires it to design all of these modules one by one through the use of digital, analog and mixed signal techniques. Our desired system is supposed to be able to drive an external DAC through I2S protocol as well as a fully integrated internal DAC and amplifier.
For the development of this solution, rapid prototyping of the digital part is done through testing on a FPGA as well as RTL synthesis. The analog and mixed signal part is tested pre-silicon by thorough simulation. That way we can already guarantee a high level of reliability firsthand.
There are several modules that need to be designed for our fully working system. Firstly, a module that handles the configuration of the external DAC, in our case through the SPI protocol, as well as a list of commands that are transferred via SPI. Secondly a module that takes in audio data from the memory and transmits that data to the external DAC using the I2S protocol. And for the use with the internal DAC, an encoder is needed to transform the data into thermometer code to easily drive the DAC stages. These stages need to be designed as well and coupled to an efficient amplifier circuit to be able to drive the output.
Lastly the modules need to be combined and implemented in a full layout, routed, and checked to be able to produce it on silicon.

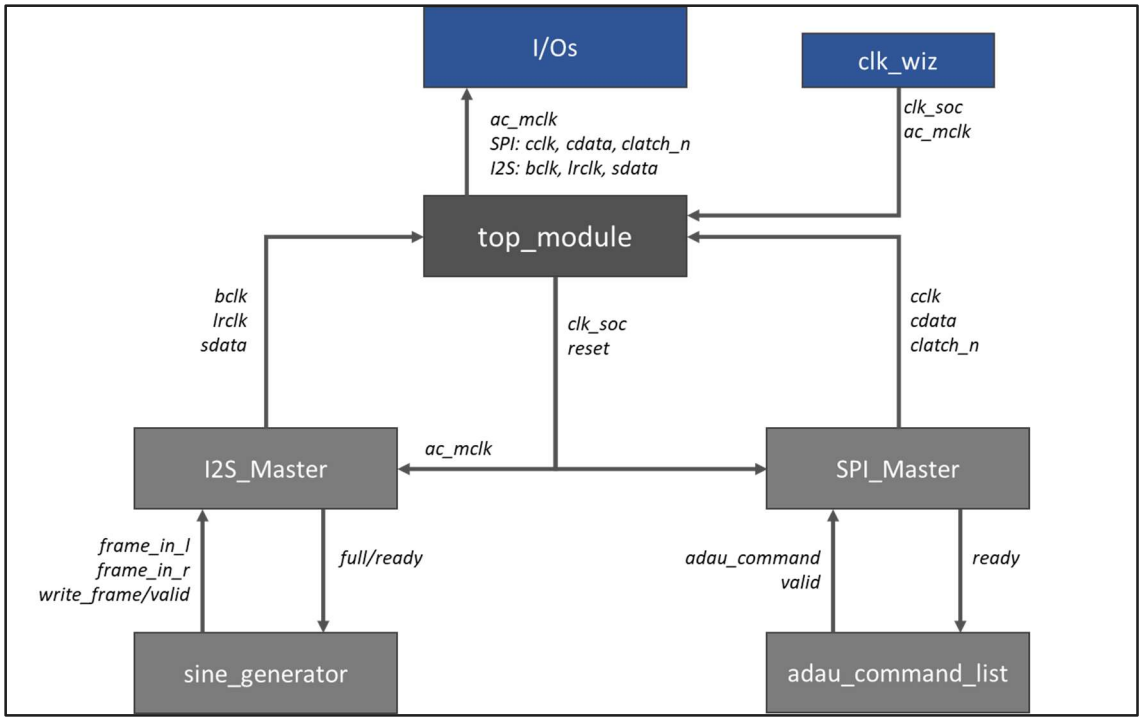
The steps and information necessary to design the systems described above, are specified closely in the following pages. The focus lies on the concrete data, instructions and calculations needed for the tasks. For more general information, scientific sources are recommended. Also, the challenges that we faced in specific tasks are discussed in the respective chapters. In the end there will also be a reflection about common challenges and solutions concerning the whole project.

2. System Overview

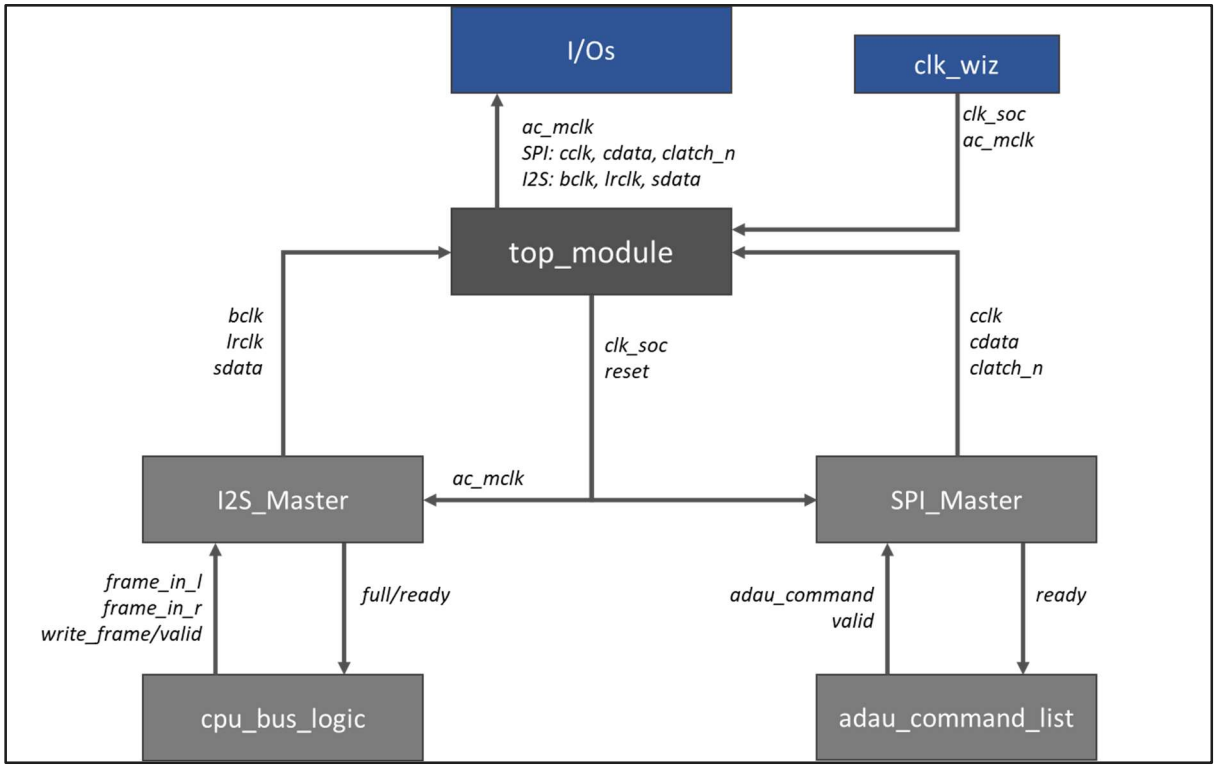
For the final system there are supposed to be two possible outputs for the audio signal. One output is using the external DAC ADAU1761, which is configured through SPI and fed with data through I2S. The other output is realized as an internal custom DAC and amplifier. This output is tightly coupled to the processing unit by the system bus. The audio data is then converted to thermometer code to drive the DAC and amplifier. The complete overview on the system can be seen in the following picture. For the control system a FPGA is used in prototyping but in general every processing unit, which includes the desired bus interfaces and has a high enough clock speed can drive the system.



The prototyping system that runs on the FPGA basically consists of two modules that implement the Master functionality for the SPI (used for the control of the DAC) and the I2S (used for transmitting the data) bus. The controls to be sent to the DAC are configured in a separate module called the “adau_command_list”. For simple testing, the test data is generated through a simple LUT sine generator that is connected to the I2S Master module.
The complete structure of the modules can be seen in the following picture. The signals that connect the modules are displayed on the arrows.



For use with real audio data, the I2S Master is connected to a bus that is fed from a RISC-V CPU. In that case the “sine_generator” module is replaced with the “cpu_bus_logic” module as seen in the following picture.



3. Completed Tasks - Digital Design

3.1. SPI Master

3.1.1. Preparation Sheet 1 & 2

3.1.1.1. Preparation 1.1: The Audio Codec

a) Find out how to connect the codec to a device like an FPGA. The IC has two ports. What are their purposes? List all pins of these ports with direction and purpose.

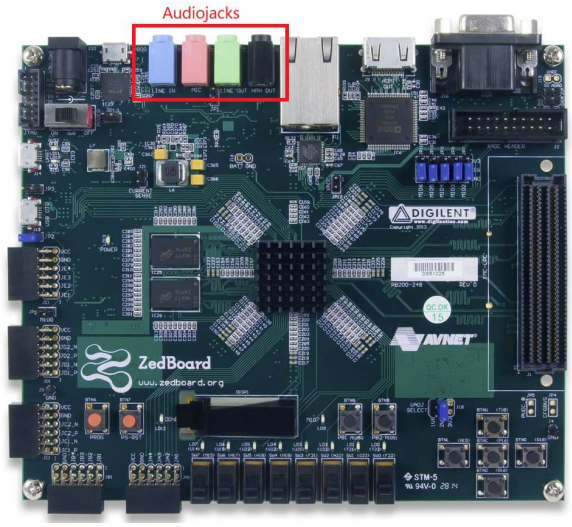
Two Ports:[1]
Control Port: (page 38) For setting I²C/SPI Mode (Read/Write)
The ADAU1761 pull the CLAT

Table 20. Control Port Pin Functions

Pin Name	I²C Mode	SPI Mode
SCL/CCLK	SCL: input clock	CCLK: input clock
SDA/COUT	SDA: open-collector input/output	COUT: output
ADDR1/CDATA	I²C Address Bit 1: input	CDATA: input
ADDR0/CLATCH	I²C Address Bit 0: input	CLATCH: input

b) We will use the headphone output of the ADAU1761. How is the headphone jack of the ZedBoard connected to the ADAU1761?

The headphone jack of the Zedboard will not be connected to the ADAU1761, since the audio data will be transferred through I2S.



The audio jacks on the zedboard are color coded. Numbers in brackets are the pin numbers:

Black: (Headphone out) Ports: LHP(20)/RHP(19)

Green: (Line out) Ports: LOUTP(18)/ROUTP(15)

Pink: (MIC in) Ports: LINN(11)/RINN(13)

Blue: (LINE in) Ports: RAUX(14)/LAUX(6)

The pins are connected with DC decouplers (Capacitors) to the chip.[2]

[1] (Analog Devices, 2021)
[2] (Trenz Electronic, 2021)

d) For our project, we will use a sampling rate of 48 kHz. If we want to use the MCLK input with the default clock divider, what frequency do we have to apply to MCLK?

The default clock divider is 256 according to the datasheet. With a desired fs = 48kHz, and the default clock divider, it is necessary to apply a frequency of $f_s = 256 * 48kHz = 12{,}288MHz$to the MCLK input.

Preparation 1.2: The Control Port

The ADAU1761 control port uses I2C or SPI transmission standard. In the PSoC lab, we will use the SPI transmission protocol.

a) The ADAU1761 is in I2C mode after power-on. How do you switch to SPI mode? For data transmission, we need to know how the ADAU1761 is interpreting the received data.

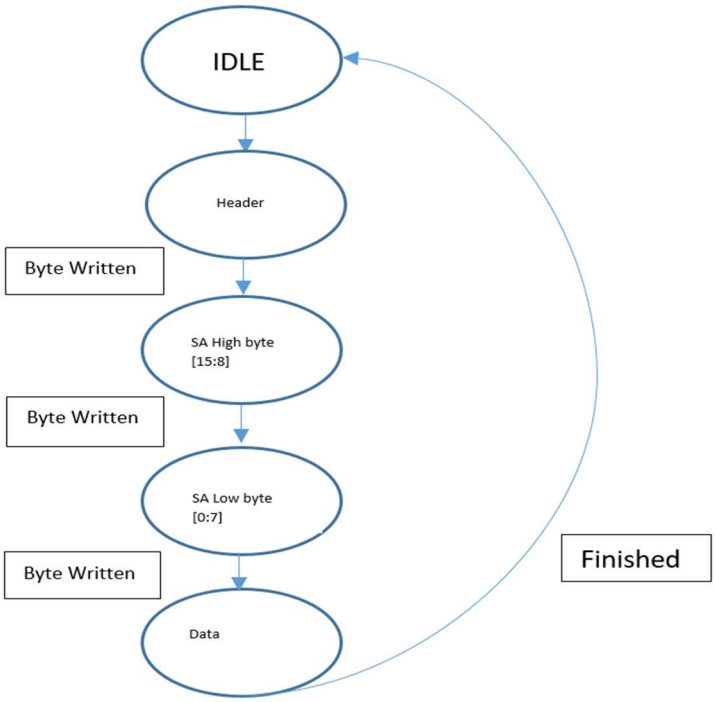
Default is the I²C mode, to change to the SPI mode the CLATCH Pin must be pulled down three times:

2. SPI Protocol:

By default, the ADAU1761 is in I2C mode, but it can be put into SPI control mode by pulling CLATCH low three times. This is done by performing three dummy writes to the SPI port (the ADAU1761 does not acknowledge these three writes). Beginning with the fourth SPI write, data can be written to or read from the IC. The ADAU1761 can be taken out of SPI mode only by a full reset initiated by power cycling the IC.

The SPI port uses a 4-wire interface, consisting of the CLATCH, CCLK, CDATA, and COUT signals, and it is always a slave port. The CLATCH signal should go low at the beginning of a transaction and high at the end of a transaction. The CCLK signal latches CDATA on a low-to-high transition. COUT data is shifted out of the ADAU1761 on the falling edge of CCLK and should be clocked into a receiving device, such as a microcontroller, on the CCLK rising edge. The CDATA signal carries the serial input data, and the COUT signal carries the serial output data. The COUT signal remains three-state until a read operation is requested. This allows other SPI-compatible peripherals to share the same readback line. All SPI transactions have the same basic format shown in Table 23.

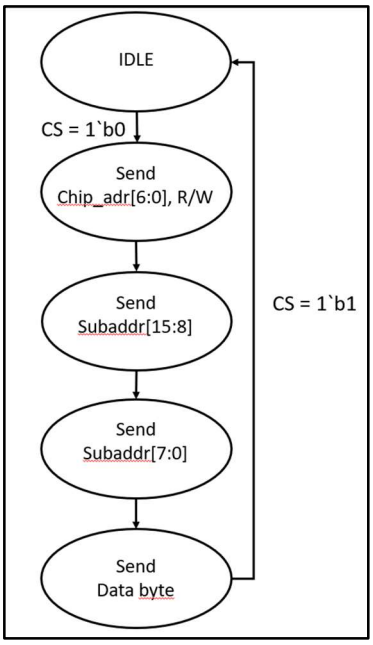
c) Draw a state chart describing a state machine which implements the transmission protocol of the ADAU1761. Make sure to include one distinct state for each data byte.



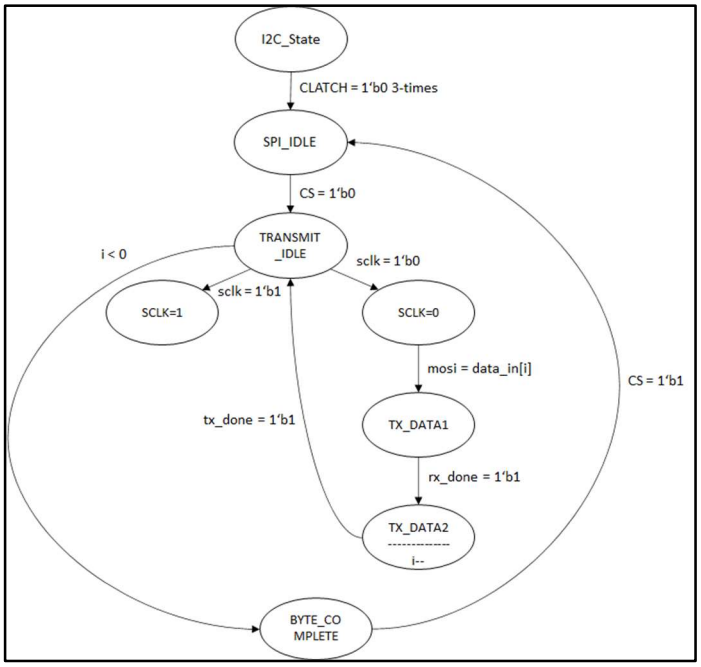
To send the actual data, we now need to know how the transmission works:

e) Design a state machine for transmitting one byte of data using the SPI protocol and include all signals involved. Also think about the data structure to store the byte while transmitting it.

The data byte is to be stored in an array of [7:0].



g) Now, design a state machine for the switching from I2C to SPI using your answer to question 1.2a. Try to reuse the state machine of question 1.2f.



State machine for general state machine (1.2g)

3.1.1.2. Preparation 2.1: The Control Port (2)

In exercise P1.2, we designed a state machine for the transmission of data.

a) Check the ADAU1761 datasheet for the maximum transmission frequency of the control port and determine a reasonably fast clock rate to use for data transmission.

P11. 10MHz (SPI port has the maximum transmission frequency)

3.1.1.3. Preparation 2.2: ADAU1761 Configuration

Now we have a means to write data to configure the ADAU1761 codec.

a) Look up the necessary configuration commands for our use case. Follow the signal path in the schematic diagram of the ADAU1761 and complete your list with the command reference from the data sheet. Sort the commands at the end according to the data sheet. We will use serial input data via the data port. This structure has to be configured, too. The final settings will be found in time. As we don't have to save power here, make sure to turn on all clocks.

Configuration commands: [4]

Input clock frequency

R0: 0x4000
Data: 0x5

Selection of clock edge to sample
R15: 0x4015
Data: tbd

Converter Control
R17: 0x4017
Data: 0x6

Playback Signal Path
Left side:
R22: 0x401C
Data: 0x21

Right side:
R24: 0x401E
Data: 0x41

Unmute headphone outputs
R29: 0x4023
Data: 0x3 (for plain unmuting), 0xD7 (for gain = 0dB)

R30: 0x4024
Data: 0x3 (for plain unmuting), 0xD7 (for gain = 0 dB)

Configuration for Stereo sound
R36: 0x402A
Data: 0x7

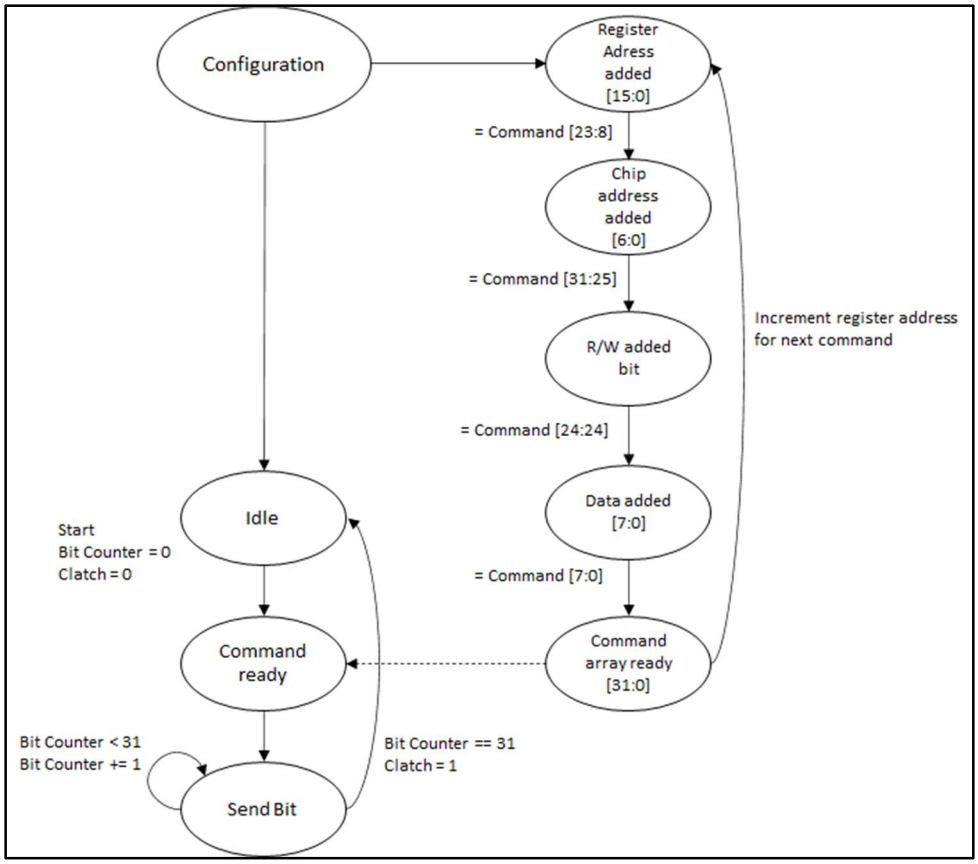
Data transmission clock rate
R64: 0x40F8
Data: tbd

Clock enable
R66: 0x40FA
Data: 0x3

the missing commands were added during implementation

This list has to be transmitted to the ADAU1761.

b) Design a state machine to feed this list to your state machine from exercise P1. You do not have to draw all states for each command from your list. Instead, design a state machine to feed 3 commands to the other state machine and then list all changes you will have to make to scale it up to your whole list. Hint 1: This should be similar to Preparation 1.2f. Hint 2: A data array can largely decrease the design effort.



General information to the state machine:
The state machine from task 1.2f was extended with a command array consisting of the register address, the chip address, the read/write bit and the data byte. With this state machine it is possible to continuously send data in the correct SPI format.

To scale up the state machine so it includes the configuration from task 2.2a we added the configuration state. Here the registers for initialization (R0, R15, R17, R22, R24, R39, R30, R36 and R64) are filled with the correct values before the actual data sending process is being started.

3.1.2. Implemented solution

3.1.2.1. Code:

The module of the spi_master requires different inputs to function correctly in the chosen setting. Besides the standard clock (clk) and the reset (reset) input there is a 32-bit data_in vector and a corresponding valid flag. This vector loads the commands waiting to be processed. The “ready” output is used for communication between other modules. The output “cdata” is one bit of the data_in vector, which is being prepared for sending due to the connection only allowing the sending of one bit at a time. The output “cclk” is a slower clock compared to the system clock, that is

required for modules that must run on a clock with a lower frequency. The last output is the clatch_n is used for example for entering the SPI mode of the ADAU.

```
module adau_spi_master(
        input clk,
        input reset,

        input [31:0] data_in,
        input valid,
        output reg ready,
        output reg cdata,
        output reg cclk = 1,
        output reg clatch_n
    );
```

Additionally, a number of registers were being used. The “clk_div” is a counter used for dividing the system clock down to the frequency that is required. The “Bit_Counter” is a countdown for iterating over the data_in vector to generate the output “cdata”. Due to the fact that a given input “data_in” may change before every bit was sent, the bit register “temp_save_reg” is used to copy the incoming data_in stream so the entire data can be sent without the need of data_in being stable.

```
reg [6:0] clk_div = 0;
    reg [6:0] Bit_Counter = 7'b0011111;
    reg [31:0] temp_save_reg; //Zwischenspeicher
    reg cclk_counter;
    reg [1:0]state;

    localparam S_IDLE = 1'b0;
    localparam S_SEND_BIT = 1'b1;
    localparam S_CLATCH = 2'b10;
```

The actual functionality is implemented in a state machine:
On every positive edge of the clk the reset condition is being checked. If the system is being reset, then it enters the waiting state “S_IDLE” and the Bit_Counter and cclk_counter is set to their starting values.

```
// STATE MACHINE
    always @(posedge clk) begin
            if(reset == 1) begin
                ready <= 1; //module is ready after reset
                state <= S_IDLE; //starting state is idle
                //cclk <= 0;
                cclk_counter <= 0;
                Bit_Counter <= 32;
                cdata <= 0;
            end else begin
```

After the reset, the state “S_IDLE” is being used. As previously said, this state is used as a waiting state until the “valid” signal is set to one, meaning that the input data in (data_in) is valid for sending. Furthermore, the “clatch_n” is set to high and the “ready” bit is set to zero since the module is busy while sending.

```
  case (state)
                        S_IDLE: begin // input valid ready hier rein
                            //led <= 3'b010; // LED
                            if (valid == 1) begin
                                ready <= 0;
                                Bit_Counter <= 32;
```

```
                state <= S_SEND_BIT;
                temp_save_reg <= data_in;
                clatch_n <= 1;
            end
        end
```

The “S_SEND_BIT” state contains a clock divider for the cclk that counts up from 0 to 11 until it switches the value of the cclk, creating a clock signal with a 24th of the frequency of the clock used for counting. This results in the cclk having a frequency of around 5 MHz. Since “cdata” is being sampled on the rising edge of the clock, it has to be assured that the state machine is working on the falling edge of the clock, so the signals are stable while they are being read. This is implemented by checking for the cclk to be high in an if-statement in combination with the whole state machine triggering on the rising edge of the system clock. During the “S_SEND_BIT” the data is being sent. The cdata bit iterates over the temp_save_reg which contains the input data. After all 32 bits are sent the Bit_Counter is 0 and the state switches to “S_CLATCH”.

```
S_SEND_BIT: begin

                //cclk_counter
                if(clk_div == 11) begin
                    clk_div <= 0; //1b`0
                    cclk <= ~cclk; // 1b`1

                    //triggers at negative edge
                    if(cclk == 1) begin
                        Bit_Counter <= Bit_Counter - 1;
                        clatch_n <= 0;
                        if (Bit_Counter == 0) begin
                            state <= S_CLATCH;
                            clatch_n <= 1;
                            cclk <= 1;
                            cclk_counter <= 0;
                        end else begin
                            cdata <= temp_save_reg[Bit_Counter - 1];  //iterates over
data
                        end

                    end
                end
                else begin
                    clk_div <= clk_div + 1;
                end
                //led <= 3'b100; // LED
            end
```

The state “S_CLATCH” is used as another wait state which acts as a buffer after the sending stage and before going back into “S_IDLE”.

```
                S_CLATCH: begin
            if(clk_div == 12) begin
                ready <= 1;
                clk_div <= 0;
                state <= S_IDLE;
            end
            else begin
```

```
                            clk_div <= clk_div + 1;
                        end
                    end
                endcase
            end
    end
endmodule
```

3.1.2.2. Challenges:

temp_save_register:

In the beginning the cdata was directly generated by iterating over data_in register. Due to the possibility of data_in changing during the reading process we chose to copy the input data into another register called “temp_save_reg” to circumvent this issue.

Implementation of the clock divider:

Initially a global clock divider like shown below was used for the generation of the cclk:

```
// // CLOCK DIVIDER
    //             if(clk_div == 11)begin
    //                 clk_div <= 0; //1b`0
    //                 cclk <= ~cclk; // 1b`1
    //             end
    //             else begin
    //                 clk_div <= clk_div + 1;
    //             end
```

This can lead to problems due to the counting variable being reset when e.g., the “S_SEND_BIT” state is started. If the cclk is running globally and the switch to the “S_SEND_BIT” state happens, the counting variable isn’t set to a specific value. So, the first change of the cclk may happen earlier in comparison to a reset variable. To avoid possible issues, it is best practice to implement the clock dividers locally in each state where they are needed.

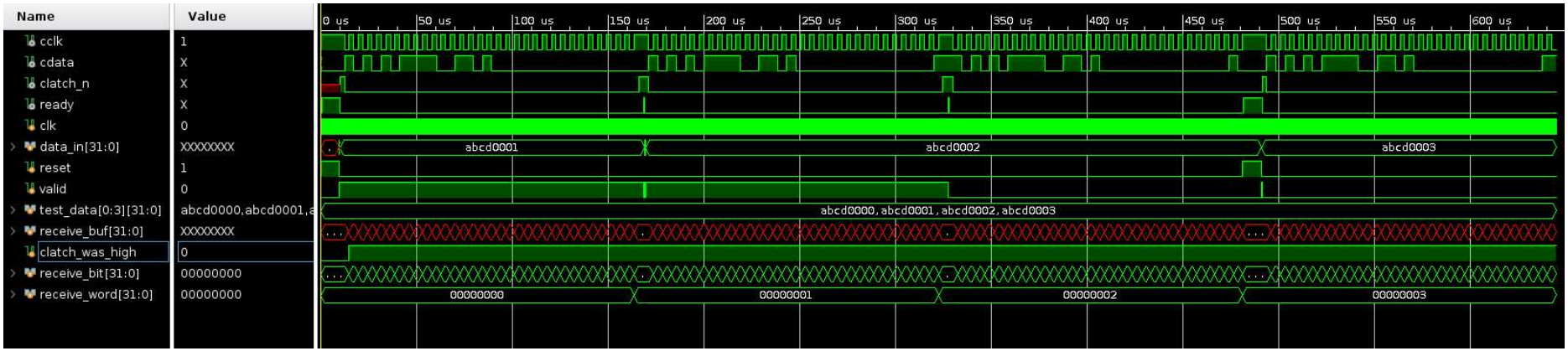
Testbench for the SPI master

To test the SPI master

```
`timescale 1ns / 1ps

module tb_spi_master();
    wire cclk;
    wire cdata;
    wire clatch_n;
    wire ready;

    reg clk;
    reg [31:0] data_in;
    reg reset;
    reg valid;

    reg [31:0] test_data [0:3];
    initial begin
```

```
    test_data[0] = 32'habcd0000;
    test_data[1] = 32'habcd0001;
    test_data[2] = 32'habcd0002;
    test_data[3] = 32'habcd0003;
end
adau_spi_master uut(
    .ready (ready),
    .cdata (cdata),
    .cclk (cclk),
    .clatch_n (clatch_n),
    .clk (clk),
    .reset (reset),
    .data_in (data_in),
    .valid (valid)
);
```

The Vivado signal analyzer of the SPI master test bench



With the following console message:

```
Received data (want: abcd0000, got: abcd0000)
Received data (want: abcd0001, got: abcd0001)
Received data (want: abcd0002, got: abcd0002)
Received data (want: abcd0003, got: abcd0003)
Test OK
$finish called at time : 645100 ns : File "/home/ws/uqtvy/Dokumente/psoc_fpga/src/sim/tb_spi_master.v" Line 73
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_spi_master_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1ms
launch_simulation: Time (s): cpu = 00:00:09 ; elapsed = 00:00:10 . Memory (MB): peak = 6589.902 ; gain = 14.797 ; free physical = 11181 ; free virtual = 13396
```

3.1.2.3. Testbench for the SPI master and the command list

To be able to test the communication between the SPI master and the command list a test bench was implemented. The idea is that after a reset the ADAU must be initialized by writing and setting certain registers. This data is provided by the command list and must be read bit by bit into the ADAU. Apart from the standard inputs like a reset and clock functionality, there is also the wire “adau_init_done” which is set to high by the command list after the initialization phase is completed in its entirety.

```
module tb_adau_spi_master_and_command_list();
    //Inputs and Outputs of modules
    reg clk;
    reg reset;
    wire adau_init_done;
    reg clk32 = 0;
    reg [6:0] clk_div = 0;
    // wires between modules
    wire ready;
    wire [31:0] data_in;
    wire valid;
    // Instantiations
    adau_spi_master uut_1(
        //Inputs
        .clk        (clk), //this clock runs faster
        .reset      (reset),
        .data_in    (data_in),
        .valid      (valid),
        //Outputs
        .ready      (ready),
        .cdata      (cdata),
        .cclk       (cclk),
        .clatch_n   (clatch_n)
    );
    adau_command_list uut_2(
        //Inputs
        .spi_ready      (ready),
        .reset          (reset),
        // one cycle spans an entire command (32 clk cycles):
        .clk            (clk32),
        //Outputs
        .adau_init_done (adau_init_done),
        .command        (data_in),
        .command_valid  (valid)
        //output [4:3] led
    );
```

A slower clock is created with a clock divider similar to the one in the SPI master. Here, the clock divider counts from 0 to 15 and inverts the bit afterwards, meaning that one cycle of the “clk32” clock equals 32 regular cycles of the input clock. This comes in handy due to the commands being 32 bits long.

```
  initial clk <= 0;
    always #100 clk <= ~clk;


   // CLOCK DIVIDER (one clk32 cycle = 32 regular clock cycles)
   always @(posedge clk) begin
```

```
        if(clk_div == 15) begin
            clk_div <= 0; //1b`0
            clk32 <= ~clk32; // 1b`1
        end
        else begin
            clk_div <= clk_div + 1;
        end
    end
```

The following part of code is used for starting the test bench with a reset so the behavior of the modules can be observed in the Vivado signal analyzer.

```
    //task handshake;
     initial begin
         // resetting
         reset <= 1;
         repeat(2) begin
             @(posedge clk);
         end
         reset <= 0;
     end
endmodule
```

3.2. ADAU Command List & Debug session

3.2.1. Implemented solution

3.2.1.1. Code:

First the input and output ports will be defined. The spi ready signal will later come from the SPI master. rest and clk signal will come from the fpga_top module from the clock wizard function. The command_valid signal will be sent to the SPI master module to indicate that the command is going to be sent out from the command list. Also, the command will be sent out. When all the data has been sent to SPI, the adau_init_done signal will be sent out.

```
module adau_command_list(
        // TODO: Add ports
        input spi_ready,
        input reset,
        input clk,
        output reg adau_init_done,
        output reg [31:0] command,
        output reg command_valid
    );
```

The used parameters and register are going to be defined next. CMDs array carries 15 commands with 32 bits of messages each. The state will be set for the state machine. There are in total two states: send command state and idle state. The cmd_index will indicate which command will be sent from the cmds array.

```
    reg [31:0] cmds[0:14]; //there are 15 commands, each has 32 bits of messages
    reg [0:0] state; //there are only two states. one bit is enough
    //send command state or idle state
    parameter SEND_C = 1'b0, IDLE = 1'b1;
    //set an index to show the exact position in cmds array
    reg [5:0] cmd_index = 5'b00000;
```

Then the commands to be sent are going to be initialized in the initialize part. The top uncomment commands are going to be sent when the SPI is in slave mode. The lower parts of the code in comment are going to be sent when the SPI is in master mode.

```
  initial begin
        cmds [0] = 32'h00_0000_00;
        cmds [1] = 32'h00_0000_00;
        cmds [2] = 32'h00_0000_00;
        cmds [3] = 32'h00_4000_01;
        cmds [4] = 32'h00_4015_00;
        cmds [5] = 32'h00_4016_40;
        cmds [6] = 32'h00_401C_21;
        cmds [7] = 32'h00_401E_41;
        cmds [8] = 32'h00_4023_E7;
        cmds [9] = 32'h00_4024_E7;
        cmds [10] = 32'h00_4029_03;
        cmds [11] = 32'h00_402A_03;
        cmds [12] = 32'h00_40F2_01;
        cmds [13] = 32'h00_40F9_FF;
        cmds [14] = 32'h00_40FA_01;
        //commands for initializing master mode (task 2.3c)
        // cmds [0] = 32'h00_0000_00;
```

```
        // cmds [1] = 32'h00_0000_00;
        // cmds [2] = 32'h00_0000_00;
        // cmds [3] = 32'h00_4000_01;
        // cmds [4] = 32'h00_4015_01;
        // cmds [5] = 32'h00_4016_40;
        // cmds [6] = 32'h00_401C_21;
        // cmds [7] = 32'h00_401E_41;
        // cmds [8] = 32'h00_4023_E7;
        // cmds [9] = 32'h00_4024_E7;
        // cmds [10] = 32'h00_4029_03;
        // cmds [11] = 32'h00_402A_03;
        // cmds [12] = 32'h00_40F2_01;
        // cmds [13] = 32'h00_40F9_FF;
        // cmds [14] = 32'h00_40FA_03;
    end
```

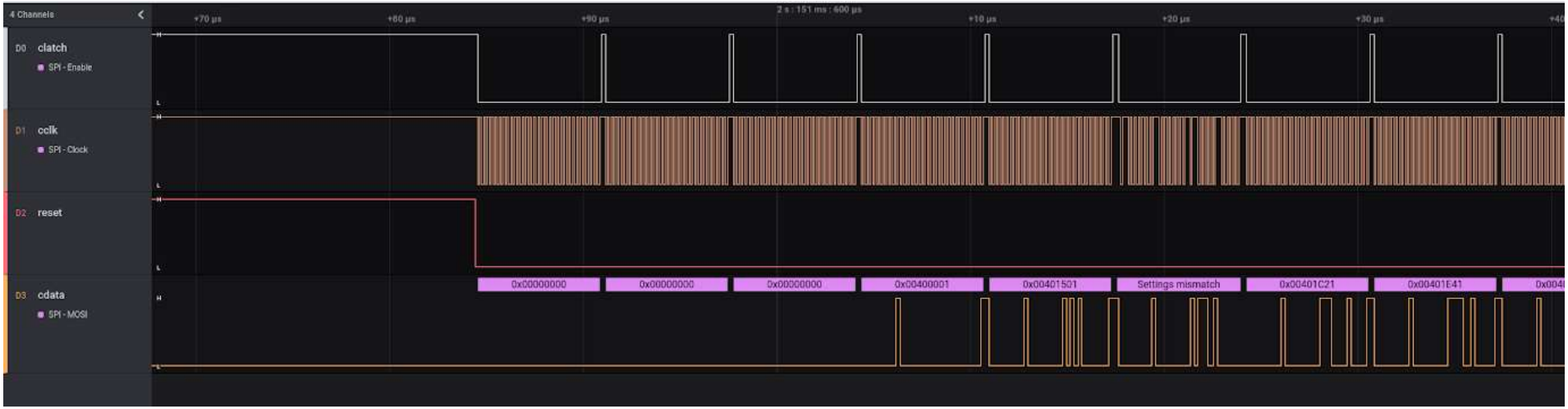
The initial state of the state machine is set as the send command state. When the reset signal arrives, the codes shown below will initialize the state machine.

```
always @(posedge clk) begin
        /*when reset signal arrive
        set the state to send code SEND_C
        set the index to 0
        commands are ready to be sent
        command_valid set to high
        */
        if(reset == 1'b1) begin
            state <= SEND_C;
            cmd_index <= 5'b00000;
            //command <= cmds[0];
            adau_init_done <= 0;
            command_valid <= 1;
            command <= cmds[cmd_index];
        end
```

After the state machine is reset, the machine gets into the state SEND_C, as it was initialized before. And it starts to send the commands and increments the index one by one.
When all the commands are sent out, the state machine will change to idle state and the adau_init_done signal will be set to high.

```
else
            case (state)
                SEND_C: begin //state send command
                    if(spi_ready) begin //if spi is initialized: spi_ready=1
                        if(cmd_index <= 13) begin //if there are still unsent commands
                            //send out the next command
                            cmd_index <= cmd_index + 1;
                            command <= cmds[cmd_index + 1];
                        end
                        //after all the commands are sent, goto idle state and set init_done to high
                        else begin
                            state <= IDLE;
                            adau_init_done <= 1; //adau_init_done high when all data has been sent
                        end
                    end
                end
                     IDLE: begin
                    command_valid <= 0;
                end
                default:;
            endcase
```

In- and output signals of the spi_master module while running on an FPGA (recorded with the logic analyzers):
The screenshot below shows the result for the adau_command_list and spi_master test in the logic analyzer with the attached signals clatch, cclk, rest and cdata. By setting the analyzing mode of the logic analyzer to SPI mode, the signals that are transferred through the SPI can be read on screen. By comparing them to our original set of commands, the content matches.

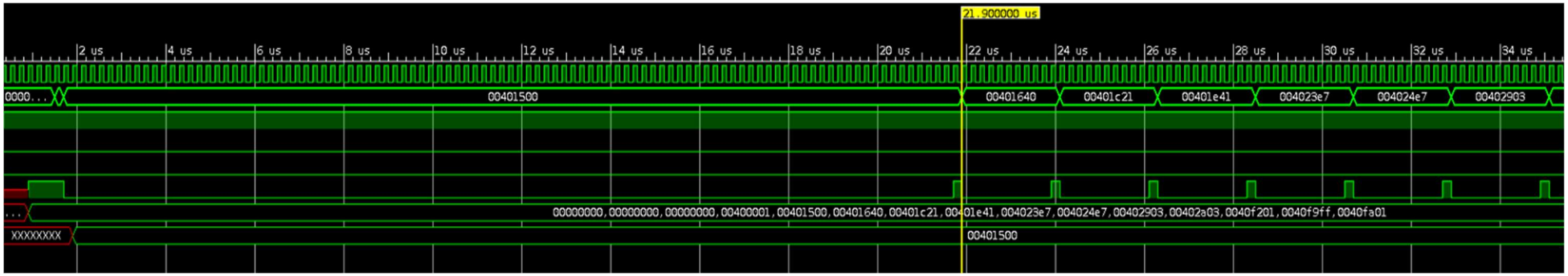


Shown in the graph above are the commands of the ADAU_command_list getting written onto the cdata output. Beginning with command zero it sends the commands one after another until the initialization is complete.

The command list for reference:

```
initial begin
        cmds [0] = 32'h00_0000_00;
        cmds [1] = 32'h00_0000_00;
        cmds [2] = 32'h00_0000_00;
        cmds [3] = 32'h00_4000_01;
        cmds [4] = 32'h00_4015_00;
        cmds [5] = 32'h00_4016_40;
        cmds [6] = 32'h00_401C_21;
        cmds [7] = 32'h00_401E_41;
```

The screenshot below shows the simulated results of the adau_command_list module in combination with the SPI master module. As we can see, the sent out and received commands match correctly.



3.2.1.2. Debug session:

Extend the fpga_standalone_top module and constraints file to add debug output signals. Debug through the logic analyzer.
At this section, the debug settings, the ports from this module and the connection between the modules should be configured in the fpga_standalone_top module. In order to do so, the ports that are being used from the former modules, the adau_command_list, the adau_spi_master, the i2s_generator and the sine_generator, should be sorted out and compared to see which one is connected with the top module. The names should be connected correctly.

```
module fpga_standalone_top(
        // system clock
        input sys_clk,

        // for debugging
        output [7:0] led,
        output [7:0] debug,
        input [7:0] dip,
        input btn_c,
        input btn_d,
        input btn_l,
        input btn_r,
        input btn_u,

        // ADAU signals
        output ac_mclk,
        output cdata,
        output sdata,
        output cclk,
        output clatch_n,
        output bclk,
        output lrclk
    );
```

First, in the TODO section the ADAU signals that are being used in other modules should be added and defined as output here, as those output signals from the top module would be defined as input signals from the other modules.

```
//assign the debug port to the signals that need to be read on logic analyzer
    assign debug[0] = clatch_n;
    assign debug[1] = cclk;
    assign debug[2] = reset;
    assign debug[3] = bclk;
    assign debug[4] = lrclk;
    assign debug[5] = cdata;
    assign debug[6] = sdata;
    //assign debug[7] = sys_clk;
```

Then the debug ports will be assigned with the signals that need to be read later through the logic analyzer. The interesting signals will be the ones listed above. As the sys_clk cannot be read anyway. The eighth debug port debug[7] will be left apart in the comment line.

After that, constraints in the constraint file zedboard.xdc should be added and configured to connect the ports from the FPGA with the actual pins on the Zedboard. The debug ports will be also added in the constraint file to assign them with the actual ports on the Zedboard, so that the signals can be read with a logic analyzer with physical pins connecting the physical ports.
As it is shown in the table, for example, the SPI Latch Signal AC-ADR0 is connected with the pin AB1.

Table 9 - CODEC Connections

Signal Name	Description	Zynq pin	ADAU1761 pin
AC-ADR0	I2C Address Bit 0/SPI Latch Signal	AB1	3
AC-ADR1	I2C Address Bit 1/SPI Data Input	Y5	30
AC-MCLK	Master Clock Input	AB2	2
AC-GPIO2	Digital Audio Bit Clock Input/Output	AA6	28
AC-GPIO3	Digital Audio Left-Right Clock Input/Output	Y6	29
AC-GPIO0	Digital Audio Serial-Data DAC Input	Y8	27
AC-GPIO1	Digital Audio Serial Data ADC Output	AA7	26
AC-SDA	I2C Serial Data interface	AB5	31
AC-SCK	I2C Serial Data interface	AB4	32

[3]

Then according to the commands listed in the Programmable Logic Master User Constraints, in the file ZedBoard Master XDC Rev C_D v3, the used digital ports can be configured with the physical pins. It is important to set the right IO standard voltage of ports to the right amount according to the User constraints. For example, the Audio Codec is connected with Bank 13. According to the file "# Note that the bank voltage for IO Bank 13 is fixed to 3.3V on ZedBoard.", which indicates the voltage here should be modified as 3.3V using the command "

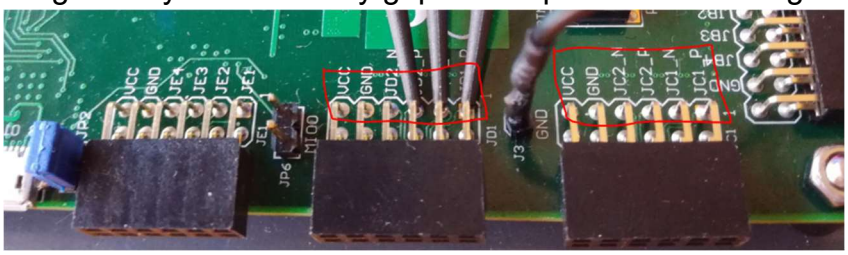
```
set_property IOSTANDARD LVCMOS33 [get_ports {signal}];"
```

```
# Audio Codec - Bank 13
#----------------------------------------------------------------------
set_property PACKAGE_PIN AB1 [get_ports {ac_addr0_clatch}];  # "AC-ADR0"
set_property IOSTANDARD LVCMOS33 [get_ports {ac_addr0_clatch}]; # set the voltage to 3.3V
set_property PACKAGE_PIN Y5  [get_ports {ac_addr1_cdata}];  # "AC-ADR1"
set_property IOSTANDARD LVCMOS33 [get_ports {ac_addr1_cdata}];
set_property PACKAGE_PIN Y8  [get_ports {ac_dac_sdata}];  # "AC-GPIO0"
set_property IOSTANDARD LVCMOS33 [get_ports {ac_dac_sdata}];
#set_property PACKAGE_PIN AA7 [get_ports {AC_GPIO1}];  # "AC-GPIO1"
set_property PACKAGE_PIN AA6 [get_ports {ac_bclk}];  # "AC-GPIO2"
set_property IOSTANDARD LVCMOS33 [get_ports {ac_bclk}];
set_property PACKAGE_PIN Y6  [get_ports {ac_lrclk}];  # "AC-GPIO3"
set_property IOSTANDARD LVCMOS33 [get_ports {ac_lrclk}];
set_property PACKAGE_PIN AB2 [get_ports {ac_mclk}];  # "AC-MCLK"
set_property IOSTANDARD LVCMOS33 [get_ports {ac_mclk}];
set_property PACKAGE_PIN AB4 [get_ports {ac_scl_cclk}];  # "AC-SCK"
set_property IOSTANDARD LVCMOS33 [get_ports {ac_scl_cclk}];
```

[3] (Avnet, 2021), p. 17

```
#set_property PACKAGE_PIN AB5 [get_ports {AC-SDA}];  # "AC-SDA"
```

First, it is very important to choose which physical pins we are going to use as debug pins, as they are going to be connected to the logic analyzer with metal grips. We are going to choose the outside pins from the socket on board, so that the metal grips from the logic analyzer can easily grip on the pins without falling off.



And then we are going to configure the right physical pin to the digital Ports, which has been assigned to the corresponding signals before in the *fpga_standalone_top.v* file. The code below shows how the ports are assigned to the pins with comments beside, showing which signal is connected.

```
# JA Pmod - Bank 13
#----------------------------------------------------------------------
set_property PACKAGE_PIN W7 [get_ports {debug[1]}];  # "JD1_N" cclk
set_property IOSTANDARD LVCMOS33 [get_ports {debug[1]}];
set_property PACKAGE_PIN V7 [get_ports {debug[0]}];  # "JD1_P" clatch_n
set_property IOSTANDARD LVCMOS33 [get_ports {debug[0]}];
set_property PACKAGE_PIN V4 [get_ports {debug[3]}];  # "JD2_N" bclk
set_property IOSTANDARD LVCMOS33 [get_ports {debug[3]}];
set_property PACKAGE_PIN V5 [get_ports {debug[2]}];  # "JD2_P" reset
set_property IOSTANDARD LVCMOS33 [get_ports {debug[2]}];
set_property PACKAGE_PIN AB6 [get_ports {debug[4]}];  # "JC1_N" lrclk
set_property IOSTANDARD LVCMOS33 [get_ports {debug[4]}];
set_property PACKAGE_PIN AB7 [get_ports {debug[5]}];  # "JC1_P" cdata
set_property IOSTANDARD LVCMOS33 [get_ports {debug[5]}];
set_property PACKAGE_PIN AA4 [get_ports {debug[6]}];  # "JC2_N" sdata
set_property IOSTANDARD LVCMOS33 [get_ports {debug[6]}];
set_property PACKAGE_PIN Y4 [get_ports {debug[7]}];  # "JC2_P"
set_property IOSTANDARD LVCMOS33 [get_ports {debug[7]}];
```

3.2.1.3. Challenges

Finding the right document with port configuration

One of the biggest challenges in the debugging and the port configuration part is to find the right pins and ports on the board to connect to the digital signals. In order to do that, we need to read through the documents on the official website of the zedboard. We also need to read through the relevant document to find the right Codec port configuration and assign them to the constraint files. It is extremely hard, as we are not familiar with the files and the right port configuration.

Finding the right standard voltage

Another hard part in this section is to find the right standard voltage supply for the port channel. Without the right voltage, it could happen that the board will be destroyed, or the synthesis part will continue to give the failure reports.

Assign all the digital port with a physical pin

It is also a confusing step, when we are trying to synthesize the whole design and send the code to the hardware. Vivado keeps sending failure reports before all the digital ports are signed to a physical pin, even though there is one of the ports unused. It is said that Vivado needs the complete commands to finish the synthesis part. Otherwise, it will recognize that there is one port missing and refuse to continue the synthesis.

3.3. I2S Master

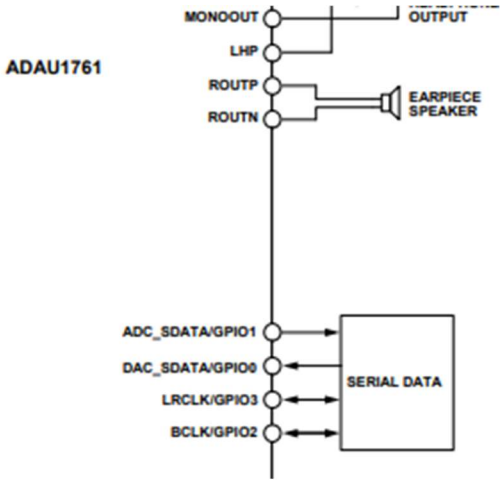
3.3.1. Preparation Sheet 3

3.3.1.1. Preparation 3.1: The ADAU I2S Data Port

At first, we need to know the data format to provide to the codec.

b) Assume we are using the codec's master mode. What signals are provided by the codec? Which of them do we need for sending the data to the codec?

From ADAU to the FPGA, we use LRCLK/GPIO3 and BCLK/GPIO2. From FPGA to ADAU, we use DAC_SDATA/GPIO0. [4]



c) What changes if we use the slave mode?

If we use slave mode, the LRCLK and BCLK are decided by the FPGA side.

d) We want to use data with a sample width of 24 bit, and we want to transmit stereo sound. Is this possible with the specifications of the ADAU1761? Consider also the BCLK clock rate and how it relates to MCLK. The MCLK frequency has been chosen on a previous exercise sheet.

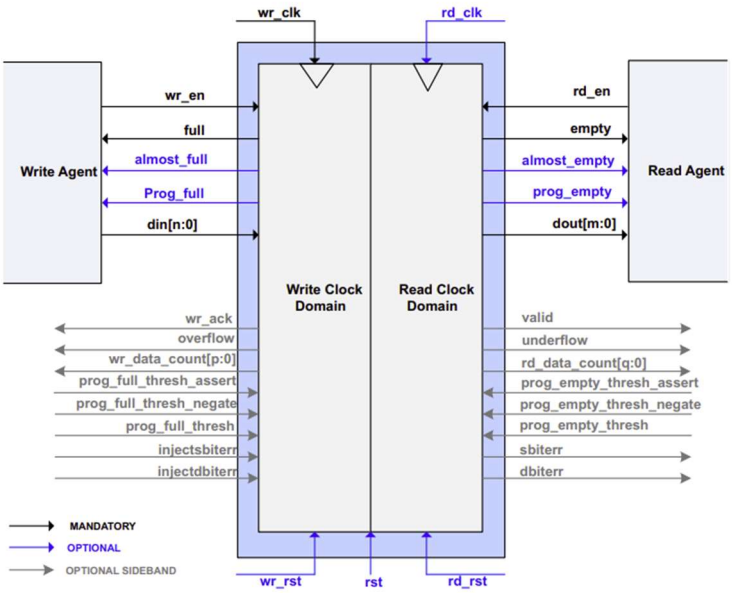
p. 42 In all modes except for the right-justified modes, the serial port inputs an arbitrary number of bits up to a limit of 24.
BCLK = 24 bits * 48 kHz(fs)
MCLK = 256 (divider) * 48 kHz(fs)

[4] (Analog Devices, 2021), p. 21

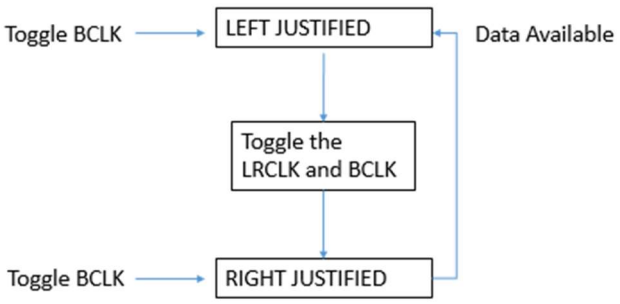
3.3.1.2. Preparation 3.2: FIFOs

d) Give and explain the signals typically used in an independent clock FIFO interface.

A typical independent FIFO interface could look like the one in illustration.[5] The FIFO is split into the write and the read clock domain that both have different in- and outputs but share the same reset (rst). Both sides contain an enable port named rd_en and wr_en and they get supplied with their own clock input (wr_clk and rd_clk respectively). Other crucial ports are the full and empty ports. These can communicate through the interface whether the FIFO is filled completely and cannot take in more data or if it is completely empty and it cannot read any data anymore. The ports din[n:0] and dout[m:0] describe what data width the incoming and outgoing data has.



f) Draw a state chart for a state machine which reads data from the FIFO buffers and writes it in I2S format to the codec. The state machine should work as I2S master, ADAU should be the I2S slave. Include all signals for codec and FIFO in your state machine.



g) Which clock edge do you have to use for the implementation of your state machine to make sure that the data received by the ADAU is correct?

[5] (Xilinx, 2021)

The state machine has to use the falling edge so that the data is valid. This comes from SDATA being sampled on the rising BCLK edge [9]. Therefore the state machine has to also sample on the rising BCLK edge for the data to be valid.

3.3.2. Implemented solution:

3.3.2.1. Code:

The I2S protocol is based on three signals: Bitclock (“bclk”), Left-Right-Clock (“lrclk”) and the serial data (“sdata”). These signals are therefore designed as outputs to this module. The data to be sent is put into the module through “frame_in_l” and “frame_in_r” and valid data is signaled through “write_frame”.

```
module i2s_master(
        //other signals
        input clk_soc,
        input ac_mclk,
        input reset,

        //I2S signals
        output reg bclk,
        output reg lrclk,
        output reg sdata,

        //audio input data
        input [23:0] frame_in_l, frame_in_r,
        input write_frame,
        output full
    );
```

The module consists of a two-state State Machine, one for the transmission of the left channel and one for the transmission of the right channel. Also there are two variables created that help to sequence the code correctly. “rd_en” is used to trigger the readout from the FIFO. Through “data_av_flag” it is ensured that data is only put on the “sdata” bus line, if data was read from the FIFO before.

```
reg [0:0] i2s_state;
localparam S_LEFT = 1'b0;
localparam S_RIGHT = 1'b1;

reg rd_en;
reg data_av_flag;                       //used to signal the first data read from the FIFO
```

The registers “clk_counter” and “Bit_Counter” are used to count the number of clock cycles passed respectively the number of Bits put onto “sdata” during a transmission.

```
reg [6:0] clk_counter;                  //Counts the clock cycles to time the bclk
reg [5:0] Bit_Counter;                  //Counts the number of transmitted bits on the bus
```

When reading in the data from the FIFO, it is stored in a buffer register before outputted onto the bus.

```
wire [23:0] left_data;              //Signal to read in the data for the left channel from the FIFO
reg [23:0] left_data_buffer;        //Buffers the FIFO signal to be put on the bus
wire [23:0] right_data;             //Signal to read in the data for the left channel from the FIFO
reg[23:0] right_data_buffer;
```

Since we are using 2 FIFOs, one for each output channel, the full and empty signals will be logically coupled in our implementation.

```
wire full_l, full_r;
wire empty_l, empty_r;

assign full = full_l | full_r;
assign empty = empty_l | empty_r;
```

The FIFO instantiation is the same for the left and right channel.

```
fifo_generator_0 fifo_left (
  .rst(reset),                          // input wire rst
  .wr_clk(clk_soc),                     // input wire wr_clk
  .rd_clk(ac_mclk),                     // input wire rd_clk
  .din(frame_in_l),                     // input wire [23 : 0] din
  .wr_en(write_frame),                  // input wire wr_en
  .rd_en(rd_en),                        // input wire rd_en
  .dout(left_data),                     // output wire [23 : 0] dout
  .full(full_l),                        // output wire full
  .empty(empty_l),                      // output wire empty
  .wr_rst_busy(wr_rst_busy_l),          // output wire wr_rst_busy
  .rd_rst_busy(rd_rst_busy_l)           // output wire rd_rst_busy
  );

fifo_generator_0 fifo_right (
  .rst(reset),                          // input wire rst
  .wr_clk(clk_soc),                     // input wire wr_clk
  .rd_clk(ac_mclk),                     // input wire rd_clk
  .din(frame_in_r),                     // input wire [23 : 0] din
  .wr_en(write_frame),                  // input wire wr_en
  .rd_en(rd_en),                        // input wire rd_en
  .dout(right_data),                    // output wire [23 : 0] dout
  .full(full_r),                        // output wire full
  .empty(empty_r),                      // output wire empty
  .wr_rst_busy(wr_rst_busy_r),          // output wire wr_rst_busy
  .rd_rst_busy(rd_rst_busy_r)           // output wire rd_rst_busy
);
```

The state machine is mostly controlled through the “clk_counter”. At reset it is loaded with the number 63 and is counting down to 0.

```
always @(posedge ac_mclk) begin

    //clk_counter initialization and reset condition
```

```
    if (clk_counter == 0 || reset == 1) begin
        clk_counter <= 7'b0111111;  // 63       //counts 64 clock cycles, to create 32 bclk cycles
    end else begin
        clk_counter <= clk_counter - 1;
    end
```

The State machine is always initialized at the right channel, because in that state, the data from the FIFOs will be read in. The “Bit_Counter” is reset to 23, so it can count down to 0.

```
//common reset condition
    if (reset == 1) begin
        i2s_state <= S_RIGHT;
        bclk <= 0;
        lrclk <= 1'b1;
        Bit_Counter <= 23;
        rd_en <= 1'b1;
        data_av_flag <= 1'b0;

    end else begin
```

For 48 cycles, while the “clk_counter” is > 16, the 24 bit from the left or right buffer register are put onto “sdata” serially, the “Bit_counter” which controls which bit is put onto the bus, is decremented and the “bclk” is triggered 24 times.

```
if (empty == 0)                     //check if FIFO has already received data

        rd_en <= 0;

        case (i2s_state)
             S_RIGHT: begin              //Right channel data transfer

                if (clk_counter > 16) begin     //check if 24 Bits / 48 cycles have already passed
                    if (data_av_flag == 1) begin    //only transmit data if there is some data in
the buffer
                        if (bclk == 1) begin
                            sdata <= right_data_buffer[Bit_Counter];    //Right channel data is put
on the bus serially
                            Bit_Counter <= Bit_Counter - 1;
                        end

                       bclk <= ~bclk;
                    end
```

While the “Bit_counter” is between 15 and 0, the “bclk” is put to a constant 0, since no data is transmitted in that time. The “Bit_Counter” is reset to 23. When “clk_counter” reaches 2, “rd_en” is put high to prepare the next read operation from the FIFOs. One clock cycle later, the data from the FIFOs is written to the respective buffer registers. The “data_av_flag” signals that a transmission of data is possible now.

```
end else if (clk_counter > 0 && clk_counter <= 15)begin
                    bclk <= 0;
```

```
                    Bit_Counter <= 23;

                    if (clk_counter == 2) begin
                        rd_en <= 1;

                    end else if (clk_counter == 1) begin
                        left_data_buffer <= left_data;      //read in the data from the FIFO for
both channels
                        right_data_buffer <= right_data;
                        data_av_flag = 1'b1;                //set data available flag to signal that
there is now data available
                        lrclk <= 1'b0;
                    end
```

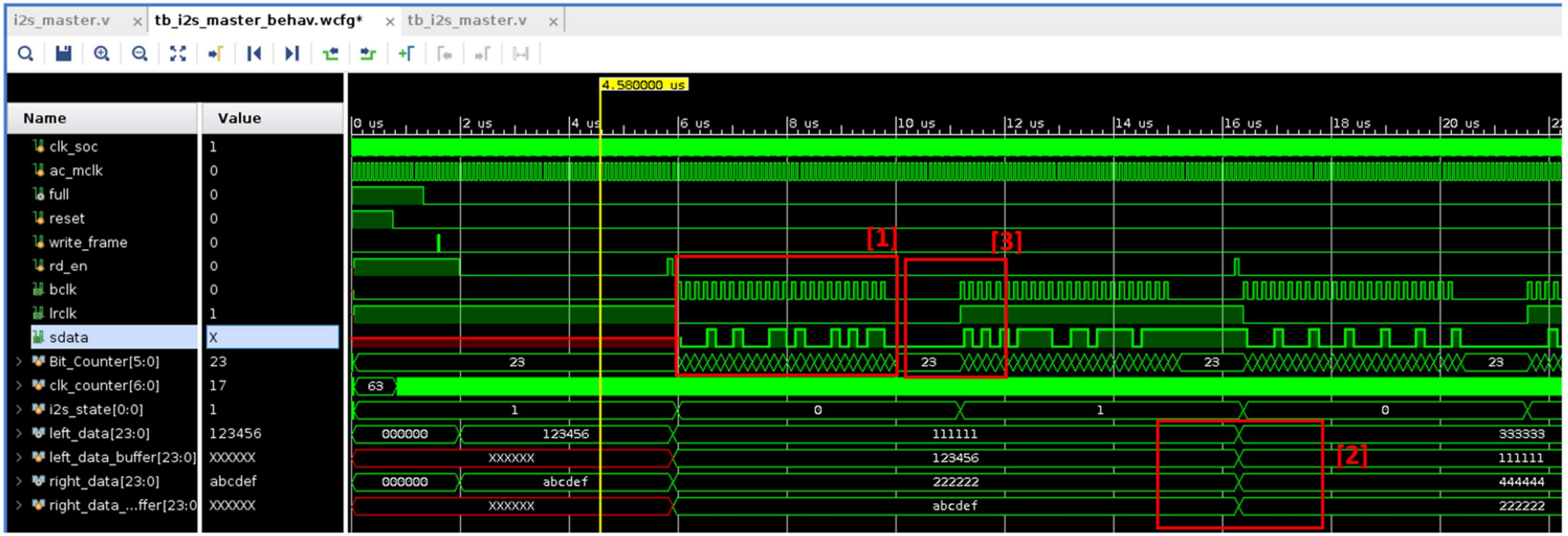
Also the lrclk changes to the next channel, in this case to the left channel. When the “clk_counter” reaches 0, the channel state changes to the other channel.

```
end else if (clk_counter == 0) begin
                    i2s_state <= S_LEFT;        //Change the state to the other channel
                    bclk <= 1'b1;
                end
            end
```

This procedure is equivalent for the left channel. Only without the read operations, since there is always data for both channels read in.

```
S_LEFT: begin               //Left channel data transfer

                //still data left to be sent
                if (clk_counter > 16) begin     //check if 24 Bits / 48 cycles have already passed
                        if (bclk == 1) begin
                            sdata <= left_data_buffer[Bit_Counter]; //Left channel data is put on
the bus serially
                            Bit_Counter <= Bit_Counter - 1;
                        end

                        bclk <= ~bclk;

                //all 24 bits have been send, waiting for 8 bits /16 cycles
                end else if (clk_counter > 0 && clk_counter <= 15)begin
                    bclk <= 0;                  //blck set to 0 while no data is transmitted
                    sdata <= 1'b0;
                    Bit_Counter = 23;           //Counter reset

                    if (clk_counter == 1) begin
                    lrclk <= 1'b1;
                    end


                //32 bits / 64 ac_mclk cycles have passed
                end else if (clk_counter == 0) begin
                    i2s_state <= S_RIGHT;      //Change the state to the other channel
                    bclk <= 1'b1;
                end

            end
```

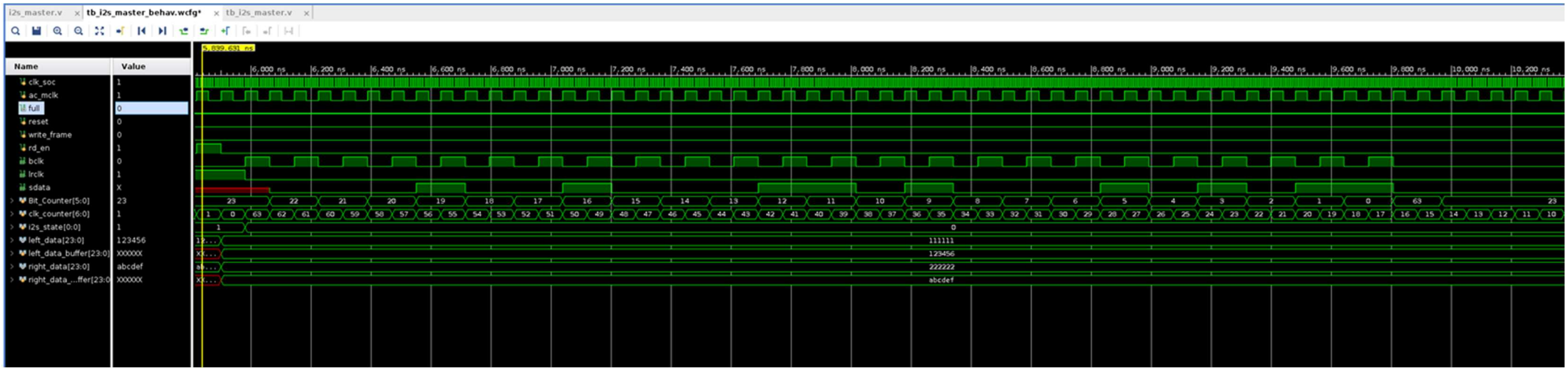
3.3.2.2. Simulation pictures:



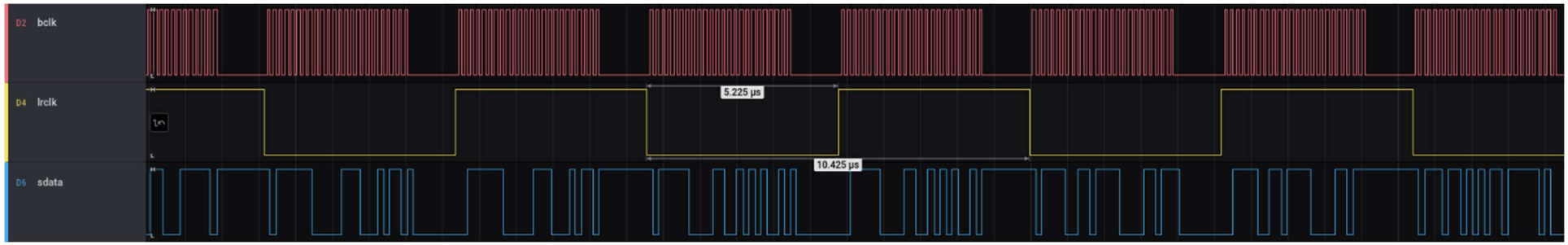
[1] 24 “blck” cycles and the corresponding data transmission on “sdata”, the “Bit_Counter” is counting down from 23 to 0

[2] The data on the wires “left_data” and “right_data” are written into the respective buffer registers

[3] The “blck” stays low for 16 “ac_mclk” cycles until the next transmission begins, the “lrclk” is pulled high and therefore the transmission of the right channel starts



In this picture it is clear how the “Bit_Counter” and “clk_counter” are working. The “clk_counter” is the one that is used to controlled the whole behavior, while the “Bit_Counter” is only responsible to put the right data bit onto “sdata”. After the transmission “blck” stays low and the “Bit_Counter” is reset to 23, while “clk_counter” is still running.



The picture above shows the real signals of the data transmission captured through the logic analyzer. Here it can also be seen that the “bclk” switches 48 times per “lrclk” cycle to transmit the 24 data bits on the “sdata” line. After the 48 cycles, the “bclk” stays low until the next change of the “lrclk”. This shows the desired behavior that was desired when programming.

3.3.2.3. Challenges:

The main challenge we faced while implementing the I2S Master, was to make sure to read in the data from the FIFO in time and not start the transmission of the “bclk” unless there was data to be sent, because this resulted in some failures during test and debugging. We solved this challenge using an additional flag (“data_av_flag”), that is pulled high when data is ready to be sent. Also we realized how critical it is, that the signal flanks are put at the correct timing. For example with the “lrclk”, we had issues when changing it at the same time as the “bclk” starting the next transmission. Because of that, we changed the timing, so that the “lrclk” changes one clock cycle ahead of the “bclk”. The same issue and solution applies to the “rd_en” signal that is used to initialize the data transmission from the FIFO

3.4. Sine Generator

3.4.1. Preparation Sheet 4

3.4.1.1. Preparation 4.2: Implementation on the ZedBoard

With the various components implemented and partially tested in previous labs, we can start to implement our test system on the Digilent ZedBoard.

a) Find your clock speed choices for all clocks used and list them to prepare the clock generation setup.

```
R0 Clock Control 0x00_4000_00
MCLK = 256 * fs = 12.288 MHz
R15 Serial Port 0 0x00_4015_00
LRCLK 50% duty cycle
CCLK = SPI Clock 5 MHz
BCLK = 48 kHz * 48 Bit Clock Cycles(24 bits * 2 channels) = 2.304
MHz

Verilog Clock Generator [4][5]
```

```
`timescale 1ns/1ps
module clock generation();
reg MCLK;
reg CCLK;
reg BCLK;

initial begin
        MCLK = 0;
        CCLK = 0;
        BCLK = 0;
end
always
        #81     MCLK = ~MCLK;
        #200    CCLK = ~CCLK;
        #434    BCLK = ~BCLK;
end
```

3.4.1.2. Preparation 4.3: Sine Wave Generation

To test the system, we also need some sound to output. We will start with a sine wave.

a) One way to implement this function on the FPGA is to use a lookup table. Write a short program (in any programming language you prefer) which generates a case block for Verilog with the sample points for 440 Hz and maximum amplitude. Note

that the output data must be in stereo format, i.e. it uses two 2 channels. For first tests, simply duplicate the channel. For further tests you may use different frequencies for the channels to test whether correct output is generated for the individual channels. In the lab course, you will insert the generated piece of code in your module.

```
import math

samplef = 48000
basef = 440
samples = math.ceil(samplef / basef) # number of samples for one full sinus curve @basef
bitwidth = 24

lut_dec = [None] * samples # List of values in decimal
lut_hex = [None] * samples # List of binary values

numbers_dec = [0] * 5000
numbers_b = [0] * 120
numbers_b_trunc = [0] * 120
numbers_b_final = [0] * 120
bitstring_length = 8

i = 0

hex_format = "{:06"+"x}"

print(f"case(expression)")

while i < samples: # List with sample values

    numbers_dec[i] = i
    numbers_b[i] = bin(numbers_dec[i])
    numbers_b_trunc[i] = numbers_b[i][2:]

    if len(numbers_b_trunc[i]) < bitstring_length:
        numbers_b_final[i] = '0' * (bitstring_length - len(numbers_b_trunc[i])) + numbers_b_trunc[i]


    lut_dec[i] = math.floor((pow(2, bitwidth-1)-1) * math.sin(((2 * math.pi)/samplef) * i * basef ))
# sin for 440Hz and a sample rate of 48kHz

    lut_hex[i] = hex_format.format((lut_dec[i] + (1 << bitwidth)) % (1 << bitwidth))

    numbers_b[i] = bin(numbers_dec[i])

    print(f"{'':<5}8'b{numbers_b_final[i]}{'':<3}:")
    print(f"{'':<10}{bitwidth}'h{lut_hex[i]};")
    print(f"{'':<10}break;")


    i +=1

print(f"endcase")
```

The plotted finished case statement can be found in a separate file also uploaded to ILIAS.
For the implementation in a two-channel environment, the case statement has to be duplicated for the left and right channel.

3.4.2. Implemented solution

3.4.2.1. Code:

To test the system, we also need some sound to output. We will use a sine wave. The sine generator is used for checking if our spi_master, command list and i2s master is working together. Later it is not used again, because we want to output real audio data like music and not just a single tone.

In our Lab exercise we will use the most straightforward method for generating the sine wave using a lookup table. There are more precise methods, but this one is fast and simple and will suffice many of our applications.

We know that the sine quadrants have a simple symmetry, so our table only needs to cover a quarter of a circle: 0 to 90 degree. Our code will handle a complete circle using simple adjustment for each of the four quadrants and this sine look up table is generated using a python script(gen_sine.py). The sin_lut_90x24mem file is the sine wave lookup table which consists of 91 sample points for 440 Hz frequency and maximum amplitude. From the preparation sheet we see that from the figure of sine wave in task 4.1 has a blue curve with frequency of 440Hz and the sample rate of 48kHz. After generating the Lookup table for the sine wave which contains the 91 sample points using the python script (Below code snippet is responsible for generating the samples for 440 Hz and a sample rate of 48kHz),

```
    lut_dec[i] = math.floor((pow(2, bitwidth-1)-1) * math.sin(((2 *
math.pi)/samplef) * i * basef )) # sin for 440Hz and a sample rate of
48kHz
```

In order to feed the audio data to the ADAU1761 through the I2S Master, we implement a new module called sine_generator providing the lookup table for the sine function (Below is the code for module sine_generator). The inputs to the sin_generator module are clock, reset and ready. The output's are register valid and register out which is either 16 bits or 24 bits per channel and we are using 24 bits per channel.

```
module sine_generator(
        // Port initialization
        input clk, //input clock
        input reset, // input reset
        input ready, //input ready

        output reg valid, // output valid
```

```
        output reg [23:0] out //output register [23:0] out
    );
```

After initializing all the ports, we then have to read the data from the look-up table and the command $readmemh(“sin_lut_90x24.mem”) is responsible to read the data from sin_lut_90x24.mem.

```
localparam LUT_HEIGHT = 90; //Number of test data
reg [23:0] lut [0:LUT_HEIGHT-1];
initial $readmemh("sin_lut_90x24.mem", lut); //reading the look-up table

reg [7:0] cnt;
```

Next, we check for reset conditions. So, if there is a reset signal then the count and the valid signal will be reset. If there is an active ready signal, then we increment the counter and then send the data samples from the look up table to out. Lastly, if the counter reaches the maximum count value, then the counter is reset to zero.

```
always @(posedge clk) begin
    if (reset==1) begin //condition for reset
        cnt <=0;
        valid <= 0; //from I2S bus and also ready
        out <= 24'b0;
    end
    else begin
        if (ready == 0) begin //if not reset
            out <= lut[cnt];
            valid <= 1;
            if(cnt < LUT_HEIGHT -1) //next data
                cnt <= cnt +1; //increment the counter
            else
                cnt <= 0; // else don't increment
        end

    end
end

endmodule
```

In the next part, we link the sine_generator module to the i2s_master module in the fpga_standalone_top module.

```
// Linking the sine_generator module to the fpga_standalone_top module
    sine_generator sin(
        .clk(clk_soc),
        .reset(reset),
        .ready(full),
        .valid(write_frame),
        .out(frame_in_l)
        //.out(frame_in_r)

    );
```

These samples are then tested using test bench tb_sine_generator to check if we are getting the audio tone which ensures that spi_master, command list and i2s_master is working properly.

```
module tb_sine_generator;
    // Generate clock
    reg clk = 0;
    always #10 clk <= ~clk;

    reg reset = 1;
    reg ready = 0;

    wire [23:0] out;
    wire valid;

    sine_generator uut(
        // Outputs
        .valid            (valid),
        .out              (out[23:0]),
        // Inputs
        .clk              (clk),
        .reset            (reset),
        .ready            (ready)
    );

    // Wait at least till next rising edge when valid=1
    // Timeout after 1000 ns
    task next_sample;
        begin
            fork
                begin: waiting // waiting for the rising edge
                    @(posedge clk);
                    while (!valid)
                        @(posedge clk);
                    disable timeout;
                end
                begin: timeout //if no rising edge in 1000ns then timeout
                    #1000 disable waiting;
                    $error("VALID did not rise after waiting 1000ns");
```

First, we are generating a clock of periodicity of 20ns.
Here we will be waiting at least till the next rising edge when valid=1 and timeout after 1000ns. In these conditions we are basically waiting for the rising edge and if we do not receive the rising edge then we do a timeout.

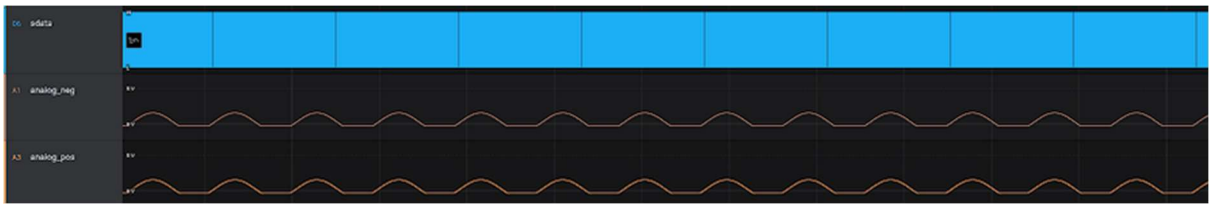
```
$finish;
                end
            join
        end
    endtask

    task check;
        input [23:0] want;
        begin
            next_sample;
            if (out !== want) begin
                $error("bad output data (want: %06x, got: %06x)", want, out);
                $finish;
            end
        end
    endtask
```

```
    initial begin
        // Keep reset high
        repeat(5)
            @(posedge clk);
        reset <= 0;

        // Adjusting the expected values
        $display("checking the first 5 samples");
        ready <= 1;



        // If your sin generator immed
        check(24'h000000);
        check(24'h08edc7);
        check(24'h11d06c);
        check(24'h1a9cd9);
        check(24'h234815);
```

In the task check, we are checking if the samples we are getting are the samples we expected, so the want is the sample which we expect and the out is the sample which is being outputted by the sine generator. We are basically having various test conditions for the samples which we generated are correct or incorrect and we repeat this for all the 90 samples.

```
//Expected number of samples
        $display("testing periodicity (output should repeat after 91 samples)");
        repeat(90 - 5)
            next_sample;

        //Expected values
        check(24'h000000);
        check(24'h08edc7);
        check(24'h11d06c);

        $display("testing that OUT is static when READY is low");
        ready <= 0;
        check(24'h1a9cd9);

        $display("testing that OUT changes after pulling READY high again");
        ready <= 1;
        // h1a5310 was not read yet
        check(24'h1a9cd9);
        check(24'h234815);

        $display("Test OK");
        $finish;
    end
endmodule
```

Finally, the above integration was analyzed and tested with the use of a Logic analyzer. The data stream was generated and then we had to program the Zedboard. The below figure shows the result observed in the logic analyzer. As mentioned earlier, this proves that the spi_master, command list and i2s_master are working properly.



3.4.2.2. Challenges:

Writing a program that generates a LUT that stores the values of a sine wave of fixed frequency was the main challenge involving this module. The main challenge in this resulted from the choice of programming language and the missing knowledge on some concepts. We initially wanted to use C, since we already had some experience with it, but this idea soon became too complicated, because we realized C is not practical to generate a file with text and values from it. We then decided to switch to Python. Python provides an easy way to generate console outputs and is an intuitive programming language to get into due to relatively loose rules when it comes to variable declaration. Its drawbacks are a lack of the functionality of having variables saved as binary in two's complement. Additionally, Python in turn brought up the problem that variables have no fixed defined length, which made it very complicated to realize the two's complement calculation for negative values. We solved the problem using the format commands in Python and first a transformation to hex values and afterwards to binary values.
We implemented the LUT generator in a way that produces a list of case statements, that a state machine could go through and output the respective value when a given time in the case statements is reached. After seeing the final solution and concept of the sine generator, we realized these case statements are not necessary, which would possibly have made the use of C for programming possible. All in all, it should be noted that for future projects, the use of Matlab for this is recommended and will be used.

3.5. RISC-V

3.5.1. Preparation Sheet 5

3.5.1.1. Preparation 5.1: The RISC V Processor

In the next lab, we will replace the sine generator module we used for debugging with a proper RISC V processor. This will allow software on the processor to output audio using the ADAU. Later we will transfer this digital design including the processor to an ASIC. As we will use the PicoRV32 core (https://github.com/cliffordwolf/pi corv32), there are some things we have to look up first. As we want to implement a memory mapped device, we will have a look at the PicoRV32 memory interface on this preparation sheet.

a) Which two options are available for the memory interface? What's the difference between the interfaces?

The first option is to define precisely the type of memory we are going to use in this application (RAM, Flash, EEROM, DRAM, SRAM) and produce a specific interface that will work for only that type of memory.
Another approach is to consider that we will treat whatever type of memory we have as generic RAM internally, and to design a memory block that will interface to the actual memory—we will treat the memory interface as essentially a virtual RAM block.
For the initial design, therefore, we can treat the memory as a simple synchronous RAM block that has a clock, data bus, address bus, and write and read signals.[6]

Signal	Name	Direction	Type	Notes
Clock	mem_clk	out	reg	
Data bus	mem_data	inout	reg [31:0]	
Address bus	mem_addr	out	reg [31:0]	
Write	mem_nwr	out	reg	(active low)
Read	mem_nrd	out	reg	(active low)

The first option is to define a precise type of memory we are going to use.
The second option is to define a virtual RAM block that will interface to the actual memory, so that we don't necessarily need to know what type of memory we are going to use in advance.

[6] (Wilson, 2021)

c) We will use the simple interface of PicoRV32 which does not make a difference between system and peripheral buses. Name the signals which are used for this interface.

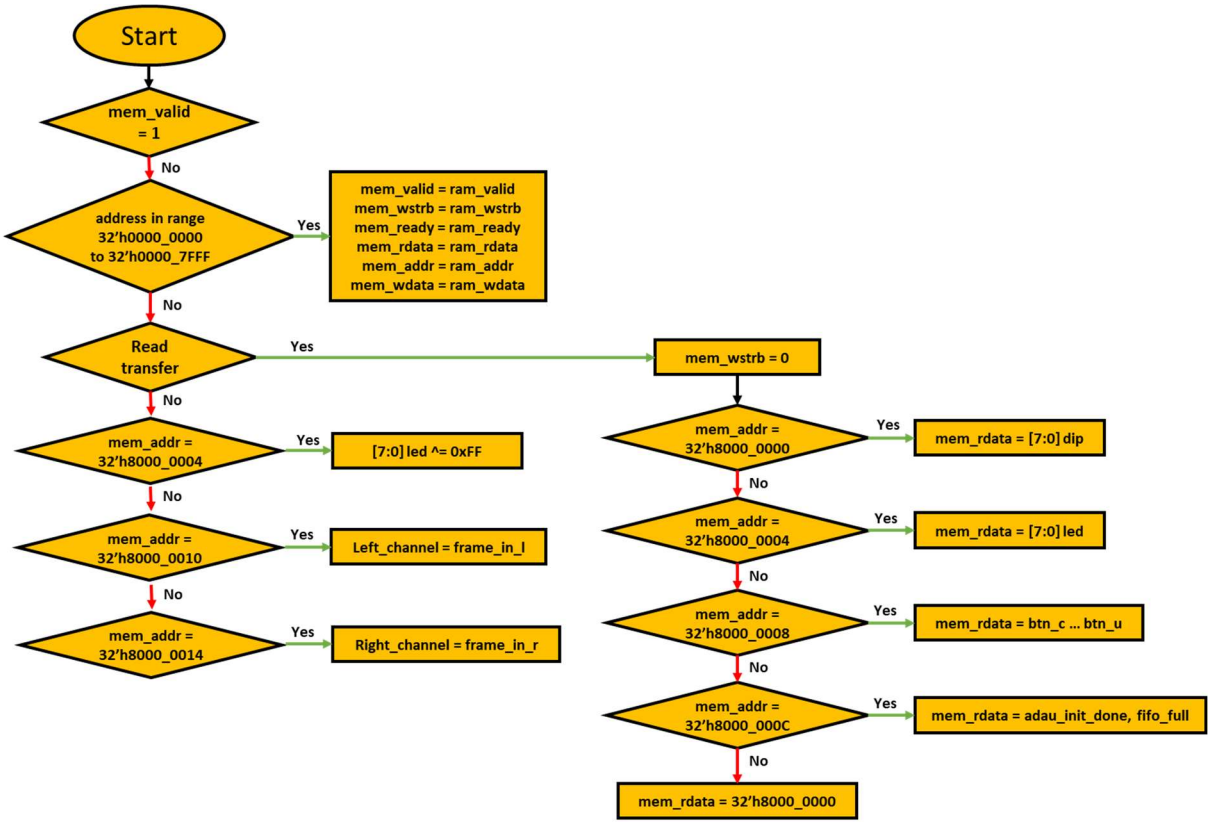
PicoRV32 Native Memory Interface

```
output        mem_valid
output        mem_instr
input         mem_ready

output [31:0] mem_addr
output [31:0] mem_wdata
output [ 3:0] mem_wstrb
input  [31:0] mem_rdata
```

[4]

3.5.1.2. Preparation 5.2: PicoRV32 Memory Interface

Draw a flow chart implementing the simple memory interface. Name the signals as in Preparation 5.1c and adhere to the following specifications:



3.5.2. Implemented solution

Now the sine generator module we used for debugging is replaced with a proper RISC V processor, that can access memory through the AXI bus to get audio data. This will allow software on the processor to output audio using the ADAU. This CPU is meant to be used as auxiliary processor in FPGA designs and ASICs. Due to its high fmax it can be integrated in most existing designs without crossing clock

domains. When operated on a lower frequency, it will have a lot of timing slack and thus can be added to a design without compromising timing closure. The core consists of 3 variations: picorv32, picorv32_axi and picorv32_wb. We will be using the picorv32 core and it provides a simple native memory interface that is easy to use in simple environments.

PicoRV32 Native Memory Interface: It is a simple valid-ready interface that one can run one memory transfer at a time.

```
output        mem_valid
output        mem_instr
input         mem_ready

output [31:0] mem_addr
output [31:0] mem_wdata
output [ 3:0] mem_wstrb
input  [31:0] mem_rdata
```

The core initiates a memory transfer by addressing mem_valid. The valid signal stays high until the peer asserts mem_ready. All core outputs are stable over the mem_valid period. If the memory transfer is an instruction fetch, the core asserts mem_instr.

Read Transfer:

In a read transfer mem_wstrb has the value 0 and mem_wdata is unused.
The memory reads the address mem_addr and makes the read value available on mem_rdata in the cycle mem_ready is high.
There is no need for an external wait cycle. The memory read can be implemented asynchronously with mem_ready going high in the same cycle as mem_valid, or mem_ready being tied to constant 1.

Write Transfer:

In a write transfer mem_wstrb is not 0 and mem_rdata is unused. The memory write the data at mem_wdata to the address mem_addr and acknowledges the transfer by asserting mem_ready.
The 4 bits of mem_wstrb are write enables for the four bytes in the addressed word. Only the 8 values 0000, 1111, 1100, 0011, 1000, 0100, 0010, and 0001 are possible, i.e. no write, write 32 bits, write upper 16 bits, write lower 16, or write a single byte respectively.
There is no need for an external wait cycle. The memory can acknowledge the write immediately with mem_ready going high in the same cycle as mem_valid, or mem_ready being tied to constant 1.

Upon finalizing the memory interface we use (which is shown in the flowchart from the preparation sheet)

We then finalize the address decoder in the cup_bus_logic.v

3.5.2.1. Code:

The cpu_bus_logic module interfaces to several peripheral modules, including external memory, where the audio data is stored, LEDs & switches for debugging and signals that connect to the I2S master for interfacing the ADAU.

```
module cpu_bus_logic(
      input clk,
      input reset,
      // CPU connections to memory
      input [31:0] addr,
      input [31:0] wdata,
      output reg [31:0] rdata,
      input [3:0] wstrb,
      input valid,
      output reg ready,
      // debugging stuff
      input [7:0] dip,
      input [4:0] buttons,
      output reg [7:0] led,
      // RAM interface
      output [14:0] ram_addr,
      output [31:0] ram_wdata,
      output reg ram_valid,
      output [3:0] ram_wstrb,
      input [31:0] ram_rdata,
      input ram_ready,
      // adau_interface signals
      output reg [23:0] adau_audio_l, adau_audio_r,
      output reg adau_audio_valid,
      input adau_audio_full,
      input adau_init_done
   );
```

To be able to access the RAM from the CPU, the memory signals are assigned.

```
//Connecting the RAM memory access to the cpu memory access
   assign ram_addr = addr[14:0];
   assign ram_wstrb = wstrb;
   assign ram_wdata = wdata;
```

The module is split into read and write operations. The read operations include RAM access, switch, button & LED status as well as ADAU status bits.

```
// read logic
   always @(*) begin
```

```
      ram_valid = 0;
      ready = 1;
      casez(addr)
         // 15 bit address / 256 kiBit
         // 0x0000_0000 - 0x0000_7FFF
         32'b0000_0000_0000_0000_0???_????_????_????: begin //address to be accessed is located in
the RAM
            rdata = ram_rdata;
            ram_valid = valid;
            ready = ram_ready;
         end
         32'h8000_0000: begin //read in data from the dip switches
            rdata = {24'b0, dip};
            ram_valid = 0;
            ready = 1;
         end
         32'h8000_0004: begin //read in LED status
            rdata = {24'b0,led};
            ram_valid = 0;
            ready = 1;
         end
         32'h8000_0008: begin //read in button status
            rdata = {24'b0,buttons};
            ram_valid = 0;
            ready = 1;
         end
         32'h8000_000c: begin //read in ADAU status
            rdata = {24'b0,adau_init_done, adau_audio_full};
            ram_valid = 0;
            ready = 1;
         end
         default: begin //default case for read operations - returns 0s
            rdata = 32'h0000_0000;
            ram_valid = 0;
            ready = 1;
         end
      endcase
   end
```

The write operations include setting status LEDs and more importantly writing the audio data to the respective channel on the I2S master that it directly interfaces to.

```
// write logic
   always @(posedge clk) begin
      if(reset) begin
         led <= 8'h00;
         adau_audio_l <= 24'h000000;   //initial write for left and right data
         adau_audio_r <= 24'h000000;
      end else begin
          if (!adau_audio_full && adau_audio_valid)
              adau_audio_valid <= 0;   //reset valid bit, if FIFO is full, no new data can be read
          if(valid) begin
             case(addr)
               32'h8000_0004: begin    //set the status of the LEDs
                   if(wstrb[0])
                      led <= wdata[7:0];
                end

               32'h8000_0010: begin    //write data to the left audio channel
                  if(wstrb[2] && wstrb[1] && wstrb[0])
                     adau_audio_l[23:0] <= wdata[23:0];     //Data from the left channel is written
```

```
to the FIFO
                end

                32'h8000_0014: begin
                   if(wstrb[2] && wstrb[1] && wstrb[0]) begin //write data to the right audio channel
                      adau_audio_valid <= 1;                  //valid for write enable in FIFO,
should only be set once until FIFO is full
                      adau_audio_r[23:0] <= wdata[23:0];      //Data from the right channel is
written to the FIFO
                   end
                end

             endcase
          end
       end
    end
endmodule
```

Upon deciding the address decode we then integrate the i2s_master, adau_command_list and adau_spi_master modules in the fpga_riscv_top.v file.

```
//command list instantiation
    adau_command_list ctrl(
        .clk(clk_soc),
        .reset(reset),
        .spi_ready(spi_ready),
        .adau_init_done(adau_init_done),
        .command(adau_command),
        .command_valid(adau_command_valid)
    );
    //SPI master instantiation
    adau_spi_master spi(
        .reset(reset),
        .clk(clk_soc),
        .data_in(adau_command),
        .ready(spi_ready),
        .valid(adau_command_valid),
        .cdata(ac_addr1_cdata),
        .cclk(ac_scl_cclk),
        .clatch_n(ac_addr0_clatch)
    );

    // sin <=> i2s
    wire [23:0] adau_audio_in_l, adau_audio_in_r;
    wire adau_audio_in_valid, adau_audio_full;
    //I2S master instantiation
    i2s_master i2s(
        .clk_soc(clk_soc),
        .ac_mclk(ac_mclk),
        .reset(reset),
        .bclk(ac_bclk),
        .lrclk(ac_lrclk),
        .sdata(ac_dac_sdata),
        .frame_in_l(adau_audio_in_l),
        .frame_in_r(adau_audio_in_r),
        .full(adau_audio_full),
        .write_frame(adau_audio_in_valid)
    );
```

4. Completed Tasks - Analog Design

4.1. Differential Amplifier

4.1.1. Preparation Sheet 6

4.1.1.1. Preparation 6.1: MOSFET Devices

e) Which geometric parameters affect the electrical characteristics? Explain how these parameters affect the characteristics.

The drain source current is described by the following equation: $Ids = \mu Cox\,(W/L)\,(Vgs - Vth)\,Vds$ [2] where W is the width of the depletion channel whereas L is the length of the channel. As it can be seen in the equation, an increase in channel width leads to an increase in drain source current. An increase in channel length leads to a decrease in drain source current.

4.1.1.2. Preparation 6.2: Analog Amplifier Circuits

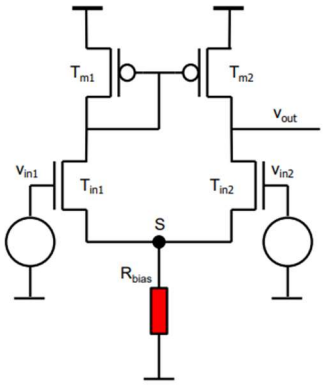
In this exercise, we'll revisit various MOSFET based amplifier circuits. We first have a look at single ended amplifiers. For background information, refer to the DAS lecture:

c) Give formulas for the voltage gain, input impedance and output resistance for both circuits.

common source amplifier:
Voltage Gain: -Gm*r
Input Impedance: 1/jωC
output Impedance: Rds_in ll Rds_load

Source follower:
Voltage gain = 1
input impedance = infinite
output resistance = 1/Gm_in

d) Draw schematics for a basic differential amplifier with two inputs and one output.



e) Shortly describe the working principle of this differential amplifier.

*The current mirror keeps the same current in both branches
*These transistors operate in linear mode and Vin1 and Vin2 control the drain source voltage drop

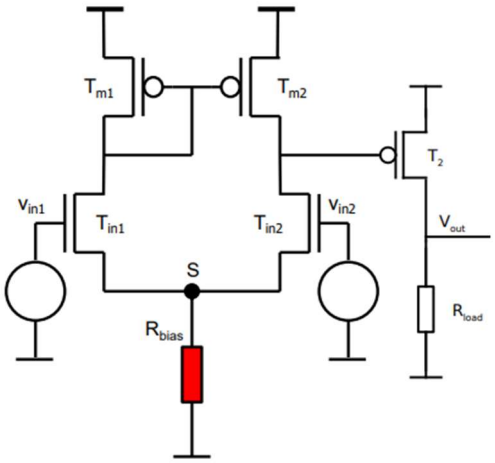
f) What are the advantages compared to amplifiers with single-ended inputs?

Differential amplifiers offer many advantages for manipulating differential signals. They provide immunity to external noise; a 6-dB increase in dynamic range, which is a clear advantage for low-voltage systems; and reduced second-order harmonics. Integrated fully differential amplifiers are well-suited for driving differential ADC inputs and differential transmission lines. They provide an easy means of anti alias filtering, and a dedicated input easily sets the required common-mode voltage. These amplifiers are also well-suited for driving differential transmission lines, and active termination provides for increased efficiency. You can easily adapt inverting-amplifier topologies to fully differential amplifiers by implementing two symmetric feedback paths.[7]

The simple differential amplifier has a high output impedance. We now add a second stage to remedy this.

g) In schematic for Preparation 6.2d, add the second stage to achieve low output impedance. The amplifier should have differential inputs, a single ended output.

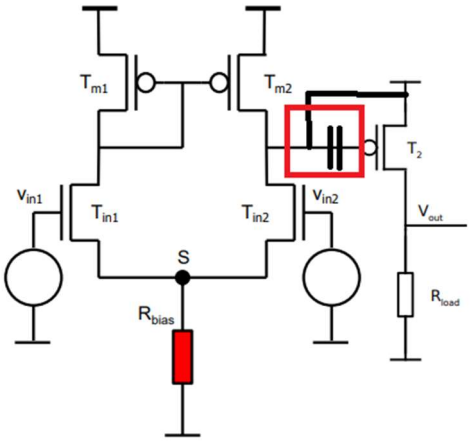
[7] (Karki, 2021)



h) Give formulas for gain, input impedance and output resistance for this amplifier.

Gain: Gm_in*Rds_in*Rload
input Impedance: 1/(jωCgs/2)
Output Resistance: Rload

j) In order to make the simple two-stage amplifier stable, we have to add an compensation capacitance. Add it to your solution for Preparation 6.2d.



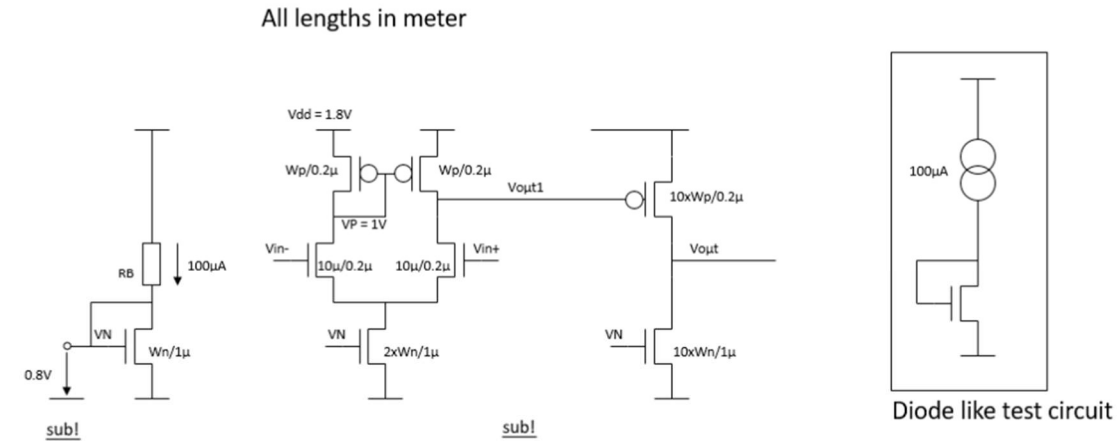
4.1.1.3. Preparation 6.3: Digital to Analog Converter Circuits

e) Why do we need an amplifier for the output of the DAC if we want to attach headphones? Note: We don't need the amplifier for gain here, the open-load output voltage of the DAC is sufficient.

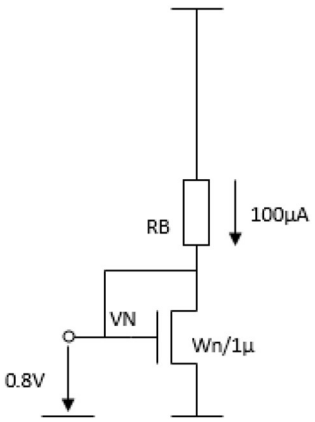
The purpose of a DAC is solely to convert digital signals into analog waveforms. When the conversion has been made, the audio signal is too weak to be received by the sound source. Therefore, **an amplifier is needed to boost the signal to an optimal level.**[8]

4.1.2. Implemented solution

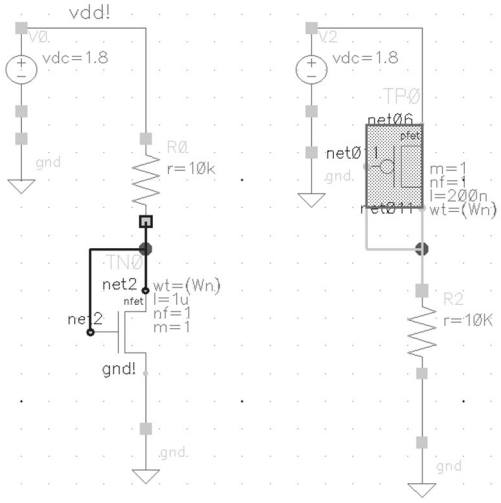
4.1.2.1. Lab task 6.1: Design of a two stage differential amplifier



a) To calculate the value of the resistor RB it is important to determine the voltage drop across the resistor. This Voltage is 1V due to VN being 0.8V and the supply voltage being 1.8V. With ohms law one can determine the resistance to be 10kOhm with the given current of 100uA.

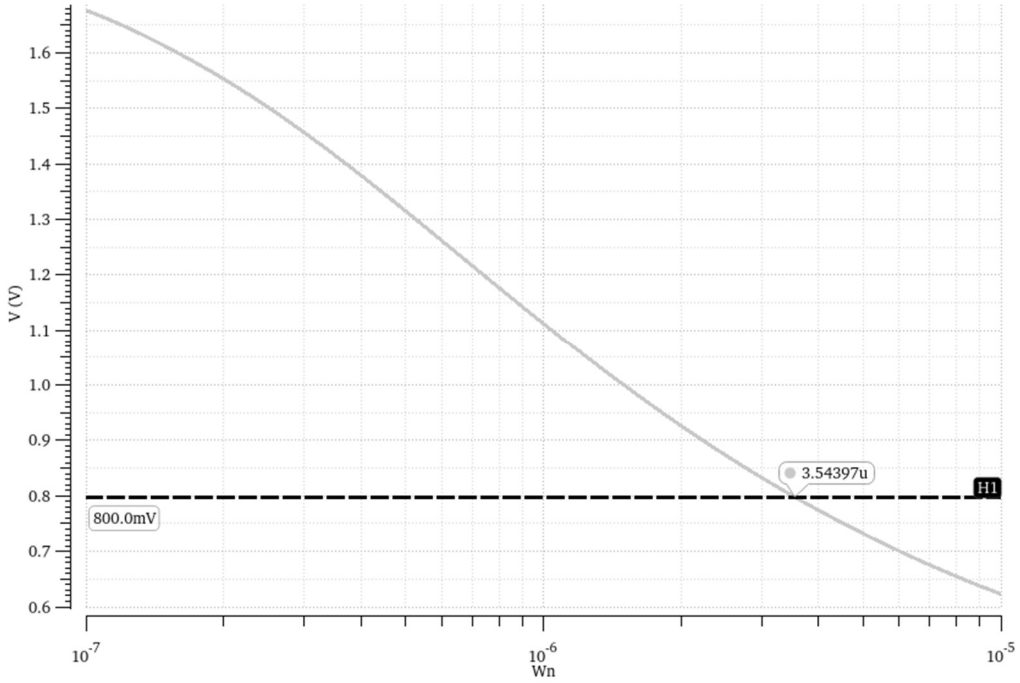


[8] (Stamp Sound, 2022)

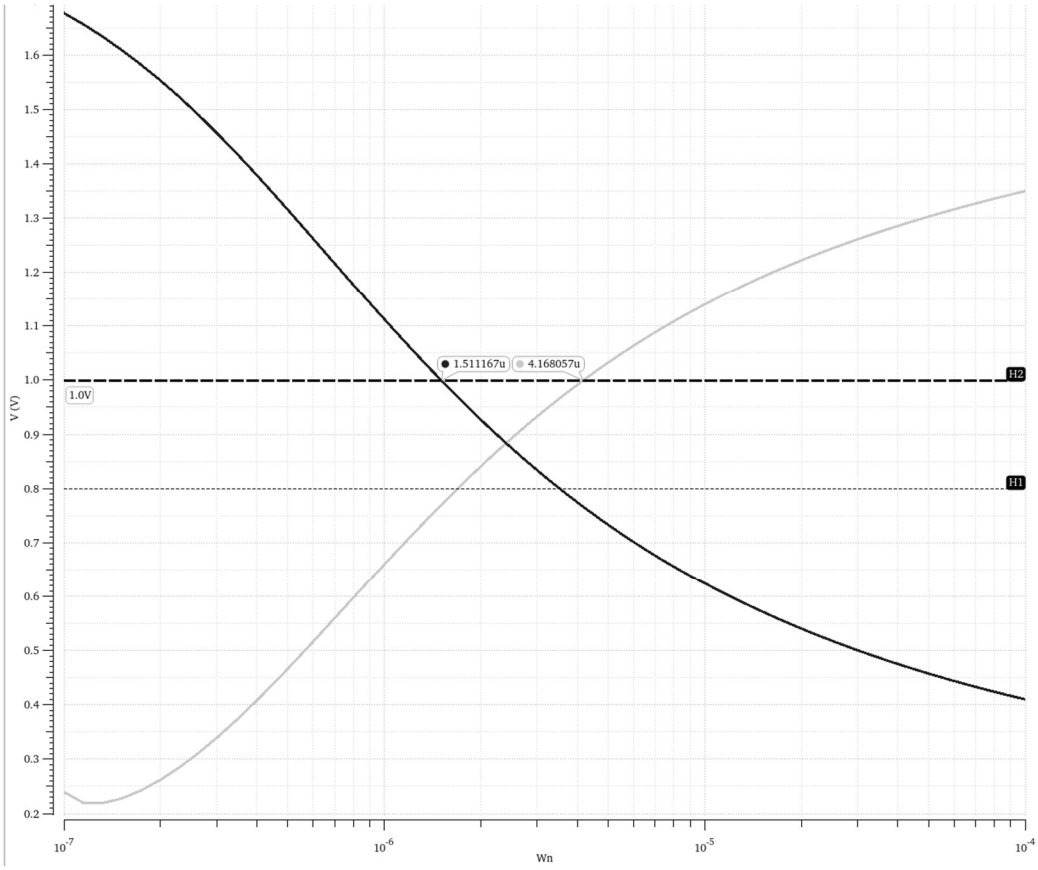


b) - f)
To determine the transistor width the two circuits to the right were used for testing. The dc simulation was used to find the exact width where the voltage drop over the NMOS is 0.8V and the PMOS is 1.0V.

The width for the NMOS is 3.54um:



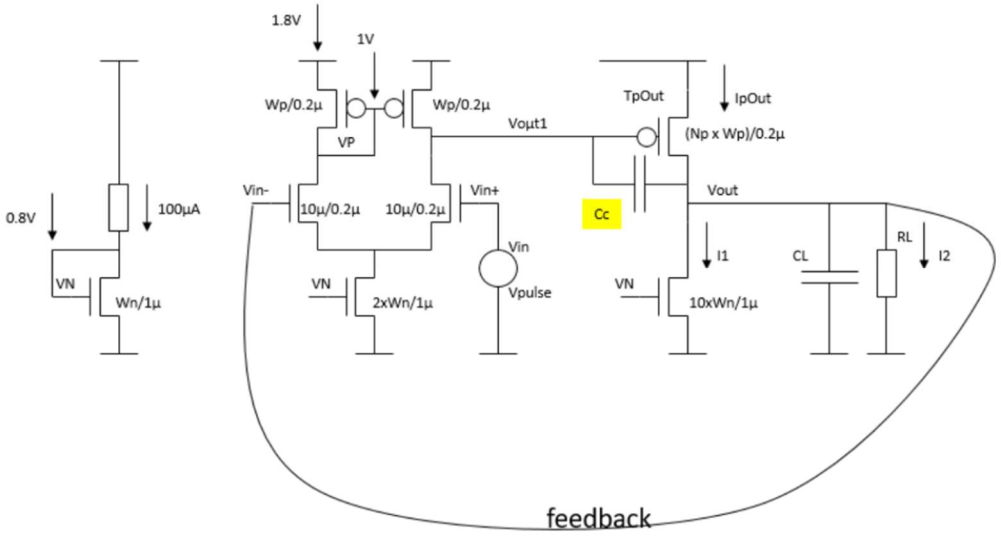
The width for the PMOS is 4.19um:



4.2. Differential Amplifier & Stability Analysis

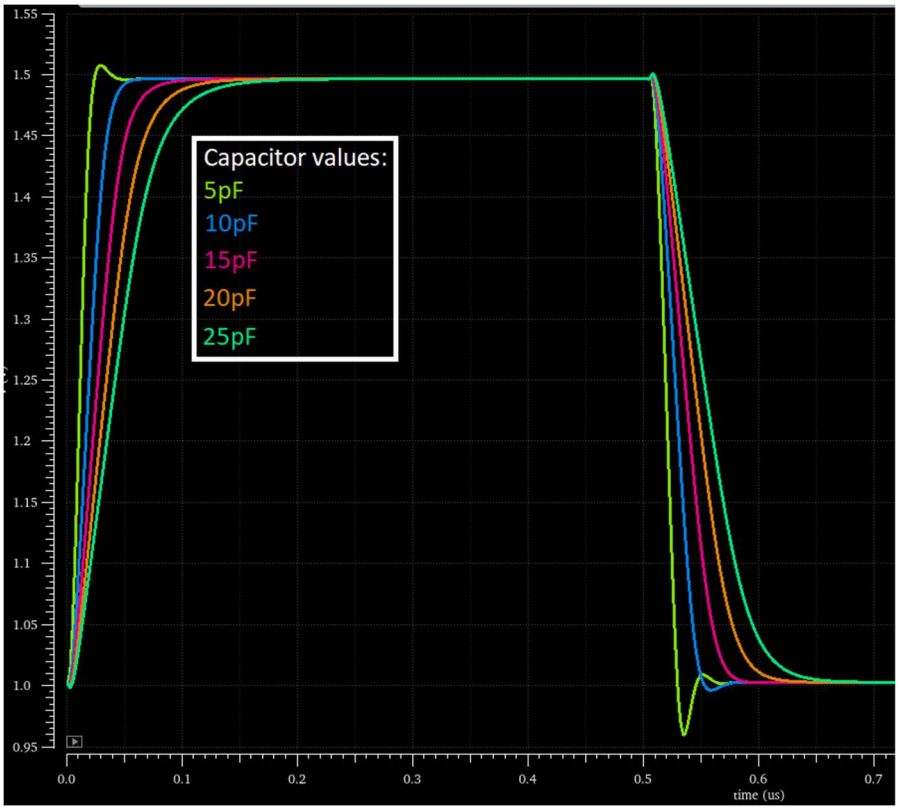
4.2.1. Implemented solution

4.2.1.1. Simulation:



[9]

Using a square signal on the input to determine the right value for the capacitance Cc (5 pF, 10 pF, 15 pF, 20 pF) to get rid of voltage overshooting the targeted output voltage. As seen below, a capacitance of 5 pF is too low and the output voltage is still showing over- and undershooting of the values at the positive and negative signal edge. A value of 10 pF is an improvement, but it still undershoots the desired value at the negative edge. A capacitance of 15 pF is the first value that tracks the desired voltages perfectly. Larger values shouldn't be used due to an increase in the delay until the stable output value is reached.

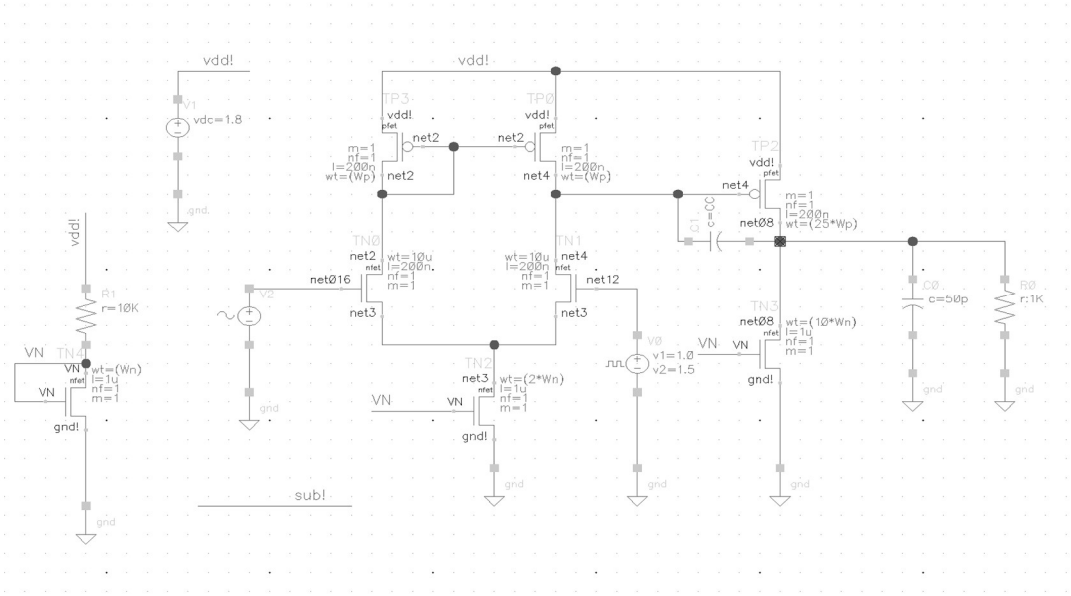


[9] (KIT - Karlsruher Institut für Technologie, 2022)

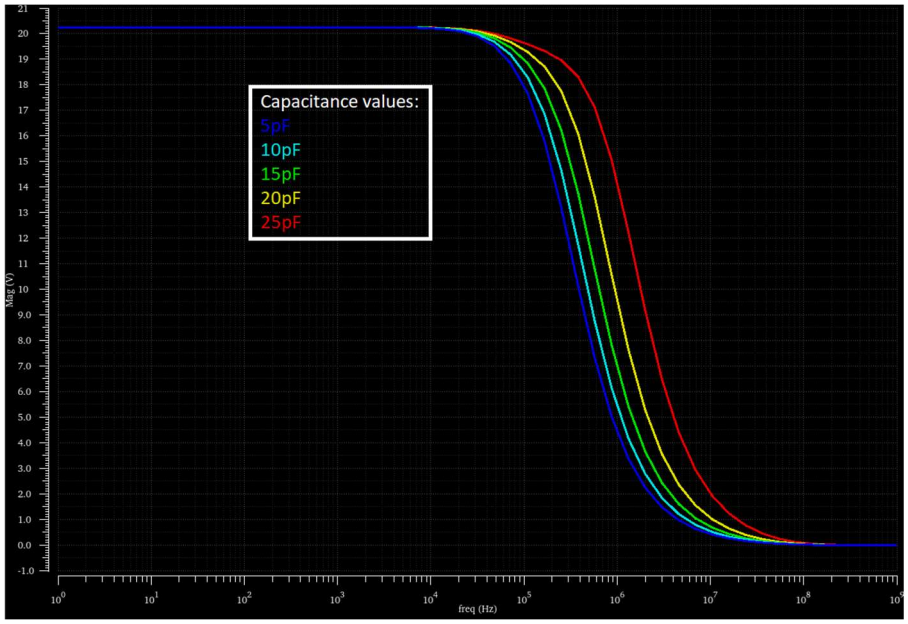
Output signal with different capacitance Cc values (5 pF, 10 pF, 15 pF, 20 pF)

4.2.1.2. Lab Exercise 7.4 Stability Analysis:

For the next part the feedback loop is cut and a voltage source Vsin is added.



The AC Magnitude is shown in the graph below with its corresponding capacitance values for frequency values of 1 Hz to 1 GHz:



The calculated phase margins are shown in the table below:

Capacitance (pF)	Phase Margin (deg)
5	-112.9
10	-104.3
15	-100.3
20	-98.07

4.3. DAC Integration

4.3.1. Preparation Sheet 7

4.3.1.1. Preparation 7.2: ASIC DAC Interface

Thermometer code resembles the output produced by a thermometer. In thermometer code, a value representing number 'N' has the lower most 'N' bits as '1'; others as 0. So, to move from N to 'N + 1', just change the rightmost '0' to '1'. Figure 1 below graphically shows the thermometer codes for values from '0' to '7'. As is evident, each value resembles a reading in the thermometer. This is how the thermometer code got its name. Flash ADCs, time-to-digital converters (TDC) are some of the circuits that utilize thermometer code.(https://vlsiuniverse.blogspot.com/2016/06/binary-to-thermometer-encoder.html)

Characteristics of thermometer code:
1.Each symbol in thermometer code is a sequence of 0s followed by a sequence of 1s
2.There cannot be 0s in-between two 1s. For example, a symbol 01011 is invalid in thermometer code
3. For an n-bit binary code, the corresponding thermometer code will have 2^n - 1 symbols; hence, as many bits will be needed to represent the thermometer code for the same.

This version below is for the unknown value of N

```
module therm(
input wire[7:0] In,
output wire[255:0] result
);

assign result[254:0] = (255'b1 << In) - 255'1;

endmodule


module btothermo (
    input [n:0] binary;
    output [2^n-1:0] thermometer
    );

parameter n;
reg [2^n-1:0] counter;
integer i;

always @ (binary) begin
    for (i = 1; i <= binary ; i = i + 1)
        counter = ((2^n-1)b'1 << 1)     //shift to right for 1 bit
        if(i = binary)
            thermometer = counter - 1;  //100 - 1 = 011
    end
end
```

```
endmodule
```

d)

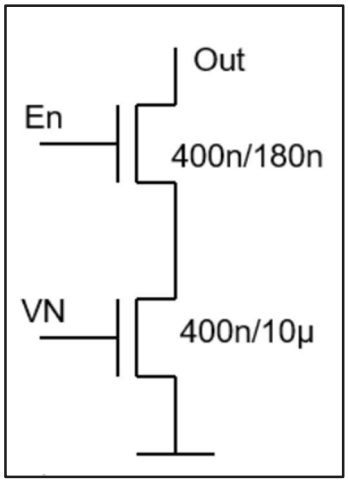
```
module upordown_counter(
    Clk,
    reset,
    N1, //upper limit
    N2, //lower limit
    Count
    );

    input Clk,reset,N1,N2;
    output Count;
    //Internal variables
    reg [3 : 0] Count = 0;
    reg up = 0;

    always @(posedge(Clk) or posedge(reset))
    begin
        if(reset == 1) begin
            Count <= 0;
            up <= 1; //will count up after reset
        else
            if(Count < N1 && up == 1) begin
                Count = Count + 1;
            else if (Count == N1 && up == 1)
                up <= 0; //will count down
            else if(Count > N2 && up == 0)
                Count = Count - 1;
                    //Count <= 0;
            else if (Count == N2 && up == 0)
                up <= 1;
            end
        end
    end
endmodule
```

4.3.2. Implemented solution

With the knowledge of thermometer code an 8-bit DAC can be designed by converting the 8 bit value to the respective 256 digit thermometer value. Each bit of this thermometer value is then used to trigger the “enable” signal of a current source. All of the 256 current sources are supplied with the same current through a current mirror. The current sources are joined together and fed into a two-stage differential amplifier that amplifies the signal in a way so that it can be played on a headphone.

Current sources

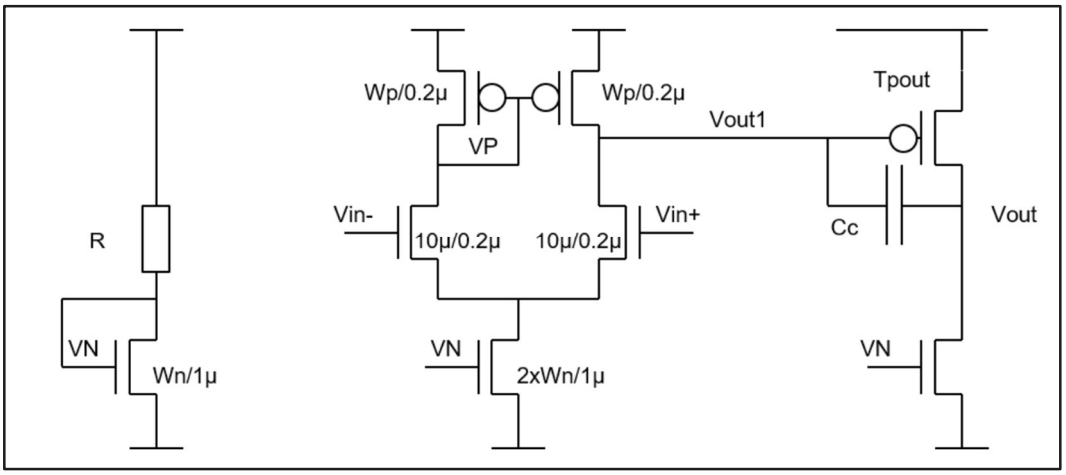
The DAC current sources are simply two n-fet transistors in series. The lower transistor is part of a current mirror, which provides the stable current for each DAC stage. The upper transistor is the “enable” signal, which controls if the respective current source is activated or not. This enable signal is provided by the thermometer code generator, which is fed with the audio data.



Schematic of one of the DAC current sources[10]

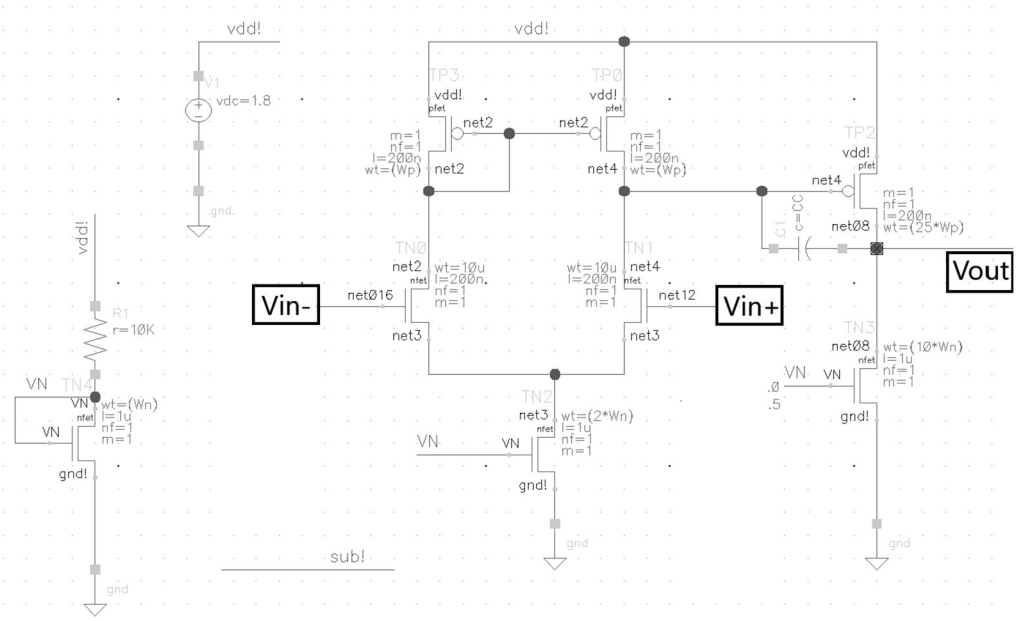
Amplifier

For this use, an asymmetric differential amplifier designed in the step before is used. The output of the differential amplifier is connected to a common source amplifier, similar to the use in an LDO. The “cc” Capacitor is used to stabilize the frequency response, as described before.



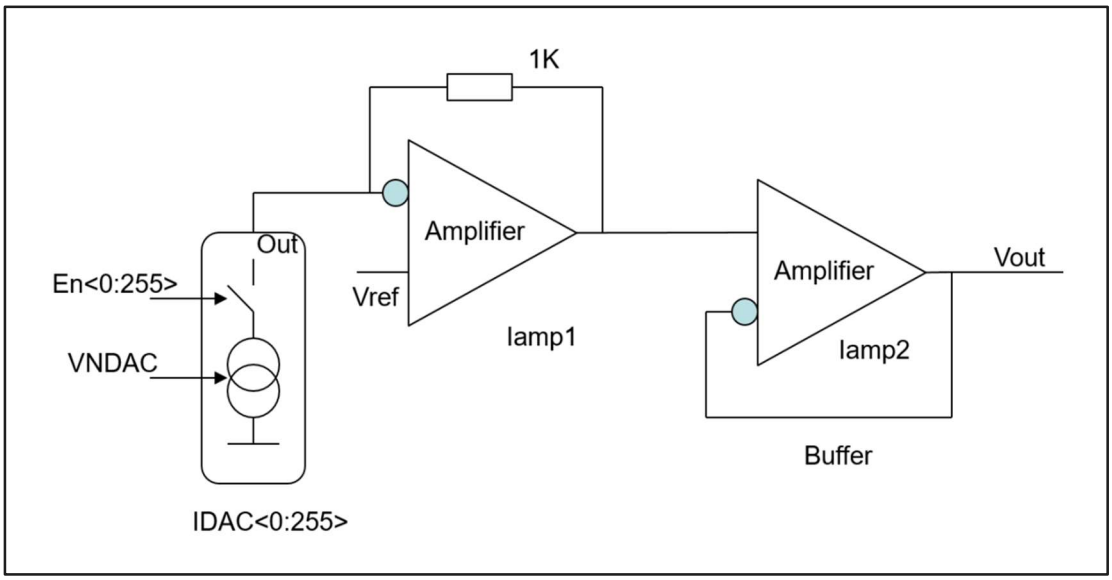
Schematic of the differential amplifier[11]

[10] (KIT - Karlsruher Institut für Technologie, 2022)
[11] (KIT - Karlsruher Institut für Technologie, 2022)



Final Design

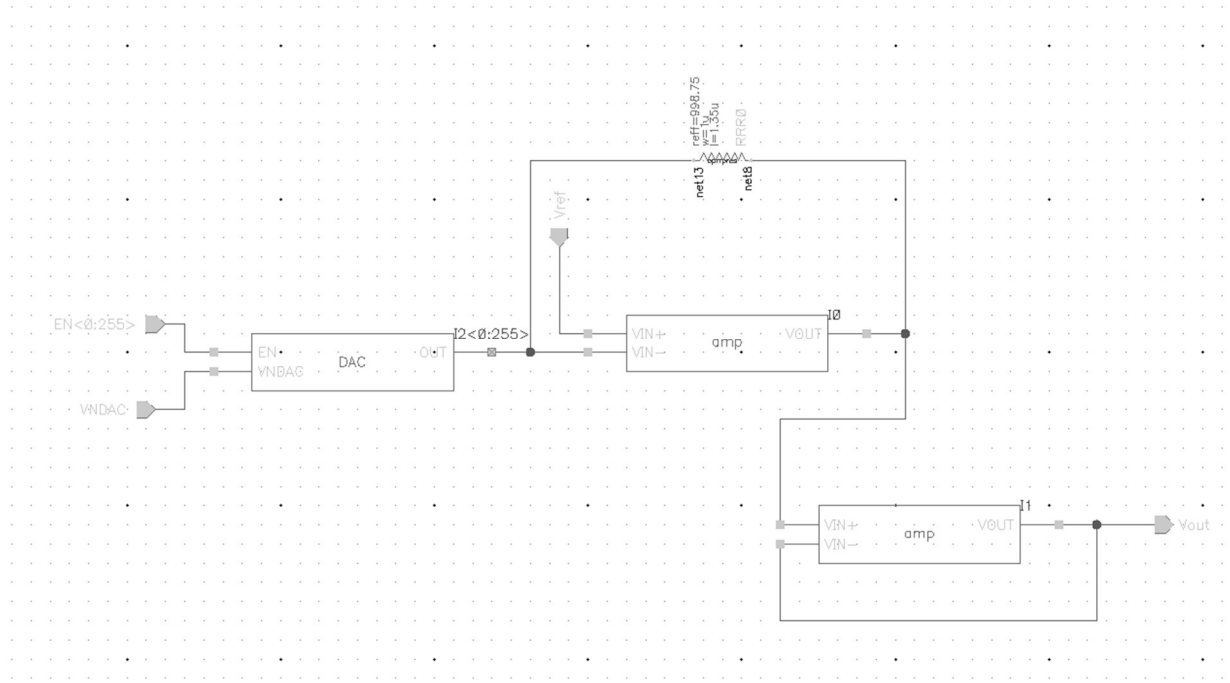
The connected components are connected to each other as follows. The first amplifier serves as an I-U-transformer, the second one amplifies the output voltage of the first one.



Schematic of the complete DAC with 256 current sources[12]

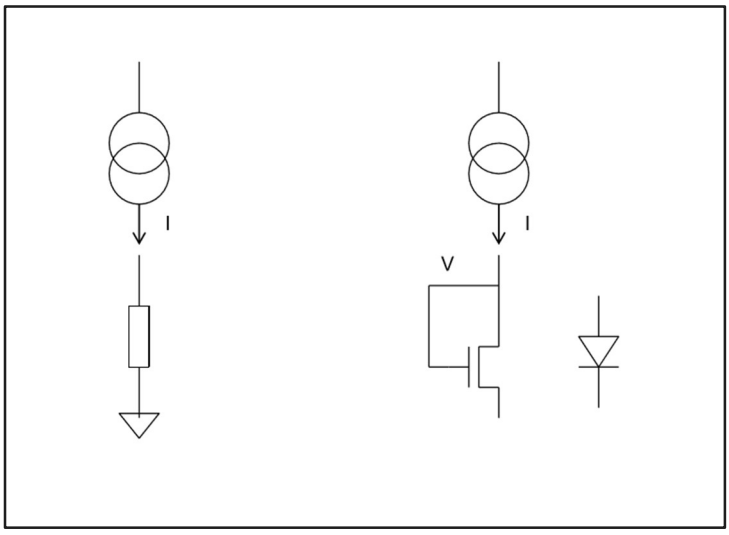
To simplify the design of this DAC, a separate symbol for the amplifiers and current sources is created. The final DAC circuit can be seen in the following picture.

[12] (KIT - Karlsruher Institut für Technologie, 2022)



Full DAC in Virtuoso

The resistors and capacitors used in this design are implemented on-chip and are not additional parts that will be mounted on a board. This is crucial to meet space requirements of implemented circuits used in a lot of modern devices, where a DAC is needed, such as phones, laptops and especially Bluetooth headphones. Even though on chip resistors save space on the board they need an over proportional amount of space on the silicon, which is why often resistors are exchanged with smaller transistors in combination with a current source that can mimic the functionality of a resistor.



Replacement of a resistor through a transistor[13]

[13] (Peric, 2021), MOSFET Diodes

4.4. Thermometer encoder

4.4.1. Preparation Sheet 8

4.4.1.1. Preparation 8.2: Circuit Setup and Hold Time Analysis

a) Setup & Hold conditions

The following propagation delays apply:

$$T_{D2} = T_{outdelay} + T_{CL}$$
$$T_{clk2} = T_{clk1} + T_B$$

The hold condition for this case is defined by:

Hold Slack = $S_H > 0$

$$S_H = T_{D2} - T_{hold} - T_{clk2} + T_{clk1} > 0$$
$$S_H = (T_{outdelay} + T_{CL)} - T_{hold} - (T_{clk1} + T_{B)} + T_{clk1} > 0$$
$$S_H = T_{outdelay} + T_{CL} - T_{hol} - T_B > 0$$

The setup conditon for this case is defined by:

Setup Slack = $S_S > 0$

$$S_S = T_{clk2} - T_{setup} - T_{D2} + (\frac{1}{f_{clk}}) - T_{clk1} > 0$$
$$S_S = (T_{clk1} + T_{B)} - T_{setup} - (T_{outdelay} + T_{CL}) + (\frac{1}{f_{clk}}) - T_{clk1} > 0$$
$$S_S = T_B - T_{setup} - T_{outdelay} - T_{CL} + (\frac{1}{f_{clk}}) > 0$$

4.4.2. Implemented solution

4.4.2.1. Implemented Code:

First the binary counter is implemented. This function lets the result count up and down with an upper and lower boundary. The changes we have made to the implementation is that we have set the input upper and lower boundary as a static parameter, so that we don't have to define it somewhere later. The reason the upper and lower boundary is set as parameter but not as register will be later explained in the challenge of this part.

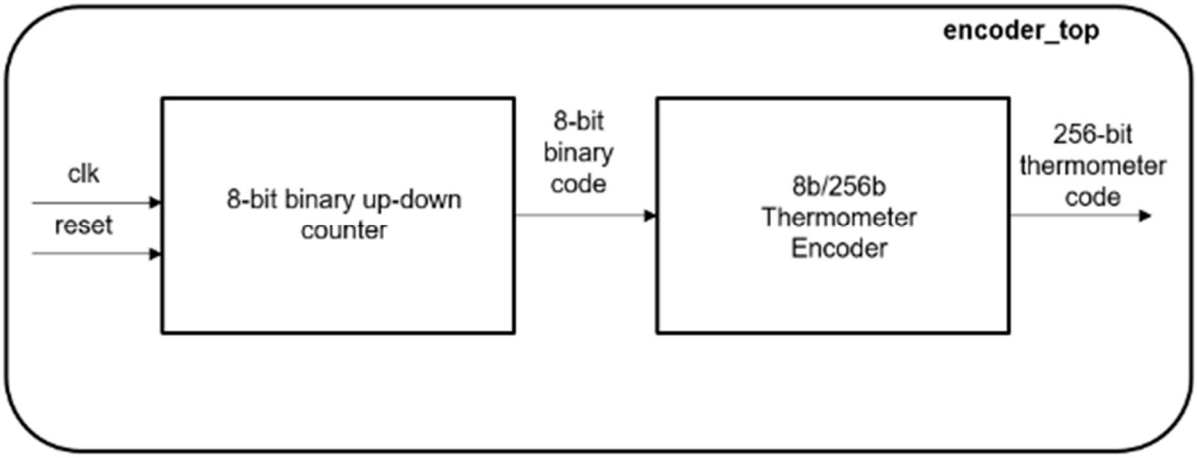
It is important to only use posedge clock as an income signal instead of clock and reset together. The reason will be later discussed in the challenge.

```
module binary_counter(
   input wire reset,
   input wire clock,
   output reg [7:0] result
);
   // instead of "reg" bc that lead to problems during gtl simulation
   parameter min = 0;
   parameter max = 200;
   reg updown; //up = 1 ; down = 0
always @(posedge clock) begin
   if(reset) begin
       result <= 0;
       updown <= 1;
   end
   else begin
       if(result == (min+1) && updown == 0) begin
           updown <= 1;
       end
       else if(result == (max-1) && updown == 1) begin
           updown <= 0;
       end

   if(updown) begin
           result <= result + 8'b1;
   end else
           result <= result - 8'b1;
   end
end
endmodule
```

Then the thermometer encoder will be implemented to convert the binary code to an thermometer counter. The wire here is defined as 255 in total, although later in the

```
`timescale 1ns/1ps
module encoder_sim ();
   // Write your testbench here!
   reg clk;
   reg reset;
   wire [254:0] thermometer_code ; //value

  encoder_top uut_0(
   //Inputs
   .reset      (reset),
   .clk        (clk),
   .thermometer_code (thermometer_code)
   );
   initial clk <= 0;
   always #200 clk <= ~clk;
   initial begin
       // resetting
       reset <= 1;
       repeat(2) begin
           @(posedge clk);
           end
           reset <= 0;
   end
endmodule
```

The whole DAC block should look like that and it is necessary to bind the both module from binary up down counter and Thermometer encoder together in one module to an encoder top module. The block looks like in the picture below. The implemented instantiation is also shown below. It is important to set the output thermometer encoder of the top also as 254 outputs so that the module can match to each other.

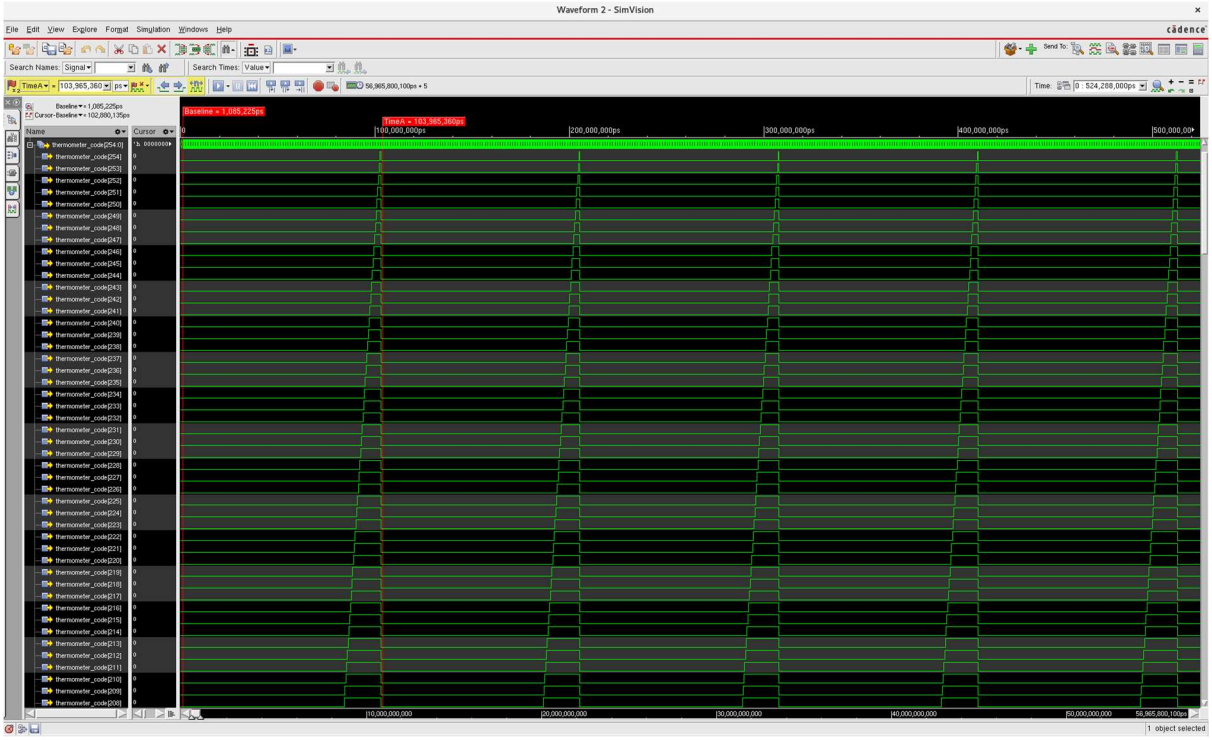


Structure of the thermometer encoder[14]

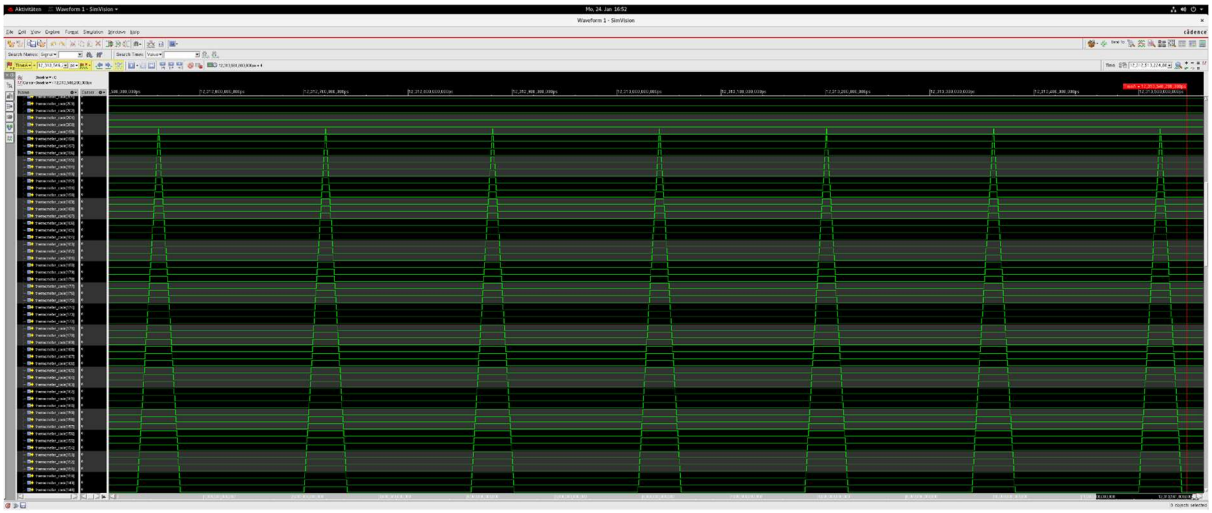
```
module encoder_top(
   input clk,
   input reset,
   output [254:0] thermometer_code
);
wire [7:0] bin_value;
binary_counter binary_counter(
   .reset(reset),
   .clock(clk),
   .result(bin_value)
);
thermometer_encoder therm_encode(
   .value(bin_value),
   .result(thermometer_code)
);
endmodule
```

Now we try to simulate the implemented code using Cadence XCELIUM simulation suite. First we make a RTL simulation folder and run the simulation under that folder with the *xrun* command. As we have not implemented the $stop(); command in our code, we need to call and use -gui to stop the simulation. The GUI is also used as a window to read the simulation result. The simulation is succeeded at the Register-Transfer-Level.
The picture below is the wrong result before we change the up and down boundary from a reg definition to a parameter definition. The counter continues to count up the highest amount and then reset, which is not our aim at all.

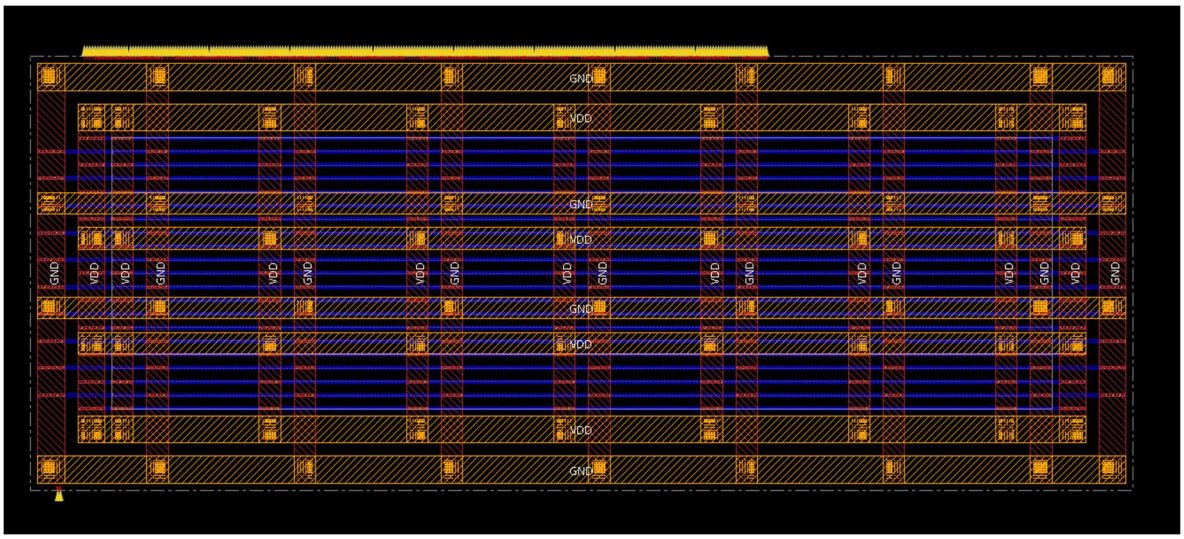
[14] (KIT - Karlsruher Institut für Technologie, 2022)



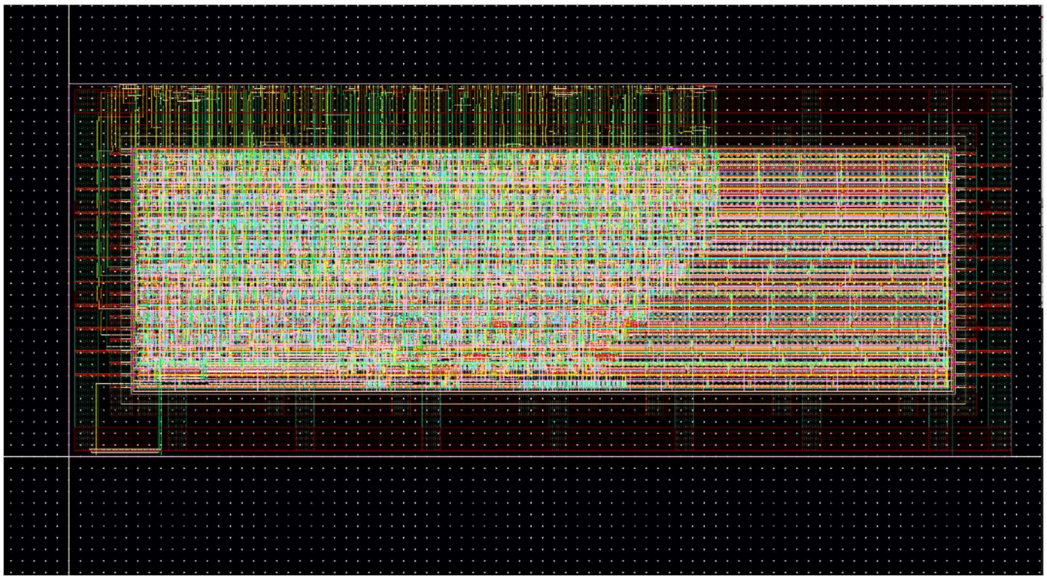
After we fixed this problem and changed the up-down boundary definition to parameter, the counter works properly with the count up and down function. In the picture below we can see the triangle form that indicates the counter is working properly. In addition the counter only counts up to the up boundary 200, which we defined before in our code.



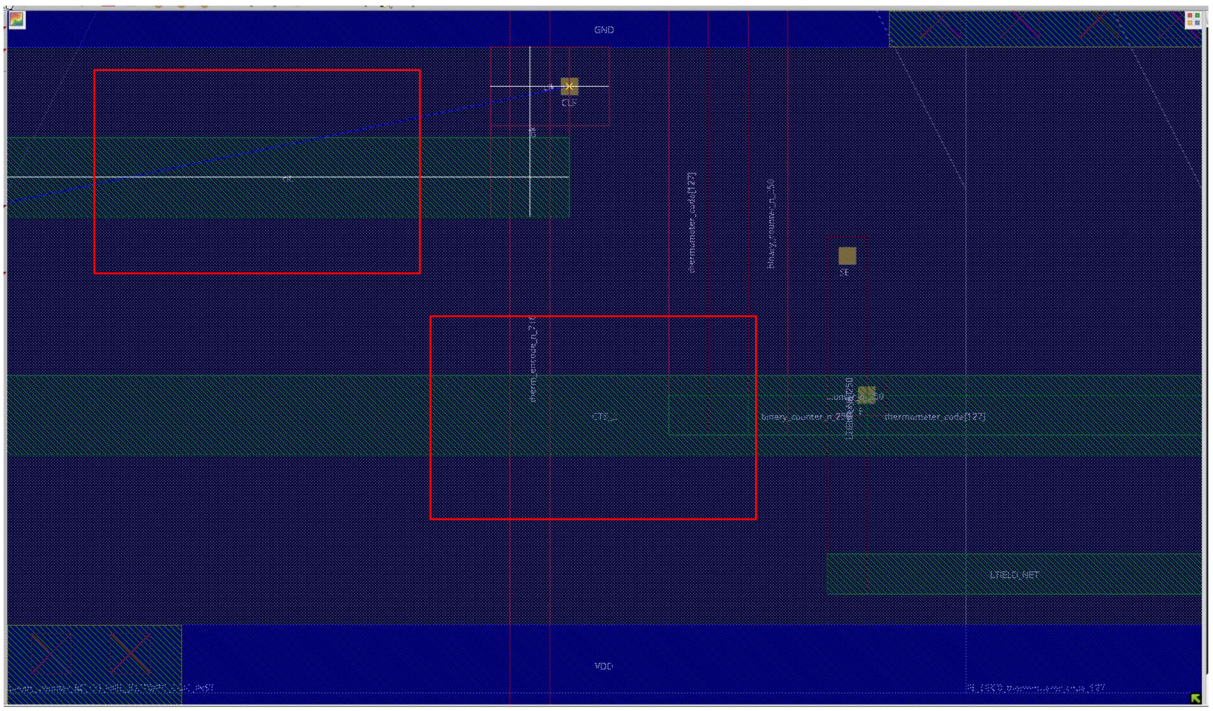
Then we created a synthesis folder to run the synthesis by invoking Genus using the command psoc_synthesis. Then we run a simulation in GTL (Gate-Transistor-Level). After that we run through the Floor Planning process. There is another problem we have encountered when we use the command flow_sub_plan. The problem is related to a synchronous rest, which will be later discussed in the challenge part. The screenshot below shows the result of floorplanning.



Then we go through the placement and routing using the command flow_sub_place. The result is shown below.

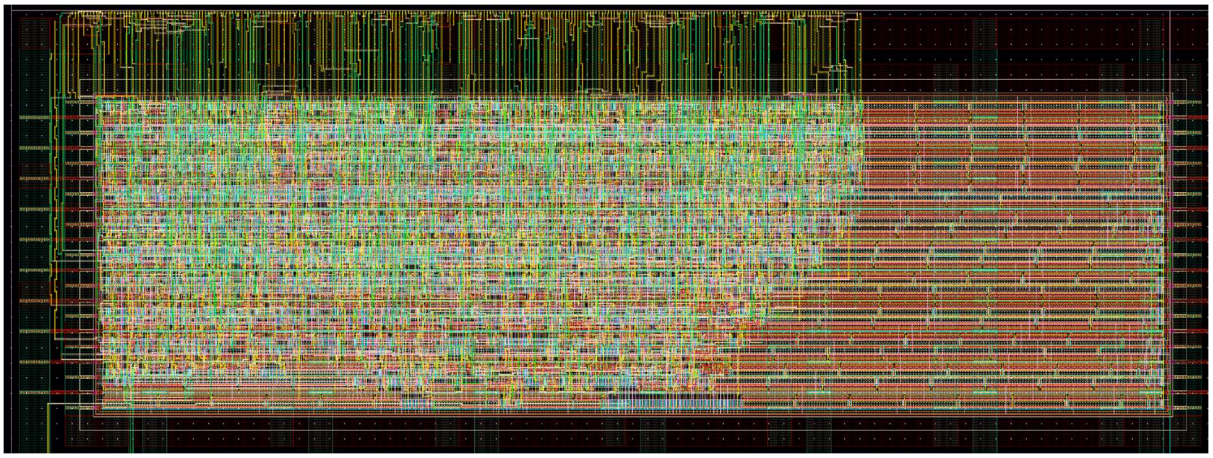


After that we go through the clock tree synthesis using the command flow_sub_cts. As we can see in the picture below, the clock and clock tree wire are much thicker than the other wire for regular signals because the scripts set a routing constraint. The clock wire and the clock tree wire are shown in the red block below.



The next step of this routing using the command flow_sub_route. Finally we use the flow_sub_streamout to finalize the design.
All the steps above can be run through using a single command flow_signoff.

The last part we import the whole physical design into virtuoso, which is shown in the screenshot below.



4.4.2.2. Challenges:

Defining UP-Down boundary as parameter

We observed some differences in Pre & Post Synthesis Simulation.
Problem occurred when setting the min and max counting values through:

```
Reg [7:0] min = 0;
```

```
Reg [7:0] max = 200;
```

It can't be synthesized, because registers don't represent a constant in that case, which is why this reg definition will be ignored during synthesis. That is why the counter in the post-synthesis simulation always counts up to the maximum value, which in this case is an 8 bit Wire and then overflows.

Solution:

```
Parameter min = 0;
Parameter max = 200;
```

From this challenge I have learned to think about the physical layer simultaneously when implementing the code in Verilog. In this case it is not possible to use a physical register to compare to a digital number. If we define two 8-bit registers, then in the physical layer there will be literally 8-bit physical parts of registers implemented, which make no sense. And not even to say comparing the values from the 8 bits register to a digital value.

Using the synchronous reset instead of asynchronous in Up-Down-counter

It is important that we use synchronous reset in implementation. There have been errors that occurred to us later in the floorplanning using the command flow_sub_plan in innovus that only 55% of the network can be placed. After asking the tutor, we are told to implement a synchronous reset to solve this problem.

5. General Challenges

5.1.1. Teamwork

Teamwork is an often-overlooked part of the workflow and solution finding within a given Problem. The general idea is efficiency, where different people in the group have different talents or are interested in different parts of workflow. For that reason, the group members were chosen based on a personality test to create heterogeneous groups with people who have complementing strengths. The underlying principle is the maximization of work efficiency. To achieve that the work has to be partitioned and given to the person who has the best set of skills to solve the problem. This could be done by openly talking about the strengths and weaknesses in the beginning of the internship. Because of that we could efficiently allocate the tasks either to individual people or two pairs of two, who have the complementary skill necessary. A good example would be the testbench that takes a closer look at the handshake between the spi_master and the adau_command_list. Both have been worked on by different individuals but the process of writing the combined testbench was sped up greatly due to both members knowing their own part of the task.

5.1.2. Transfer of knowledge to actual work

Learning a programming language can really only be done when programming with it. The same applies for example for the design and dimensioning of transistors and analog circuits. Although there was already a broad knowledge about these topics from previous studies, applying the knowledge turned out to be a challenge sometimes.
The basic concepts of state machines and the syntax of Verilog was known but when implementing it there arose some new uncertainties, e.g., on which clock edge does a state really change or how to create a counter that switched a signal always to the right time. Even though the basics to this were clear in theory, applying it was different and resulted in a way better understanding.
Another example for the difference between theory and reality was when determining the stability of an amplifier. We were familiar with the idea of bode plots and phase margin. But determining it for a larger circuit helped to gain a better understanding of it. Through the simulation, it was possible to see the impact of even a small coupling capacitor on the whole system stability.
This goes to show the importance of applying theoretical knowledge to real world problems to fully understand the concept.

6. Personal summary

I have learned a lot through practically implementing the code using Verilog and do the simulation Vivado in this Lab. Especially the debugging part with the use of logic analyser. Although the simulation in the Vivado might have worked perfectly fine, it could still be errors in the real signal transformations. The implementing of the state machine according to state chart is also something that has been taught, but never practiced by hand. By using the state machine, sometimes there might be lags between the state change or some signals are left undefined during the state change, or the sequence of the deciding part is differently implemented. Those could all be the reason that the simulation part in Vivado is fine, but the physical debugging signal read by the logic analyser is incorrect or missing. But after going through all the process of implementation and debugging, it gives me some more experience in solving the problems.

Also, the combination of this lab with other courses are great. In the analogue design part, I've learned about the structure of CMOS and how the output voltage can be influenced by the width of the CMOS. It is a good chance to review the theoretical knowledges I've learned. After running the simulation of the electrical circuit in virtuoso, I got a better understanding of the knowledge taught in DAS lecture.
It is very interesting to go through the whole chip design process using inovus and Genus. By following the instructions, I have a nice overview of how the implemented code lines in Verilog language are later interpreted through RTL and GTL simulations and how different the two processes can be, as in our practise the simulation in RTL was completely right, but the error occurs in GTL because of the lack of thinking how the physical meaning of the implemented codes.
Later in the chip design part of floor planning and routing etc. leaves me a good combination with the knowledge I've learned in HSO.

7. Sources

Analog Devices. (October 2021). *analog.com.* Von ADAU1761 data sheet: https://www.analog.com/media/en/technical-documentation/data-sheets/adau1761.pdf abgerufen

Avnet. (November 2021). *Avnet*. Von Zedboard : https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/zedboard/ abgerufen

Karki, J. (December 2021). *EDN.* Von Fully differential amplifiers remove noise from common-mode signals: https://www.edn.com/fully-differential-amplifiers-remove-noise-from-common-mode-signals/ abgerufen

KIT - Karlsruher Institut für Technologie. (January 2022). PSoC - Lab exercise 7. Karlsruhe, Baden-Württemberg, Deutschland.

KIT - Karlsruher Institut für Technologie. (January 2022). PSoC - Lab exercise 8. Karlsruhe, Baden-Württemberg, Deutschland.

Peric, I. (March 2021). Lecture Design Analoger Systeme. Karlsruhe, Baden-Württemberg, Deutschland.

Stamp Sound. (January 2022). *Stamp Sound.* Von Can You Use A DAC Without An Amp?: https://stampsound.com/can-you-use-a-dac-without-an-amp/ abgerufen

Trenz Electronic. (October 2021). *Trenz Electronic*. Von https://shop.trenz-electronic.de/en/24539-ZedBoard-Zynq-7000-ARM/FPGA-SoC-Development-Board-Academic abgerufen

Wilson, P. (November 2021). *Science Direct.* Von Memory Interface: https://www.sciencedirect.com/topics/engineering/memory-interface abgerufen

Xilinx. (November 2021). *Xilinx Support.* Von FIFO Generator v13.1: https://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v13_1/pg057-fifo-generator.pdf abgerufen