SSY011 - Electrical Systems

Laboratory Report

Authors:

John Croft
19930814-7959

Andreas Johansson
19960813-8872

October 24, 2017

CHALMERS

John Croft
Andreas Johansson

Contents

John Croft
Andreas Johansson

Summary

This laboratory report presents the design, construction, testing and verification of a digital audio transfer system. The system is composed of a transmitter and a receiver, which are themselves composed of a number of subsystems. The chief component of the system was an FPGA, which primarily was used to serve as a digital control unit. The design is modular and each individual subsystem is presented in detail in this report.

The system succeeded in its primary goal of transmitting and receiving audio using the RS-232 protocoll, though failed to meet all of the set specifications, especially the bandwidth criteria, which was far below the expected range. Despite this, the system was eminently usable for transmitting speech.

John Croft
Andreas Johansson

1 Introduction

The goal of this laboratory assignment is to design, construct, test and characterize a digital audio transmission system. The system as a whole consists of many different component subsystems, as summarized in the block diagram in figure 1.

The design revolves around the use of the Altera DE1 evelopment board and the FPGA (*Field Programmable Gate Array*) it houses. The FPGA is programmed using VHDL (*(Very High Speed Integrated Circuit) Hardware Description Language*) in order to generate timings, to implement combinational and sequential gate-level logic and especially in order to construct finite-state machines as will be discussed later on in this report. The remaining electrical subsytems are constructed using discrete components on a prototyping breadboard. The frequencies in this system are low enough that the use of a prototyping board should result in any expected behaviour.

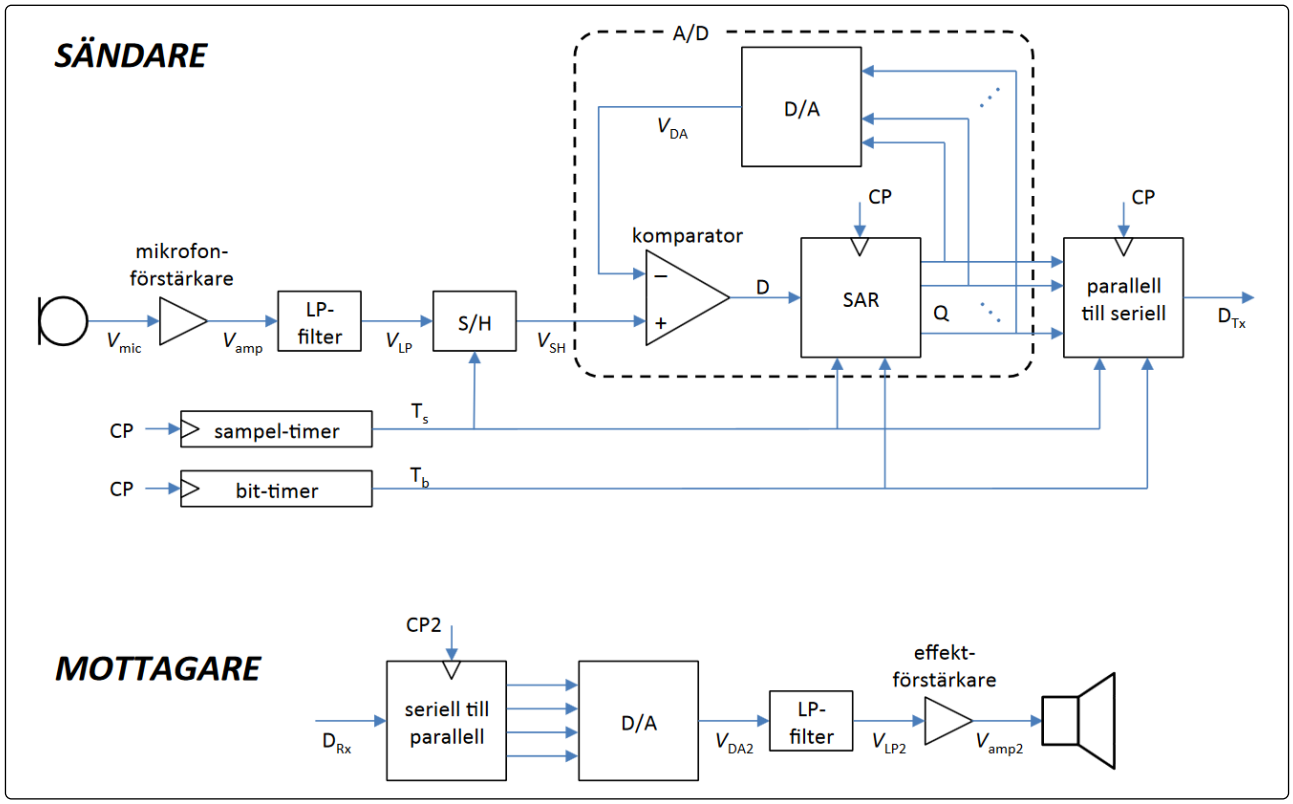

Figure 1: *Block diagram of the complete audio transmission system.*

Functionally, the system should be capable of receiving audio within the 20-12000Hz frequency range, digitize it and transmit it via a serial link using the RS-232 communication protocol to a receiver (using the exact same hardware) that then converts the digital data back to analog information and outputs it as standard 'line level' audio.

In order to realize this relatively simple functionality in practice, the subsystems required included signal amplifiers, power amplifiers (class AB), 4th order LP-filters, Sample & Hold circuits, DACs (Digital to Analog converters) and ADCs (Analog to Digital converters). Most of these subsystems were controlled by timed sequential and combinational networks on the FPGA.

2 Subsystems

This section will explain each subsystem in detail including its design specification, implementation in hardware and software and subsequent testing and verification.

Hardware designs are constrained to standard values, E12 for resistors and E6 for capacitors and component value calculations will take this into account.s

Figures and code may be edited to remove elements that are not strictly relevant to the function of the subsystem at hand. Complete schematics and code can be found in the Appendix.

2.1 Counter/Clock Generation

As seen in the block diagram in figure 1, clock signals control most of the subsystems and it is thus vital to the functionality of the system as a whole that they are sufficiently accurate and, in the case of subsystems that use both clocks, synchronized (ie. without *drift* relative to each other).

Two clock signals, the sample-clock and bit-rate-clock or T_s and T_b respectively are generated by the FPGA, based on its internal 50MHz clock. T_s has a frequency 24kHz and duty-cycle of 5% whereas T_b has a frequency of 240kHz and a duty cycle of 20ns (ie. a single clock-pulse). These particular frequencies are chosen in order to satisfy the Shannon-Nyquist sampling theorem which states that the sampling frequency, f_s, must be twice that of the sampled signal's frequency, f. Furthermore, the generated clock frequencies must be within 5% of that of the receiver's, as required by the RS-232 protocol.

The timings for these clock-signals are generated using two internally clocked registers that are incremented on each internal clock-pulse until they match preset values, whereupon the output signals change and/or the counters are reset.

Code Snippet 1 shows an implementation of the clock generators, albeit at lower frequencies than in the specification (though with the correct ratios) in order to aid in debugging.

Proper timings could be verified directly using an oscilloscope. The results from the implementation in Code Snippet 1 are shown in table 1.

Table 1: *Generated clock timings.*

Signal	T_s	T_b
Amplitude	3.3V	3.3V
Period, T	1.042 ms	∼25 ns
Frequency	960 Hz	9598 Hz
Pulswidth (at |V|/2)	52 us	18 ns

Code Snippet 1: Counter

```
architecture arch of clock_gen is
  signal  cnt_Ts  :  std_logic_vector(15 downto 0);
  signal  cnt_Tb  :  std_logic_vector(13 downto 0);
begin
  proc_Ts:  process(CLOCK_50, reset) -- (960 Hz)
    begin
      if reset='0' then --Clear counter and pull clock output LOW.
        cnt_Ts <= (others => '0');
        clk_Ts <= '0';
      elsif rising_edge(CLOCK_50) then
        if cnt_Ts = 49475 then -- at 95% of period set clock output HIGH.
          clk_Ts <= '1';
          cnt_Ts <= cnt_Ts + 1;
        elsif clk_Ts = 52079 then -- at end of period (note: (52079 + 1)*20ns => 960Hz).
          clk_Ts <= '0';          --  set clock output LOW and reset counter.
          cnt_Ts <= (others => '0');
        else
          cnt_Ts <= cnt_Ts + 1;
        end if;
      end if;
    end process;

  proc_Tb: process(CLOCK_50,reset) -- (9600 Hz)
    begin
      if reset='0' then        -- --Clear counter and pull clock output LOW.
        cnt_Tb <= (others => '0');
        clk_Tb <= '0';
      elsif rising_edge(CLOCK_50) then
        if cnt_Tb = (5207 - 1) then  --  1 clock-cycle before period end, set clock output
        ↪  HIGH.
          clk_Tb <= '1';
          cnt_Tb <= cnt_Tb + 1;
        elsif cnt_Tb = 5207 then  --  end of period, set clock output LOW.
          clk_Tb <= '0';
          cnt_Tb <= (others => '0');
        else
          cnt_Tb <= cnt_Tb + 1;
        end if;
      end if;
    end process;
end architecture;
```

2.2 D/A converter

The primary function of the D/A converter (or DAC) is to receive a digital parallel byte (0-255) and output a corresponding analog voltage (-5V to +5V). Furthermore, it should perform the conversion faster than the nominal sample-rate of the system, 12kHz.

The DAC subsystem consists of two active components: an integrated DAC chip with a parallel byte input and a corresponding analog output *current* and a current to voltage converter (*transimpedance amplifier*). The characteristics of these components are determined by auxiliary passive components which must be calculated according to the desired operation mode.

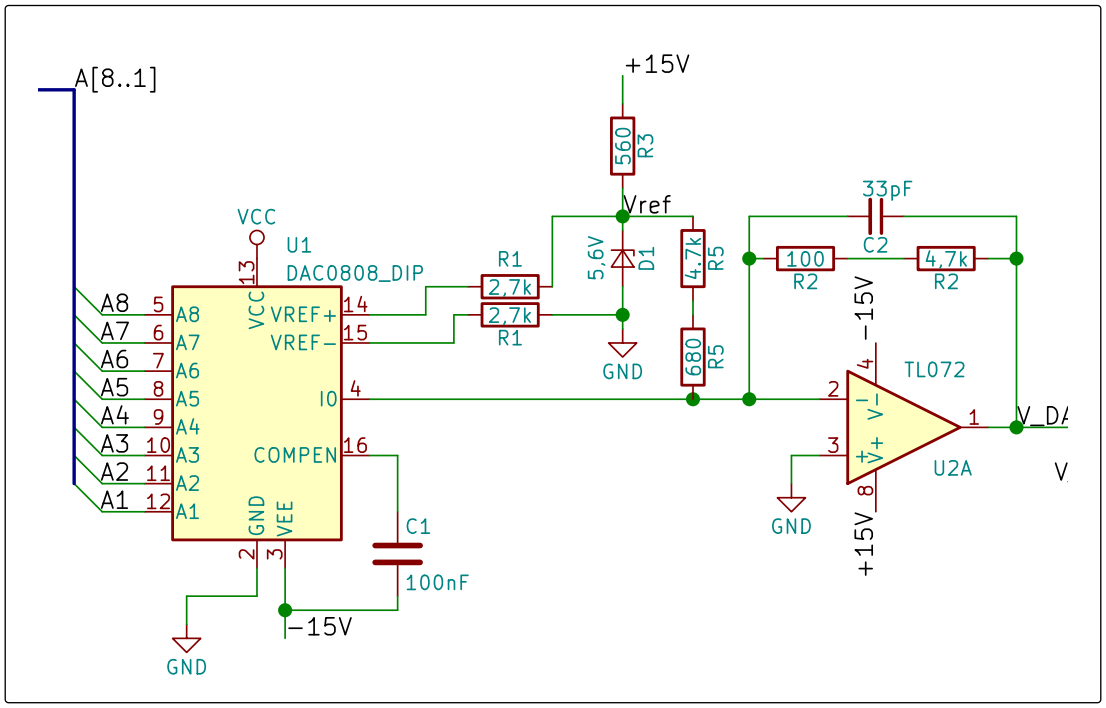

Figure 2: *DAC subsystem.*

2.2.1 Calculating component values

Several design constraints had to be observed when calculating the component values (refer to the circuit diagram in figure 2 for component, current and node names and values):

- $I_{14} \approx 2\text{mA}$.
- $I_{R_5} \approx 2\text{mA}$.
- $I_{\text{Zener}} > 10\text{mA}$ in order to stabilize the zener voltage, V_{REF}.
- $I_{R_3} < 20\text{mA}$ in order to protect R_3.
- V_{DA} should range from -5V to 5V.

First, R_1 may be calculated based on the fact that $I_{14} \approx 2\text{mA}$

$$R_1 = \frac{V_{\text{REF}}}{2\text{mA}} = \frac{5.6\text{V}}{2\text{mA}} = 2.7\text{k}\Omega \tag{1}$$

R_3 can then be calculated, as the currents at V_{REF} are known. First we let $I_{\text{zener}} > 10\text{mA}$

$$R_3 < \frac{15\text{V} - 5.6\text{V}}{10\text{mA} + 2\text{mA} + 2\text{mA}} = 671\Omega \approx 560\Omega \tag{2}$$

then we pick the nearest lower standard value resistor in order to allow for marginally more current than we originally calculated.

Here we deviate slightly from the specification and calculate R_2 for a non-bipolar DAC with a range from 0 to 10 V. This is important, as it temporarily eliminates R_5 while allowing us to examine the characteristics of the subsystem at the same voltage range. The 'conversion' formula for the DAC IC (DAC0808) is as follows

$$I_4 = I_{14}\frac{A}{256} \tag{3}$$

, where A is the digital input to the DAC0808 (ie. 0-255). With some quick circuit analysis we can quickly determine the equation for R_2 as

$$R_2 = V_{DA} \cdot \frac{R_1}{V_{\text{REF}}}\frac{256}{A} \tag{4}$$

Letting $V_{DA} = V_{DA_{\text{MAX}}} = 10\text{V}$ and $A = A_{\text{MAX}} = 255$ gives

$$R_2 = 4840\Omega \approx 4.8\text{k}\Omega = 4.7\text{k}\Omega + 100\Omega$$

From here, it is relatively easy to determine a formula for R_5 in much the same manner, being mindful of the fact that we are moving the operating point to a level at which the DAC is bipolar. V_{DA} can now be expresed as

$$\begin{aligned} V_{DA} &= I_2 \cdot R_2 \\ &= (I_4 - I_5)R_2 \\ &= (I_{14} \cdot \frac{A}{256} - I_5) \cdot R_2 \\ &= \Big(\frac{V_{\text{REF}}}{R_1}\frac{A}{256} - \frac{V_{\text{REF}}}{R_5}\Big) \cdot R_2 \end{aligned} \tag{5}$$

If we let A and V_{DA} take on their extreme values (ie. $\pm 5\text{V}$) we get

$$R_5 = 5374\Omega \approx 5.38\text{k}\Omega = 4.7\text{k}\Omega + 680\Omega$$

2.2.2 Test & Verification

The DAC was tested in its non-bipolar state in which the output range is between approx. 0V and 10V. Constant digital test inputs where used and the corresponding outputs where measured as well as theoretically calculated, as shown in table 2.

Table 2: *Measuring the analog output in relation to the digital input.*

Digital Input (decimal)	Theoretical Analog Output	Measured Analog Output
0	0	0
16	0.622	0.635
32	1.244	1.271
64	2.488	2.547
128	4.977	5.094
255	9.916	10.140

In order to test the linear as well as the step-response (or slew-rate) of the DAC, the FPGA was programmed to output a parallel digital signal in such a way that would result in a sawtooth wave on the output. The code implementation is shown in code snippet 2. The program increments or decrements a parallel byte at a rate determined by the generated bit-rate clock, implemented in chapter 2.1, but configured to run at 100 Hz. This byte is then outputted over the DE1's GPIO header to the DAC0808 IC. Whether this value increases or decreases depends on the state of a toggle switch, SW9. This allows control over whether the sawtooth signal and the 'step' at the transition between periods is rising or falling.

Code Snippet 2: Counter

```
1  ---------------------------------------------------------------------------
2  -- Process: proc_bin_cnt
3  -- Description: 100Hz binary counter.
4  --              Increments or decrements depending on the state of SW9.
5  --              Creates a sawtooth pattern by allowing bin_cnt to over/underflow.
6  -- Input(s) : CLOCK_50, SW9, reset
7  -- Output(s):
8  -- Internal Signals: bin_cnt
9  ---------------------------------------------------------------------------
10   proc_bin_cnt: process(CLOCK_50, reset)
11     begin
12       if reset='0' then
13         bin_cnt <= (others => '0');
14       elsif rising_edge(CLOCK_50) then
15         if clk_Tb_buff='1' and SW9='1' then -- if toggle switch HIGH...
16           bin_cnt <= bin_cnt + 1;           -- count up.
17         elsif clk_Tb_buff='1' and SW9='0' then -- else count down.
18           bin_cnt <= bin_cnt - 1;
19         end if;
20       end if;
21   end process;
22 end arch;
```

The resulting sawtooth signal has a theoretical period length of $\frac{256}{100} = 2.56s$ and a measured period length of approximately $2.60s$.

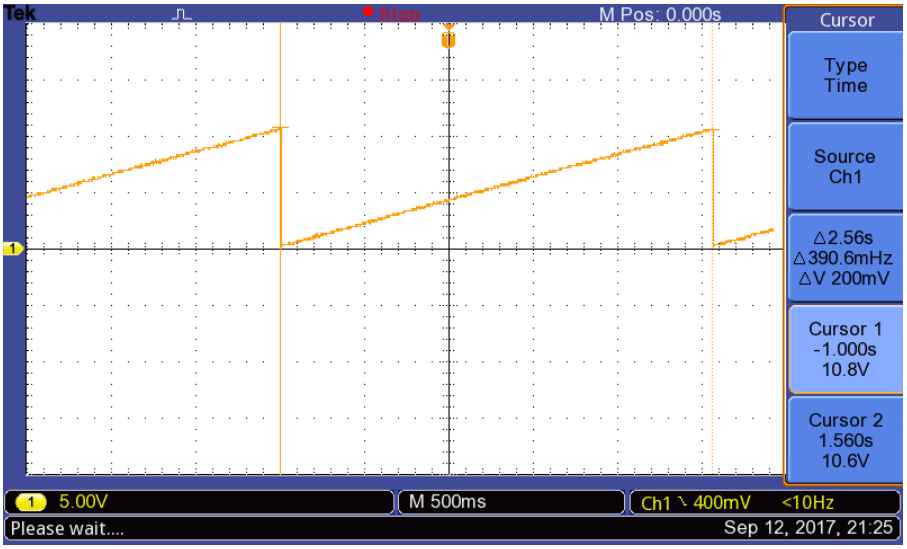

Figure 3: *2.6*Hz *sawtooth signal generated by the FPGA.*

Having ascertained the accuracy of sawtooth generator, the frequency was changed to 9600 Hz in order to examine the 'slew rate' of the DAC, that is, how quickly the output V_{DA} responds to a change on the digital input. This was measured on the 'step'-transitions under two conditions: with and without the inclusion of C_2 (see figure 2).

$$\begin{cases} \text{With } C_2 & : \text{SR} = 11.3\text{V}\,\mu\text{s}^{-1} \\ \text{Without } C_2 & : \text{SR} = 6.1\text{V}\,\mu\text{s}^{-1} \end{cases}$$

The step response resulted in rather significant ringing however, as seen in figure 4, which effectively limits the bandwidth. The maximum frequency of the DAC can then be estimated (on the presumptions that the output signal is allowed to settle before each transition, and that the fall time is similar in length).

$$f_{MAX} = \frac{1}{\text{rise-time} + \text{fall-time}} \approx 200\text{kHz} \tag{6}$$

This may seem as though it fails to meet the required bandwidth of 240 kHz, but under normal conditions the input will not jump between it's extreme values and so it will not be limited.

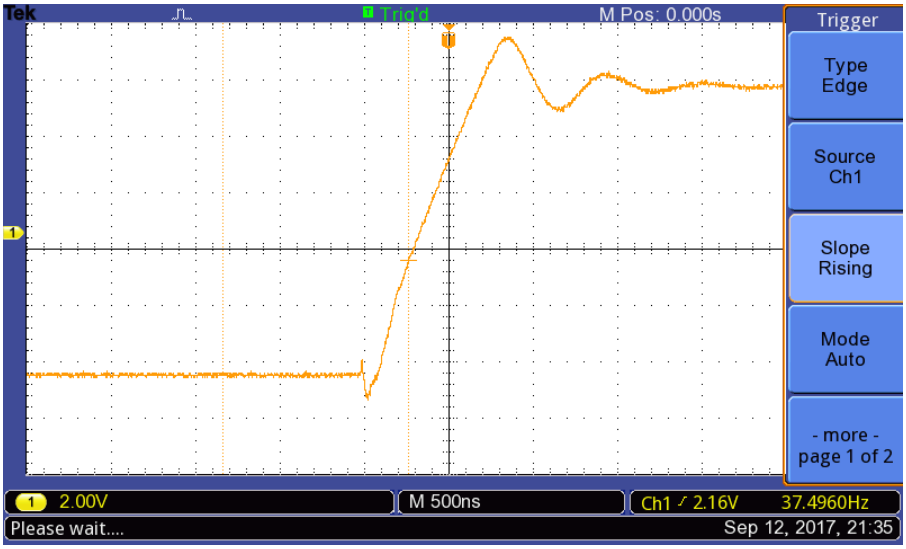

Figure 4: *Ringing on the DAC output during a large 'step' in the input signal.*

2.3 A/D converter

The function of the ADC (Analog to Digital Converter) is to convert an analog signal of $\pm 5V$ to an 8 bit digital representation (ie. 0-255). The ADC contains three main components: an FPGA implementation of a SAR (Successive Approximation Register) state-machine, a DAC (previously introduced in chapter 2.2) and a comparator based on the LM311 IC. The full circuit is shown in figure 5.

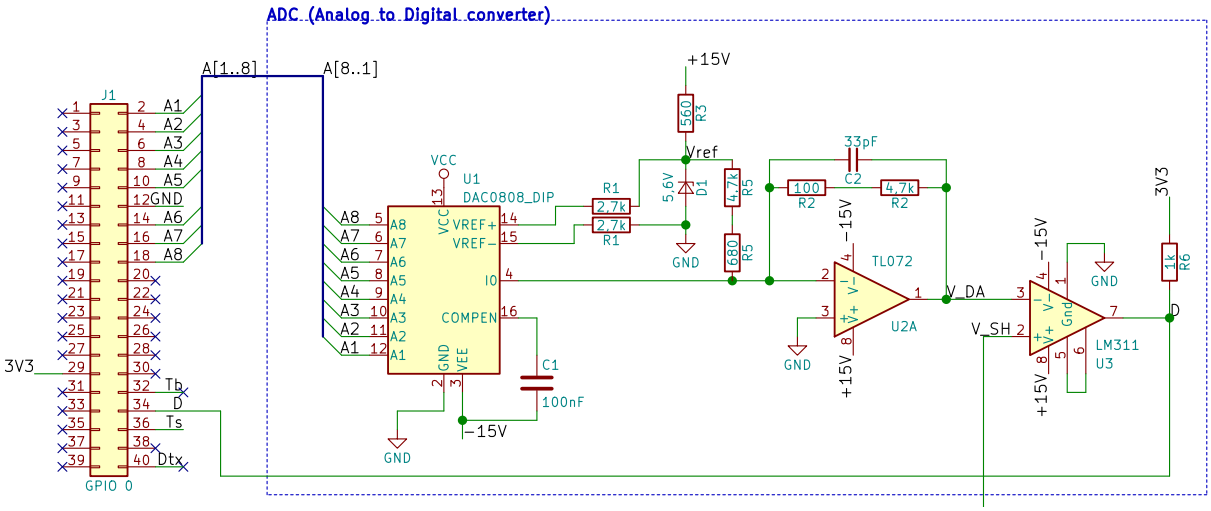

Figure 5: *ADC circuit diagram. To the left is the FPGA's GPIO connector, in the middle is the DAC subcircuit and to the right is the voltage comparator.*

The system can be thought of as having the SAR as its main component and the DAC in series with the comparator as a feedback signal. This feedback signal is labeled 'D' in the circuit diagram.

The functional principle of the ADC can be described with the help of both the circuit diagram and the SAR state-machine diagram in figure 6. The signal to be converted is inputted on V_{SH}.

To start a conversion the SAR receives a T_s pulse. It then sets its parallel output (A1-A8 in the circuit diagram) to $(10000000)_2$ ie. half of the maximum range of the DAC. This value is converted by the DAC to an analog voltage and passed on to the comparator, where it compared to the input signal. If the input signal exceeds the signal generated by the SAR, 'D' is pulled HIGH in logic terms, else LOW. The SAR will then store this value of 'D' in the MSB position of its eventual converted digital output.

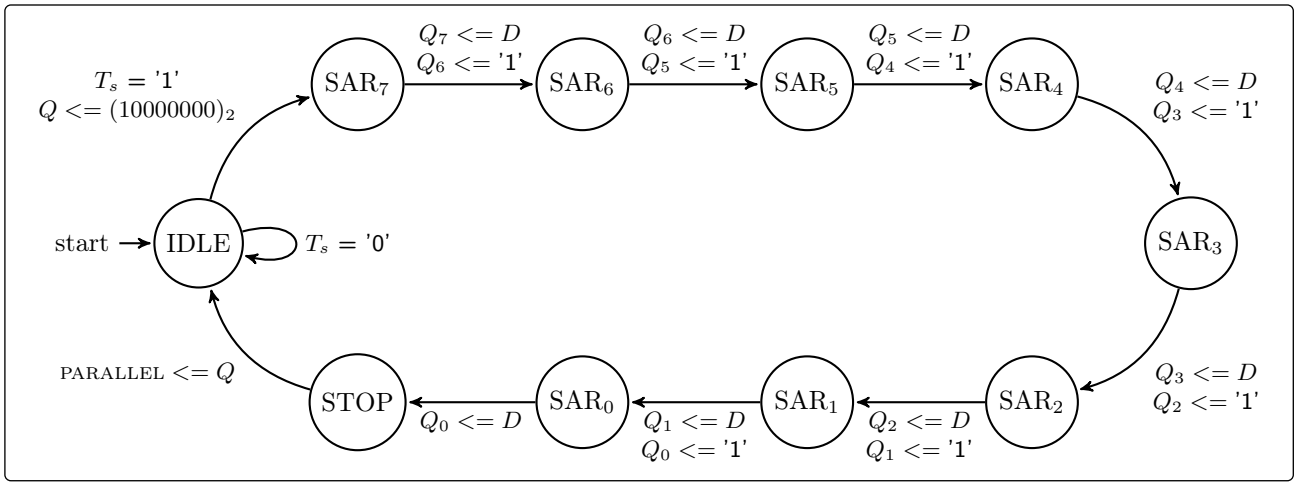

Figure 6: *Finite state-machine describing the SAR's function.*

Next, the SAR sets the MSB-1 of its parallel output HIGH, while keeping the previous value of the MSB, and the process repeats, doing so for each bit until the complete digital byte representation is determined.

The way in which this successive approximation takes place is perhaps best seen on a time-graph such as in figure 9.

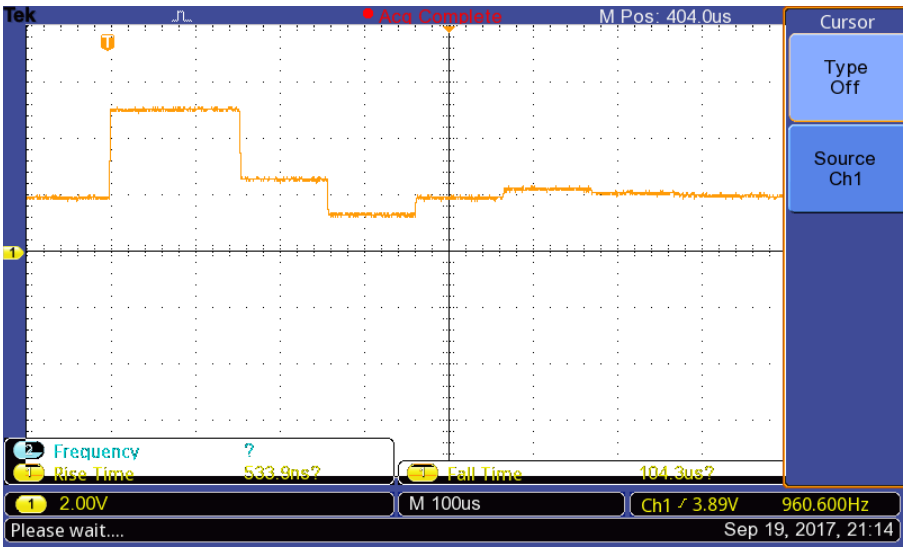

Figure 7: *ADC successively converging on 1.9V. Note: the ADC used in this image is not bipolar!*

The VDHL implementation of the SAR below shows how the SAR solves bits with a case structure by saving the current state and moving to the next state. In this process there

are three input signals state,SAR and Q. "Q" is the signal which will provide changes to the output voltage and "state" continues the bitlow by incrementing by 1 after each set bit. If V_DA approximation higher than sampled signal, D is set LOW, and thus also Q(n). When the voltage has converged the state will move to stop and loop back into the idle state and wait for the next TS signal..

Code Snippet 3: SAR Implementation as a VHDL Process

```
proc_SAR_comb : process(CLOCK_50,reset)
    begin
    if reset='0' then
        state <= IDLE;
        SAR_result <= (others => '1');
    elsif rising_edge(CLOCK_50) then
        if clk_Ts='1' then
            Q <= "10000000"; -- MSB set, first approximation is half of maximum value.
            state <= SAR7;      -- Switch to next state in sequence.
        elsif clk_Tb='1' then
            case state is
                when SAR7 =>
                    Q(7) <= D; -- If V_DA approximation higher than sampled signal,
                               -- D is set LOW, and thus also Q(n).
                    Q(6) <= '1';    -- Add 1/4 of maximum value to approximation,
                                    -- and so on for subsequent states.
                    nextState <= SAR6;
                when SAR6 =>
                    Q(6) <= D;
                    Q(5) <= '1';
                    nextState <= SAR5;
                .
                .
                . -- (truncated)
                .
                .
                when STOP =>
                    nextState <= IDLE;  -- wait for next Ts trigger.
                    SAR_result <= Q;
                when IDLE | U =>
                    -- do nothing
            end case;
        end if;
    end if;
   end process;
```

The A/D converter consists of an LM311 chip which is used to compare voltage sources from D/A aswell as the sample and hold circuit. The input sources are compared to create a digitalized source of power through recursive comparisons of the voltage from a D/A converter with the desired analog signal. The LM311 chip uses three state logic which consists of a logical zero and a high impedance state "Z". When the sample and hold circuit outputs a greater voltage than the DAC, the SAR will interpret the signal as a logical one and increase

the digital voltage output.

$$\begin{cases} \text{Vout} = 0: & V_{SH} > V_{DA} \\ \text{Vout} = \text{Z}: & V_{SH} < V_{DA} \end{cases} \tag{7}$$

Table 3: *DC inputs and corresponding ADC output.*

Analog DC Input [V]	Theoretical Digital Output	Measured Digital Output
-5	00000000	00000011
-2	01001101	01001111
-1	01100111	01101111
0	10000000	10000001
1	10011010	10011111
2	10110010	10110101
5	11111111	11111111

The testing provided a table which showed that the conversion displayed rounding errors. The ADC rounding error was calculated to be about 0.4 procent off target value as a worst case scenario, which may or may not be acceptable depending on the application.

The ADC converts with is restrained by a delay which is the creating a maxiumum bitratio. The theroetical delay was calculated in the simulation below. The delay was calculated to around 800 ns.

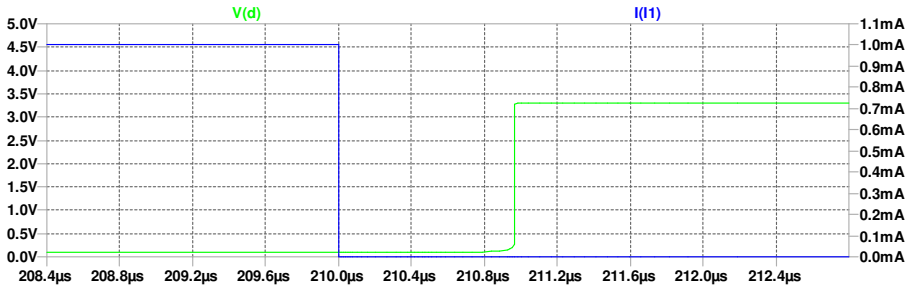

Figure 8: *Q->D falling edge*

To be able to calculate the ADC maximum bitratio the delay of the ADC has to be known. The delay can be interpreted as the fall time of the Q signal until the stability of the D signal. With this we can then equate the bit transfer to be the frequency differential of these signals. By triggering the oscilloscope on a negative flank on the signal Q, the delay could be calculated to 850 ns. With this delay the maximum bit ratio could then be calculated to be 1.18 Mb/s.

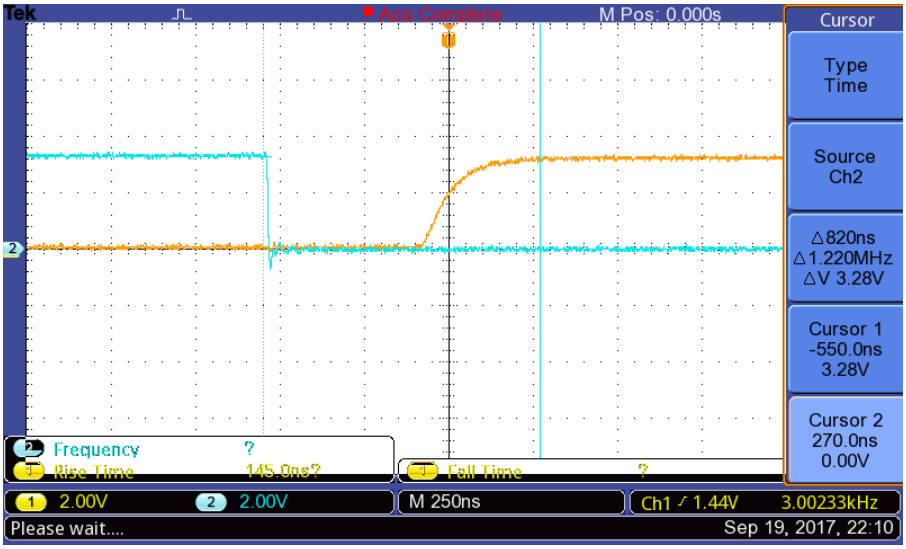

Figure 9: *Q->D falling edge*

2.4 Sample and hold

In order for the ADC to accurately convert an analog voltage, it must be held constant for the duration of the conversion. A S/H (Sample & Hold) circuit uses an external control signal (in this case the sample-clock, T_s) to either sample or 'hold', that is, keep the voltage constant.

The S/H achieves this behaviour by allowing the voltage over its capacitor (see figure 10) to vary with the input signal during 'sample' mode.

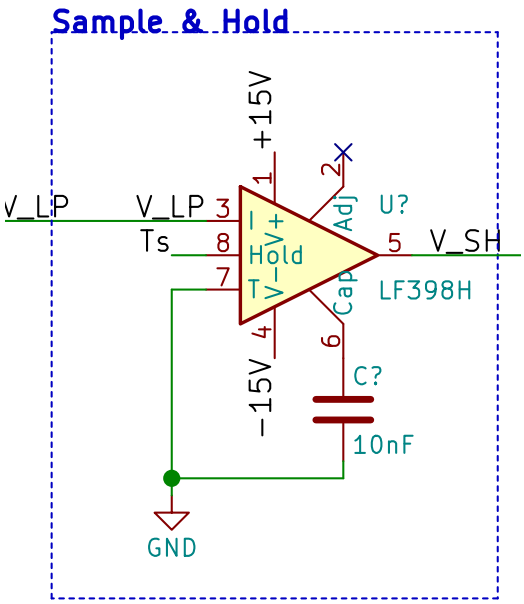

Figure 10: *S/H circuit. The 10nF capacitor is responsible for 'holding' the sampled voltage.*

The circuit used in this project uses the purpose-designed IC LF398 which has the following two states

$$\begin{cases} \text{Sample}: & V_8 - V_7 > 1.4\text{V}, \\ \text{Hold}: & V_8 - V_7 < 1.4\text{V} \end{cases} \tag{8}$$

Using this relationship and considering that the FPGA uses LVTTL (ie. 3.3V logic), setting $V_7 = \text{GND}$ and $V_8 = T_s$ (the sample clock) gives us the required behaviour. The fact that T_s has a 5% duty cycle means that there is always enough time for the capacitor to reach the level of the input signal before it is 'held'.

A comparison between the input and output signals from the S/H can be seen in figure 11, while the input signal is a triangle wave of an arbitrary frequency, the output samples the input and holds it constant with a frequency of 960Hz.

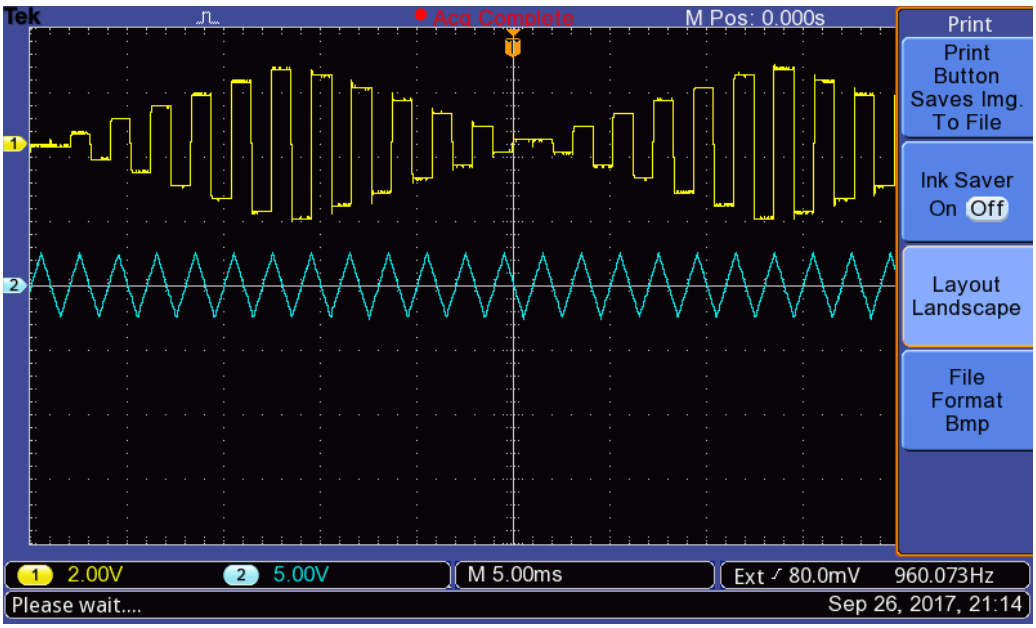

Figure 11: *The input signal (blue) is a triangle wave that gets periodically sampled by the S/H. The resulting output signal (yellow) is held between samples.*

2.5 Serial transmitter

The serial transmitter is implemented in software using a shift register and makes use of the DE1's onboard RS-232 port to translate between logic and the RS-232 specified voltage levels, which in the DE1's implementation range between ±8V (-8V being logic '0' and +8V being logic '1').

The data to be transmitted is the parallel byte 'Q', generated be the ADC in chapter 2.3. The serial transmitter completes the byte with a START bit (logic '0') before the LSB and a STOP bit (logic '1') after the MSB, according to the RS-232 protocol, before transmitting the complete 10-bit packet LSB first. When idling, the transmitter outputs a continuous STOP bit.

The serial transmitter's operation is triggered by a T_s pulse and each individual bit is then sent upon receiving a T_b pulse.

Code Snippet 4: Shift Register

```
1  proc_shift_register: process(CLOCK_50, reset)
2      begin
3          if reset='1' then
4              SR <= (others => '1');  -- Init. SR with STOP bits.
5          elsif rising_edge(CLOCK_50) then
6              if SR_start='1' then  -- Shift Register activated?
7                  if clk_Ts='1' then
8                      SR <= SAR_result & '0';  -- Load Shift Register with DATA & START bit.
9                  elsif clk_Tb='1' then
10                     SR <=  '1' & SR(8 downto 1);  -- Shift data right, Shift in STOP bit.
11                 end if;
12             end if;
13         end if;
14     end process;
15
16 serial <= SR(0); -- Serial output from LSB of Shift Register
```

Code snippet 4 shows a VHDL process implementing the serial transmitter. *'SR'* is the contents of the shift register, ie. the complete packet to be sent, *'SAR_result'* is an intermediate register to hold contents of Q (Q itself cannot be used as the ADC and serial transmitter operate in parallel, and Q would then change before it was fully sent!). *'SR_start'* is simply a flag indicating whether at least one ADC conversion has taken place, so that the transmitter under no circumstances transmits undefined data.

As seen in the code, a positive T_s signal reads the data to be sent into the shift register prepended by a START bit. A positive T_b signal then causes the shift register to logical shift right while also shifting in a STOP bit from the left (that is, in the MSB position). The LSB of the shift register is continuously outputted on the serial link. Eventually the shift register will contain and therefore output only STOP bits, which will be interpreted by the receiver as a complete transmission (or otherwise serve to synchronize the transmitter and receiver, see the next chapter).

The serial transmitter was tested by allowing the ADC to convert a known external DC voltage, and transmitting the resulting byte to a PC-based receiver. On a successful transmission the

PC interpreted the information as an ASCII character, which was then compared to both the theoretical digital value of the input signal as well as the result of the conversion in the ADC, which was outputted directly to 8 debug LEDs by the FPGA.

2.6 Serial Receiver

The serial receiver (hereafter R_x) should be able to asynchronously interpret an incoming stream of serial bits as data packets according to the RS-232 protocol, and output the encapsulated data as a parallel byte. It is in this sense a serial to parallel converter, the opposite of the serial transmitter (hereafter T_x) in 2.5. Also in accordance with the RS-232 protocol, the R_x should be able to automatically synchronise itself to the T_x (using clock recovery/symbol synchronization) and compensate for both clock drift (the frequencies of T_x and R_x do not exactly match) and triggering a read at the wrong position in a packet (block synchronisation).

The receiver is implemented in software and utilizes the DE1's onboard serial port. To implement the functionality described above, a state machine (of type Mealy) is used as described in figure 12.

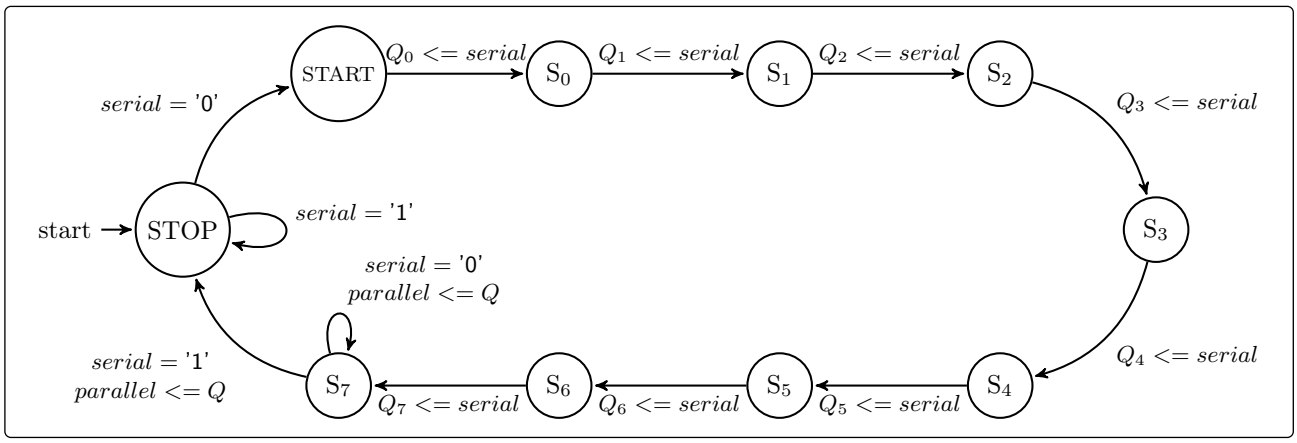

Figure 12: *Finite state-machine describing an 8-bit serial to parallel RS-232 receiver.*

The receiver initializes in the STOP state where it is assumed that it is receiving STOP-bits ('1') and waits for a START-bit, indicating the start of a packet, before transitioning to the START state. It then successively transitions from the START state to the S7 state at a rate determined by an internally generated sample-clock, reading a bit from the packet at each transition. Once at the S7 state, it awaits a STOP-bit while still sampling with the same frequency. By *synchronously* waiting until this STOP bit is received, the position in the packet where the next read will start is moved, effectively resulting in block synchronization. Once the STOP-bit is received it transitions back to the STOP state and awaits the next START-bit. However, unlike previously, it will start the next read immediately (asynchronously) upon receiving the START-bit. This results in clock drift compensation as the R_x and T_x are essentially resynchronised on each new start, and any drift in the relative timings is nullified.

The code implementation of the state machine is shown in code snippet 5. In keeping with the model of a Mealy machine, the state machine is implemented as two processes: a synchronous and an asynchronous/combinational part. The combinational part handles the logic that determines outputs and state changes, whereas the synchronous part handles timings and synchronously latches the combinational logic.

Also note that the synchronous process reads or 'samples' each data bit in the middle of its period in order to get the most reliable reading.

Code Snippet 5: Serial Receiver

```
proc_Rx_sync : process(CLOCK_50, reset) -- Synchronous process.
    begin
        if reset='0' then
            state <= STOP;
            Q <= (others => '0');
            cnt <= (others => '0');
        elsif rising_edge(CLOCK_50) then
            cnt <= cnt + 1;
            if (state = STOP AND serial = '1') then -- If state is STOP and receiving
             ↪  STOP-bits...
                cnt <= (others => '0');  -- ...reset counter and await START. (Block
                 ↪  synchronization)

            elsif cnt = 103 then  -- At half of period T, sample data and transition to
             ↪  next state.
                state <= next_state;
                Q <= next_Q;
                parallel <= next_parallel;
            elsif cnt = 207 then  -- At full period T, reset counter.
                cnt <= (others => '0');
            end if;
        end if;
    end process;

proc_Rx_comb : process(state, serial, Q) -- Combinational process.
    begin
        next_Q <= Q;
            case state is
                when STOP =>
                    next_state <= START;
                when START =>
                    next_state <= S0;
                    next_Q(0) <= serial;
                when S0 =>
                    next_state <= S1;
                    next_Q(1) <= serial;
                ......           -- continuing for Q2-6
                when S6 =>
                    next_state <= S7;
                    next_Q(7) <= serial;
                when S7 =>
                    next_parallel <= Q;
                    if serial='0' then -- If serial is NOT STOP-bit ...
                        next_state <= S7; -- ... do nothing. (Drift compensation)
                    else  -- If serial is STOP-bit...
                        next_state <= STOP; -- transition to STOP state.
                    end if;
                when others =>
                    -- undefined state
                    next_state <= STOP;
            end case;
    end process;
```

The serial receiver was tested by linking it to a PC-based serial transmitter and configuring

both for 9600 buad. ASCII characters where then sent from the PC and the resulting parallel byte on the receivers output was directly displayed by the FPGA using 8 LEDs. The test results can be seen in the table below.

Table 4: *Sent ASCII characters and resulting bit-patterns and voltages D/A conversion. Note that the receiver's DAC is bipolar at this stage.*

ASCII-character	Received bit-pattern	Measured Analog Output [V]
P	01010000	-1.95
?	00111111	-2,62
å	10000110	0.17
!	00100001	-3,80
=	01011101	-2,70
A	01000001	-2,54

2.7 Audio Amplifier

There are two audio amplifiers in the system: the transmitter has a signal amplifier for the microphone and the receiver has a power amplifier for the speaker. The difference is that the signal amplifier increases the amplitude of the microphone's signal to match the range of the ADC (ie. ±5V), whereas the power amplifier increases the amount of power that can be delivered to the load without substantially changing the amplitude.

2.7.1 Characterising the signal amplifier

The microphone and signal amplifier subcircuit is shown in figure 13.

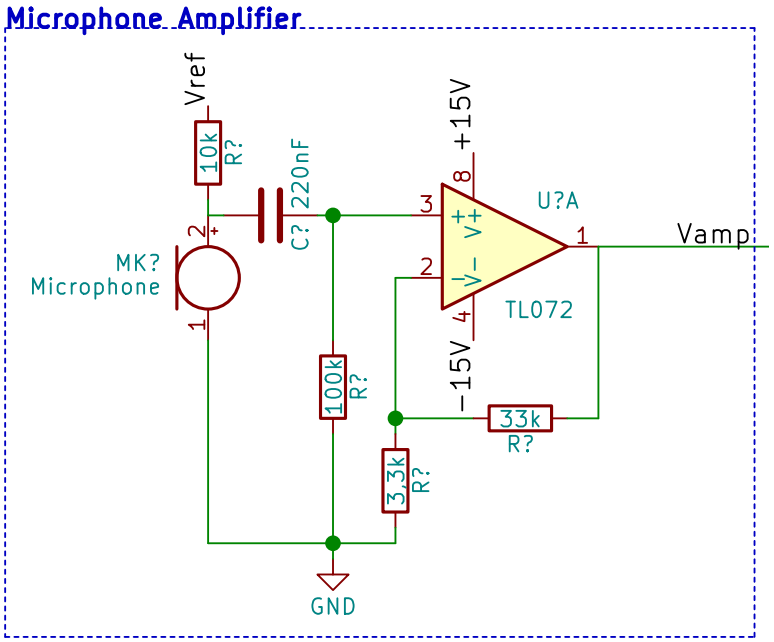

Figure 13: *An analog signal is generated by the microphone and amplified by the OP-amp on the right.*

In order to calculate the input impedance Z_{in} of the signal amplifier from the perspective of the microphone, the small signal circuit must be considered. If we let the microphone act as a current source, short any external voltage sources and consider the OP-amp ideal (in that it draws no current on its comparator inputs), then

$$Z_{IN} = \frac{10\text{k}\Omega \cdot 100\text{k}\Omega}{10\text{k}\Omega + 100\text{k}\Omega} = 9090.9\Omega \tag{9}$$

Since this result differs from the specification's 2kΩ, the sensitivity of the microphone will be affected and must be recalculated.

Since the microphone's specification gives a typical sensitivity of -39dB V Pa^{-1} @ 2kΩ input impedance, we can express this as an *effective* (RMS) voltage

$$U_{eff} = 1V \cdot 10^{\frac{-39}{20}} = 0.0112V \tag{10}$$

in order to calculate a current at the specifications input impedance

$$I_{eff} = \frac{U_{eff}}{Z_{IN}} = \frac{0.0112V}{2\text{k}\Omega} = 5.61\mu\text{A} \tag{11}$$

but at this current with the new input impedance (remember, the microphone can be seen as a *current* source) the effective voltage is

$$U_{eff} = Z_{IN_{new}} \cdot I_{eff} = 9090.9\Omega \cdot 5.61\mu\text{A} = 0.051\text{V} \tag{12}$$

giving the new sensitivity

$$20log\Big(\frac{0.051V}{1V}\Big) = -25.8\text{dB V Pa}^{-1} \tag{13}$$

It is now possible to calculate the maximum soundpressure that the system can receive before the transduced signal exceeds the range of the ADC, and clipping occurs.

If the amplification of the non-inverting amplifier in figure 13 is $A_U = 11$ and the maximum amplitud after amplification is $|V_{amp}|_{\text{MAX}} = 5\text{V}$, then then maximum microphone signal amplitude is

$$U_{\text{mic}} = \frac{5V}{11} = 0.455 \Rightarrow 20log\left(\frac{\frac{0.455V}{\sqrt{2}}}{1V}\right) = -9.85\text{dB V Pa}^{-1} \tag{14}$$

at 9090.9Ω input impedance.

Taking the difference between the microphone's maximum output as defined above, and the nominal output yields

$$-9.85\text{dB V} - -25.8\text{dB V Pa}^{-1} = 15.95\text{dB V} \tag{15}$$

which can then be directly added to the nominal SPL (ie. 1 Pa = 94dB SPL).

$$94 + 15.95 = 109.95\text{dB SPL} \tag{16}$$

to get the maximum allowed sound pressure.

Several characteristics were best calculated using SPICE such as the DC operating point, lower cutoff frequency and the amplification of the non-inverting amplifier (using a better mathematical model than the assumed ideal one). The results of these simulations follow in the table below.

Table 5: SPICE simulation of microphone amplifier.

DC Operating Point [V]	Cutoff Frequency [Hz]	Gain
3.08	6.6	10.96

2.7.2 Characterising the power amplifier

The power amplifier circuit is shown in figure 14. The total gain of the amplifier is simple to compute with the approximation that the CMOS amplifier stage has no gain and that the capacitor can be approximated by a short. This leaves two potential dividers with a non-inverting amplifier in between them, also acting as a convenient buffer.

$$\text{Thus } A_U = \frac{1\text{k}\Omega//47\text{k}\Omega}{10\text{k}\Omega + 1\text{k}\Omega//47\text{k}\Omega} \cdot \frac{33\text{k}\Omega + 10\text{k}\Omega + 100\Omega}{10\text{k}\Omega} \cdot \frac{20\Omega}{20\Omega + 47\Omega} = 0.115 \tag{17}$$

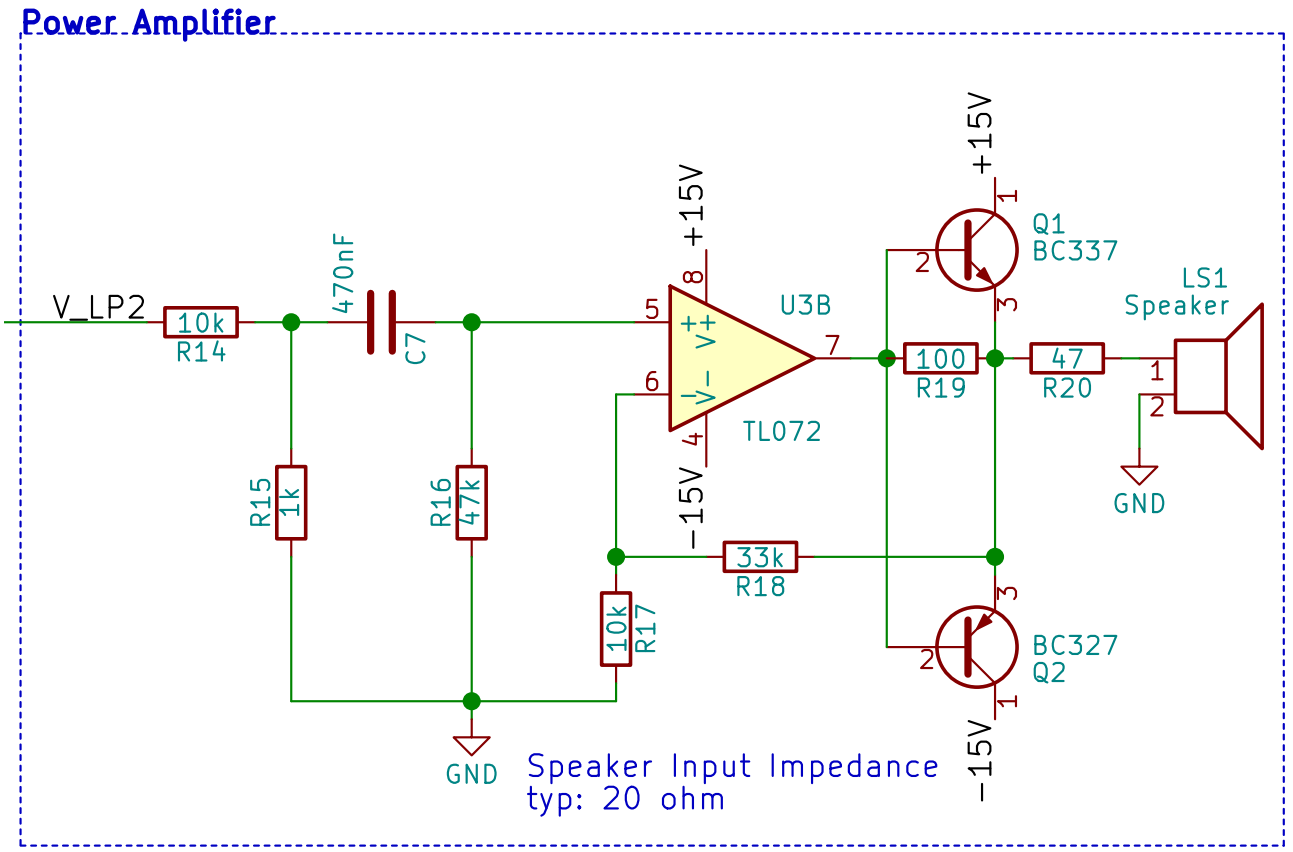

Figure 14: *The power amplifier shown here consists of several cascaded stages. From right to left: a potential divider, a passive HP-filter, a non-inverting amplifier, a CMOS amplifier and another potential divider in which the speaker completes the divider.*

The maximum SPL that a typical speaker with sensitivity of 100 dB SPL/mW can generate with a 5V AC input can now be calculated.

$$P_{\text{MAX}} = \frac{\left(\frac{5V \cdot 0.115}{\sqrt{2}}\right)^2}{20\Omega} = 8.26\text{mW} \tag{18}$$

$$\Rightarrow 10log\left(\frac{8.26\text{mW}}{1\text{mW}}\right) = 9.17\text{dB}\,\text{mW} \tag{19}$$

thus the maximum sound pressure level is

$$100\text{dB SPL/mW} + 9.17\text{dB SPL/mW} = 109.17\text{dB SPL/mW}$$

SPICE was used to simulate a few remaining characteristics and to confirm the gain calculation, with the results presented in the following table.

Table 6: *SPICE simulation of power amplifier. NOTE: power amplification is* $\frac{P_{OUT_{MAX}}}{P_{IN_{MAX}}}$.

Power Amplification	Lower Cutoff Freq. [Hz]	Upper Cutoff Freq. [kHz]	Gain
8	7	172.9	0.115

The power amplifier was tested in practise, which resulted in the gain/frequency curve seen in figure 15.

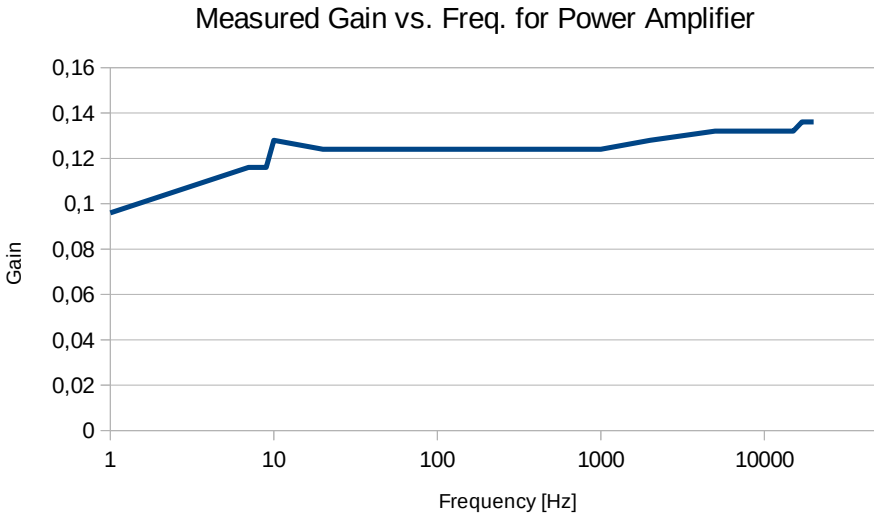

Figure 15: *Measured Gain vs. Freq. for Power Amplifier. Note the attenuation below 10 Hz.*

2.8 LP filter

The system itself is required to inherit a maxium frequency of 12 kilo hertz. To fit the requirement the circuit is required to contain a Low Pass filter. The LP filter is designed and calculated as a fourth order butterworth filter.

The equation of a general fourth order butterworth filter:

$$H_s = \frac{1}{1 + 2.613a + 3.414a^2 + 2.613a^3 + a^4} \tag{20}$$

The LP filter simply put, is set to cutoff at the a targeted frequency. The butterworth filter is set to be built in sallenkey filter structure.

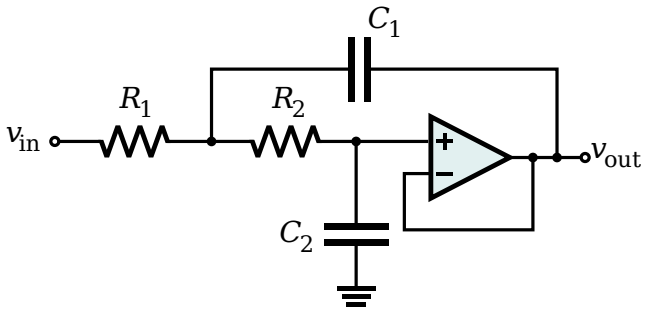

Figure 16: *A general LP sallenkey filter*

By cascading two second order sallen-key filter structures, the butterworth equation splits into two second order butterworth filters

$$H_s = \frac{1}{(1 + 0.765a + a^2) * (1 + 1.848a + a^2)} \tag{21}$$

To calculate the components for the sallenkey filter which can be equated to the butterworth filter:

$$\frac{1}{(1 + ax + a^2)} = \frac{1}{(1 + (R_1 + R_2)C_2 + R_1R_2C_1C_2)} \tag{22}$$

$$a = \frac{S}{\omega} = \frac{S * 0.765}{F_c * 2\pi} = \frac{S * 0.765}{12000 * 2\pi} \tag{23}$$

Calculating The first sallenkey filter as the first part of the butterworth filter:

$$\frac{1}{(1 + 0.765a + a^2)} = \frac{1}{(1 + (R_1 + R_2)C_2 + R_1R_2C_1C_2)} \tag{24}$$

Let C2 be equal to 1nF and assuming that the resistor are equal

$$2R = \frac{S}{12000 * 2\pi * C_2} => R = 5075\Omega \tag{25}$$

With three known components we can calculate the last capacitor

$$2R = \frac{S^2}{12000 * 2\pi * C_2 * R^2} => C_1 = 6.8nF \tag{26}$$

With the first filter calculated, computing the second filter can be used the same methodology, inserting the second part of the buttworth filter equation we get:

$$a = \frac{S}{\omega} = \frac{S * 1.848}{F_c * 2\pi} = \frac{S * 1.848}{12000 * 2\pi} \tag{27}$$

Let C2 = 1.5nF and R1=R2 => gives 8200Ω and C2 = 1,72 nF which can be rounded to 1.5 nF. With all the components solved they create the schematic:

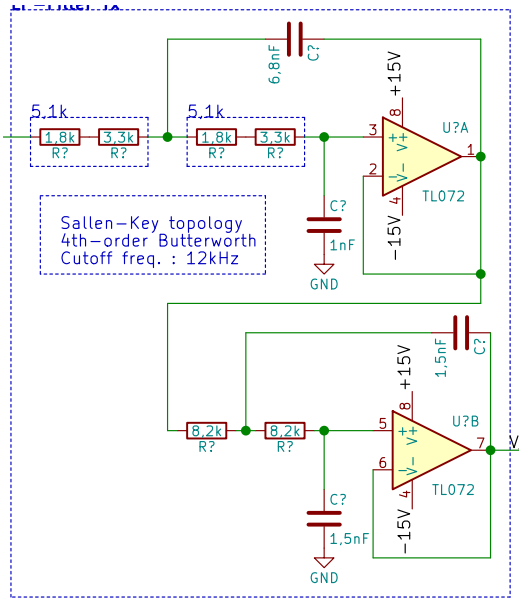

Figure 17: *Two sallenkey filters cascaded with calculated component values*

Using the above component values, we can implement the filter in LTSpice, giving us the following frequency response plot:

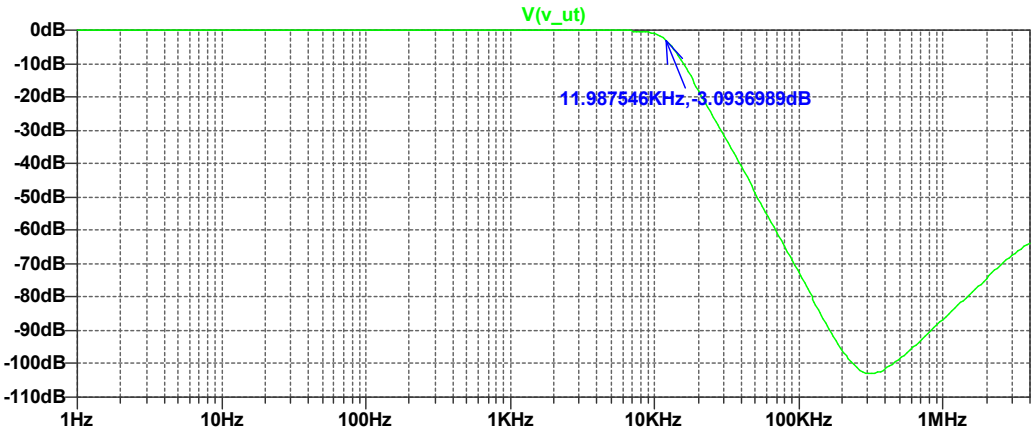

Figure 18: *The frequency response curve for the 4th order LP-filter. The cuttoff frequency is marked. Note that the attenuation reaches a maximum and then decreases again.*

The filter in practice functions as in the simulation wereas the cutoff frequency was located between 11500-12000 Hz.

3 Test and Verification

In order to examine the functionality of the system as a whole, verifying that it was up to specification, an array of measurements were performed. To test both sides of the system, without having to physically build both, we partnered with another lab group and designated our system the receiver and theirs the transmitter.

The first step was to measure the maximum input amplitude that the system can withstand before noticeable clipping occurs on the receiver output. This was done using a 1kHz sine wave and was determined to be

$$U_{IN_{MAX}} = 11\text{V}pp$$

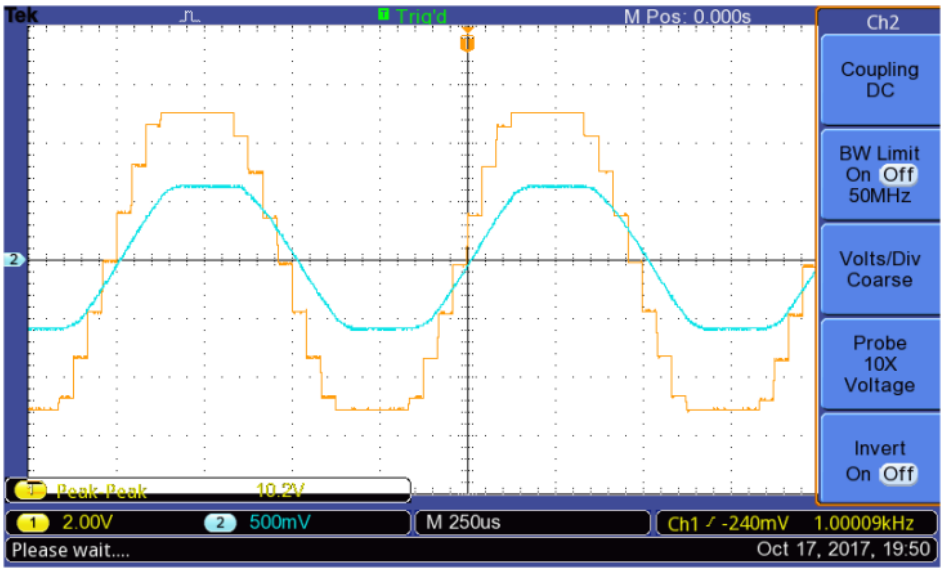

Figure 19: *Noticeable clipping on the receiver.* V_{DA} *in Yellow,* V_{amp2} *in Blue*

Likewise, the minimum input amplitude that could be detected before the receiver output was drowned out by noise was

$$U_{IN_{MIN}} = 400\text{mV}pp$$

This gives the transmission a *dynamic range* of

$$20log\left(\frac{11V}{0.4mV}\right) = 28.79dB$$

Secondly, the outputs from the LP-filters on either side were observed and compared, as shown in figure 20.

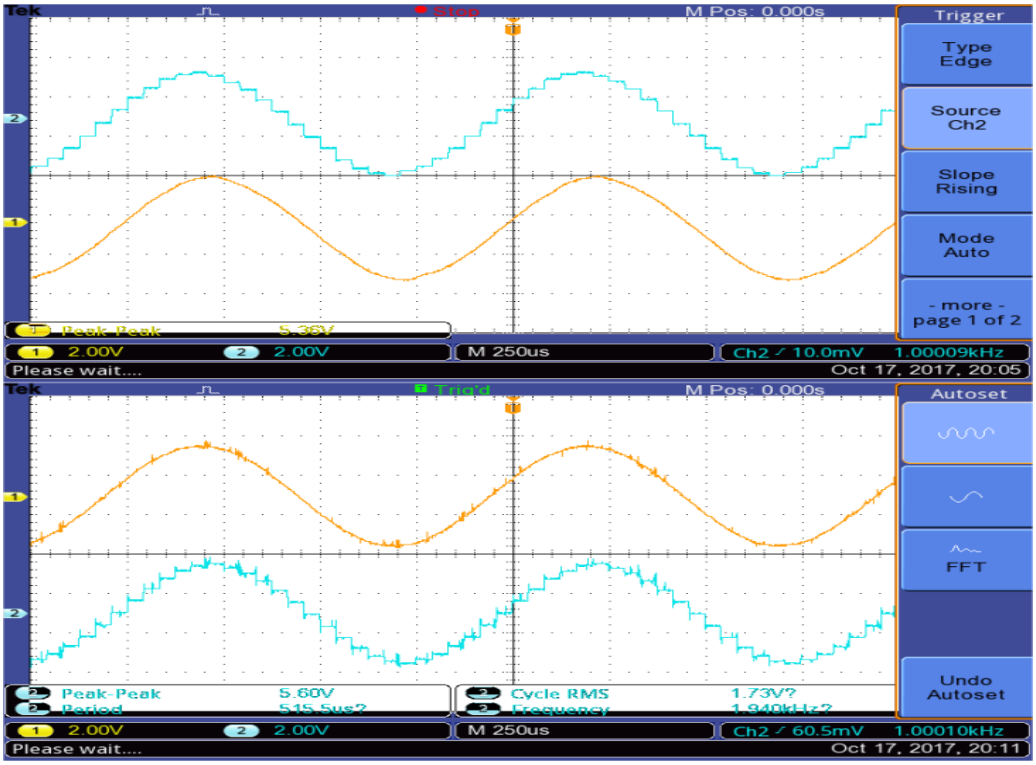

Figure 20: *The smooth lines in each image are the filter outputs. R_x on top, T_x below. Note the high frequency components in the latter are not attenuated.*

Continuing, the system's bandwidth was examined by adjusting the input signal frequency on the transmitter side at a 11Vpp amplitude. The cutoff frequencies where determined to occur at 900Hz and 3900Hz.

The systems resilience to partial transmission failure was also tested by grounding several of the least significant bits on the transmitter side, thus reducing the resolution of the transmitted data. It was unanimously decided by the group that a minimum of 5 bits of resolution is needed to communicate, anything less was rendered completely unintelligible.

A working example of normal speech transmission at the full 8-bit resolution is illustrated in figure 21.

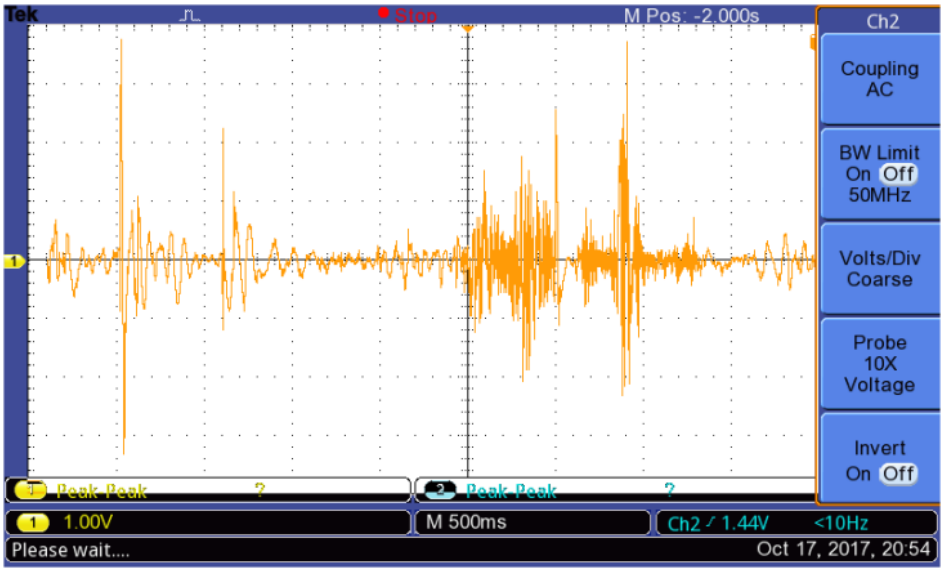

Figure 21: *Input: "jag heter john"*

4 Conclusion

The fact that clipping occurs when $U_{IN} \geq \pm 10Vpp$ is entirely expected due to the fact that it exceeds the range of the ADC, indicating that the ADC itself fulfills its specification. In reality, using a regular microphone such as the one the system was designed for (see chapter 2.7.1) would require an exceptionally loud noise before clipping occurred, unlikely to happen under normal circumstances.

The presence of high frequency components on the receiver side of the system was unanticipated and may contribute distortion. These components may be explained by examining the frequency response of the filter used in figure 18. In the graph one can see how the attenuation reaches a maximum before decreasing again, possibly allowing these high frequency components through. The fact that a breadboard is used with no regards to potentially parasitic effects may also contribute.

An area in which the system clearly did not live up to its specification was in the bandwidth. Whereas the specified bandwidth was 20-12000Hz, the actual measured bandwidth was 900-3900Hz, despite the individual subsystems (on the receiver side) performing as required. As we were partnered with another group for the final system test, we cannot account for the performance of the transmitter. An adequate explanation for this poor performance may require further testing of the individual subsystems on both sides of the transmission.

Nevertheless, this bandwidth was sufficient for regular speech, which was described as sounding "like an ordinary telephone". Decreasing the resolution of the sent data, however, quickly resulted in a loss of intelligibility.

On the whole, despite the presence of some undesirable characteristics, the system is eminently functional when used to transmit spoken audio. All the various subsystems have had their individual performances tested and verified and since there is no inherent bandwidth limiting factor, improving the system's characteristics may simply be a matter of debugging the electrical circuit.

5 Reflection

5.1 Preparatory work

The preparatory work required a large time commitment which in all likelihood exceeded the recommended 95 hours. The assignments were often, however, self explanatory and was easily reflected on what was happening inside of the laboratory.

5.2 Equipment

There were a few time consuming issues with the laboratory equipment that usually turned out to be trivial:

- The oscilloscopes can't seem to differentiate between amplitude and peak-to-peak measurements.
- The bench-top power supplies are current limited and due to inexperience a sudden drop in voltage (due to the current limiting kicking in) was interpreted as a short circuit.
- The passives (resistors especially) had very large tolerances which made it difficult to build the theoretical designs with any precision.
- The handheld DMMs cannot seem to adjust to an amplitude-varying AC signal, unlike the benchtop DMMs.

5.3 Teamwork

Teamwork in the preparation assignment was hardly divided since both had to understand and explain the meaning of the laboratory work. Inside of the laboratory the teamwork was divided into one working on coding the FPGA-chip and the other one setting up the new circuits on the breadboard. Then together solve the assignments or troubleshooting the new part of the system.

5.4 Guidance

The guidance was sufficient and wasn't over exaggerated thus leaving the students to solve the problems with some minor hints.

Appendix

VHDL Code

Code Snippet 6: Complete Transmitter Program

```
1  ----------------------------------------------------------------------------------------------------
2  ------------------------------------------------------ Description
   ↪  ------------------------------------------------
3  ----------------------------------------------------------------------------------------------------
4  -- ADC (Analog voltage in, parallel byte out) with Shift Register serial output.
5
6  -- ADC based on SAR (Successive Approximation Register) with feedback through DAC and
   ↪  subsequent comparator.
7  --
8  -- The SAR is a state machine that is initialised by a sample-pulse (clk_Ts) whereby it
   ↪  sets the MSB of its parallel output HIGH and the rest LOW,
9  -- this is fed into the DAC which outputs an analog voltage corresponding to the input (ie.
   ↪  "10000000" in -> Vmax/2 out),
10 -- this voltage is fed into the comparator and compared with the external input signal. If
   ↪  it is higher than the input signal then the comparator's output is LOW, else HIGH.
11 -- The SAR then uses this information to set the MSB HIGH or LOW before moving onto Bit
   ↪  n-1. State changes in the SAR are triggered by a bit-rate pulse (clk_Tb).
12
13 -- Parallel to serial shift register. LSB out first.
14 -- Shift register must conform to RS/EIA-232 standard and therefore uses 1 START, 1 STOP
   ↪  and 8 DATA bits.
15 -- START is logical LOW (voltage HIGH), STOP is logical HIGH (voltage LOW). There should a
   ↪  positive flank between STOP and START, seen from oscilloscope.
16 ----------------------------------------------------------------------------------------------------
17
18 ----------------------------------------------------------------------------------------------------
19 -- Libraries
20 ----------------------------------------------------------------------------------------------------
21 library ieee;
22 use ieee.std_logic_1164.all;
23 use ieee.std_logic_unsigned.all;
24
25 ----------------------------------------------------------------------------------------------------
26 -- Entity
27 ----------------------------------------------------------------------------------------------------
28 entity Tx_FINAL is
29     port(   CLOCK_50    : in std_logic;     -- 50MHz internal clock.
30             reset       : in std_logic;     -- resets and/or initializes clocked elements.
                ↪  Asynchronous.
31             D_async     : in std_logic;                         -- input from external
                ↪  (asynchronous) comparator.
32             Q           : buffer std_logic_vector(7 downto 0);  -- parallel output from
                ↪  SAR.
33             clk_Ts      : buffer    std_logic;                  -- Generated 'sample-rate'
                ↪  clock.
34             clk_Tb      : buffer    std_logic;                  -- Generated 'bit-rate'
                ↪  clock.
35             LEDR        : out   std_logic_vector(7 downto 0);   -- SAR output display on
                ↪  LEDs.
36             serial      : out   std_logic);                     -- Serial data out.
                ↪  EIA-232.
37 end entity;
```

```
38
39  ----------------------------------------------------------------------------------
40  --Architecture
41  ----------------------------------------------------------------------------------
42  architecture rtl of Tx_FINAL is
43  ----------------------------------------------------------------------------------
44  -- Constants
45  ----------------------------------------------------------------------------------
46      constant clk_Ts_CONST : integer := 2079; -- Clock-gen constants. Determines frequency.
47      constant clk_Ts_5perc_CONST : integer := 1975; -- 5% duty cycle on Ts.
48      constant clk_Tb_CONST : integer := 207;
49
50  ----------------------------------------------------------------------------------
51  -- Signals
52  ----------------------------------------------------------------------------------
53      type StateType is (U,IDLE,SAR7,SAR6,SAR5,SAR4,SAR3,SAR2,SAR1,SAR0,STOP);  -- Enumerated
        ↪  machine states. U used in simulation only.
54      signal state : StateType;
55
56      signal D_semisync   : std_logic;
57      signal D            : std_logic;
58
59      signal cnt_Ts       : std_logic_vector(11 downto 0);
60      signal cnt_Tb       : std_logic_vector(7 downto 0);
61
62      signal SR           : std_logic_vector(8 downto 0); -- shift register containing 8 DATA
        ↪  bits & 1 START bit. ((MSB -> LSB) & START).
63                                                          -- STOP bits are shifted in, so
                                                            ↪  space need not be allocated
                                                            ↪  for them specifically.
64      signal SAR_result   : std_logic_vector(7 downto 0); -- Holds the SAR output after a
        ↪  completed conversion. Multiple processes use this result (at different times) and
        ↪  so a separate signal is needed.
65      signal SR_start     : std_logic;    -- Has ADC completed at least once? Prevents SR
        ↪  from outputting undefined values before first SAR conversion.
66
67  begin
68      ----------------------------------------------------------------------------------
69      -- Architecture: - Top Level
70      ----------------------------------------------------------------------------------
71      serial <= SR(0); -- Serial output from LSB of Shift Register (in accordance with
        ↪  EIA-232).
72
73      ----------------------------------------------------------------------------------
74      -- Process: proc_SAR
75      -- Description: SAR (successive approximation register) state machine.
76      --              State sequence: IDLE -> SAR7 -> ... -> SAR0 -> STOP -> IDLE0
77      -- Input(s) : CLOCK_50, clk_Tb, clk_Ts, reset, D
78      -- Output(s): Q, LEDR
79      -- Internal Signals: state, SAR_result, SR_start
80      ----------------------------------------------------------------------------------
81      proc_SAR : process(CLOCK_50,reset)
82      begin
83          if reset='0' then -- CHECK WHETHER RESET IS INVERTED!
84              state <= IDLE;
85              LEDR <= (others => '0');
```

```
86              SAR_result <= (others => '1');
87              SR_start <= '0';
88
89          elsif rising_edge(CLOCK_50) then
90              if clk_Ts='1' then
91                  Q <= "10000000";     -- MSB set, first approximation is half of maximum
                     ↪  value.
92                  state <= SAR7;       -- Switch to next state in sequence.
93              elsif clk_Tb='1' then
94                  case state is
95                      when SAR7 =>
96                          Q(7) <= D;       -- If V_DA approximation higher than sampled
                             ↪  signal, D is set LOW, and thus also Q(n).
97                          Q(6) <= '1';     -- Add 1/4 of maximum value to approximation, and
                             ↪  so on for subsequent states.
98                          state <= SAR6;
99                      when SAR6 =>
100                         Q(6) <= D;
101                         Q(5) <= '1';
102                         state <= SAR5;
103                     when SAR5 =>
104                         Q(5) <= D;
105                         Q(4) <= '1';
106                         state <= SAR4;
107                     when SAR4 =>
108                         Q(4) <= D;
109                         Q(3) <= '1';
110                         state <= SAR3;
111                     when SAR3 =>
112                         Q(3) <= D;
113                         Q(2) <= '1';
114                         state <= SAR2;
115                     when SAR2 =>
116                         Q(2) <= D;
117                         Q(1) <= '1';
118                         state <= SAR1;
119                     when SAR1 =>
120                         Q(1) <= D;
121                         Q(0) <= '1';
122                         state <= SAR0;
123                     when SAR0 =>
124                         Q(0) <= D;
125                         state <= STOP;
126                     when STOP =>
127                         state <= IDLE;  -- Loop back to IDLE state, until next Ts trigger.
128                         LEDR <= Q;      -- Indicate converion value.
129                         SAR_result <= Q;
130                         SR_start <= '1'; -- First digital conversion completed, shift
                             ↪  register now active.
131                     when IDLE | U =>
132                         -- do nothing
133                 end case;
134             end if;
135         end if;
136     end process;
137
```

```
138     ---------------------------------------------------------------------------------
139     -- Process: proc_comparator_read
140     -- Description: Makes the asynchronous comparator output (signal D) synchronous in
        ↪  order to mitigate metastability issues.
141     --              Does this by sychronously reading D into an intermediate flip-flop,
        ↪  'D_semisync'.
142     -- Input(s) : D_async
143     -- Output(s): D
144     -- Internal Signals: D_semisync
145     ---------------------------------------------------------------------------------
146     proc_comparator_read : process(CLOCK_50)
147     begin
148         if rising_edge(CLOCK_50) then
149             D_semisync <= D_async;
150             D <= D_semisync;
151         end if;
152     end process;
153
154     ---------------------------------------------------------------------------------
155     -- Process: proc_Ts
156     -- Description: Generates the sample-rate clock, clk_Ts
157     -- Input(s) : CLOCK_50, reset
158     -- Output(s): clk_Ts
159     -- Internal Signals: cnt_Ts
160     ---------------------------------------------------------------------------------
161     proc_Ts: process(CLOCK_50,reset) -- reset is asynchronous, and so process must be
        ↪  sensitive to it!
162     begin
163         if reset='0' then   -- CHECK IF LOGIC INVERTED!
164             cnt_Ts <= (others => '0');  --Clear counter and pull clock output low.
165             clk_Ts <= '0';
166         elsif rising_edge(CLOCK_50) then
167             if cnt_Ts = clk_Ts_5perc_CONST then
168                 clk_Ts <= '1';
169                 cnt_Ts <= cnt_Ts + 1;
170             elsif cnt_Ts = clk_Ts_CONST then
171                 clk_Ts <= '0';
172                 cnt_Ts <= (others => '0');
173             else
174                 cnt_Ts <= cnt_Ts + 1;
175             end if;
176         end if;
177     end process;
178
179     ---------------------------------------------------------------------------------
180     -- Process: proc_Tb
181     -- Description: Generates the bit-rate clock, clk_Tb
182     -- Input(s) : CLOCK_50, reset
183     -- Output(s): clk_Tb
184     -- Internal Signals: cnt_Tb,
185     ---------------------------------------------------------------------------------
186     proc_Tb: process(CLOCK_50,reset)
187     begin
188         if reset='0' then   -- CHECK IF LOGIC INVERTED!
189             cnt_Tb <= (others => '0');  --Clear counter and pull clock output low.
190             clk_Tb <= '0';
```

```
191         elsif rising_edge(CLOCK_50) then
192             if cnt_Tb = (clk_Tb_CONST - 1) then
193                 clk_Tb <= '1';
194                 cnt_Tb <= cnt_Tb + 1;
195             elsif cnt_Tb = clk_Tb_CONST then
196                 clk_Tb <= '0';
197                 cnt_Tb <= (others => '0');
198             else
199                 cnt_Tb <= cnt_Tb + 1;
200             end if;
201         end if;
202     end process;
203 
204     ---------------------------------------------------------------------------------
205     -- Process: proc_shift_register
206     -- Description: Parallel in -> serial out, LSB first.
207     -- Input(s) : CLOCK_50, reset, Q, clk_Ts, clk_Tb
208     -- Output(s): serial
209     -- Internal Signals: SR, SAR_result, SR_start
210     ---------------------------------------------------------------------------------
211     proc_shift_register: process(CLOCK_50, reset)
212     begin
213         if reset='0' then
214             SR <= (others => '1'); -- Init. SR with STOP bits.
215         elsif rising_edge(CLOCK_50) then
216             if SR_start='1' then    -- Shift Register activated?
217                 if clk_Ts='1' then
218                     SR <= SAR_result & '0'; -- Load Shift Register with DATA & START bit
                        ↪  ((MSB..LSB) & START).
219                 elsif clk_Tb='1' then
220                     SR <=  '1' & SR(8 downto 1);    -- Shift data right, Shift in STOP
                        ↪  bit.
221                 end if;
222             end if;
223         end if;
224     end process;
225 end architecture;
```

Code Snippet 7: Complete Receiver Program

```
1  ---------------------------------------------------- Description
   ↪  ----------------------------------------------------
2  -- RS-232 receiver.
3
4  -- Works at 12 kbaud.
5  -- Uses 'clock recovery' to synchronize timing. (Symbolsynkronisering)
6  ------------------------------------------------------------------------------------------------------------
7
8  ------------------------------------------------------------------------------------------------------------
9  -- Libraries
10 ------------------------------------------------------------------------------------------------------------
11 library IEEE;
12 use IEEE.STD_LOGIC_1164.all;
13 use IEEE.STD_LOGIC_UNSIGNED.all;
14
15 ------------------------------------------------------------------------------------------------------------
16 -- Entity
17 ------------------------------------------------------------------------------------------------------------
18 entity Rx_FINAL is
19     port(   reset, CLOCK_50 : in std_logic;
20             serial_async : in std_logic; -- Serial in.
21             parallel : buffer std_logic_vector(7 downto 0);
22             LEDR : out std_logic_vector(7 downto 0);
23
24             DTx : out std_logic);
25 end entity;
26
27 ------------------------------------------------------------------------------------------------------------
28 ---------------------------------------------------- Architecture
   ↪  ----------------------------------------------------
29 ------------------------------------------------------------------------------------------------------------
30 architecture arch of Rx_FINAL is
31     signal cnt : std_logic_vector(7 downto 0);
32     signal Q, next_Q : std_logic_vector(7 downto 0);
33     signal next_parallel : std_logic_vector(7 downto 0);
34     signal serial_semisync, serial : std_logic;
35
36     type stateType is (U,STOP,START,S0,S1,S2,S3,S4,S5,S6,S7);
37     signal state, next_state : stateType;
38 begin
39
40     LEDR <= parallel;
41     DTx <= serial;
42
43
       ↪  ------------------------------------------------------------------------------------------------
44     -- Process: proc_Rx_sync
45     -- Description: Synchronous part of serial->parallel Mealy machine.
46
47     -- Input(s) : reset, CLOCK_50, serial
48     -- Output(s): parallel
49     -- Internal Signals: cnt, state, next_state, next_parallel, Q, next_Q
```

```
50
    ↪  ------------------------------------------------------------------------------------------
51      proc_Rx_sync : process(CLOCK_50, reset)
52      begin
53          if reset='0' then
54              state <= STOP;
55              Q <= (others => '0');
56              cnt <= (others => '0');
57          elsif rising_edge(CLOCK_50) then
58              cnt <= cnt + 1;
59              if (state = STOP AND serial = '1') then
60                  cnt <= (others => '0');
61
62              elsif cnt = 103 then
63                  state <= next_state;
64                  Q <= next_Q;
65                  parallel <= next_parallel;
66              elsif cnt = 207 then
67                  cnt <= (others => '0');
68              end if;
69          end if;
70      end process;
71
    ↪  ------------------------------------------------------------------------------------------
72      -- Process: proc_Rx_comb
73      -- Description: Combinational part of serial->parallel Mealy machine.
74
75      -- Input(s) :
76      -- Output(s):
77      -- Internal Signals: state, next_state, next_parallel, Q, next_Q, serial, S(n)
78
    ↪  ------------------------------------------------------------------------------------------
79      proc_Rx_comb : process(state, serial, Q)
80      begin
81          next_Q <= Q;
82              case state is
83                  when STOP =>
84                      next_state <= START;
85                  when START =>
86                      next_state <= S0;
87                      next_Q(0) <= serial;
88                  when S0 =>
89                      next_state <= S1;
90                      next_Q(1) <= serial;
91                  when S1 =>
92                      next_state <= S2;
93                      next_Q(2) <= serial;
94                  when S2 =>
95                      next_state <= S3;
96                      next_Q(3) <= serial;
97                  when S3 =>
98                      next_state <= S4;
99                      next_Q(4) <= serial;
100                 when S4 =>
101                     next_state <= S5;
102                     next_Q(5) <= serial;
```

```
                when S5 =>
                    next_state <= S6;
                    next_Q(6) <= serial;
                when S6 =>
                    next_state <= S7;
                    next_Q(7) <= serial;
                when S7 =>
                    next_parallel <= Q;
                    if serial='0' then
                        next_state <= S7;
                    else
                        next_state <= STOP;
                    end if;
                when others =>
                    -- undefined state
                    next_state <= STOP;
            end case;
    end process;
    \newpage

    ↪  ----------------------------------------------------------------------------------------------------
    -- Process: proc_serial_read
    -- Description: Makes the asynchronous input signal synchronous
    --              in order to mitigate metastability issues.
    --              Does this by sychronously reading the signal into
    --              an intermediate flip-flop.

    -- Input(s) : serial_async
    -- Output(s): serial
    -- Internal Signals: serial_semisync

    ↪  ----------------------------------------------------------------------------------------------------
    proc_serial_read : process(serial_async, CLOCK_50)
    begin
        if rising_edge(CLOCK_50) then
            serial_semisync <= serial_async;
            serial <= serial_semisync;
            end if;
    end process;
end architecture;
```

Schematics

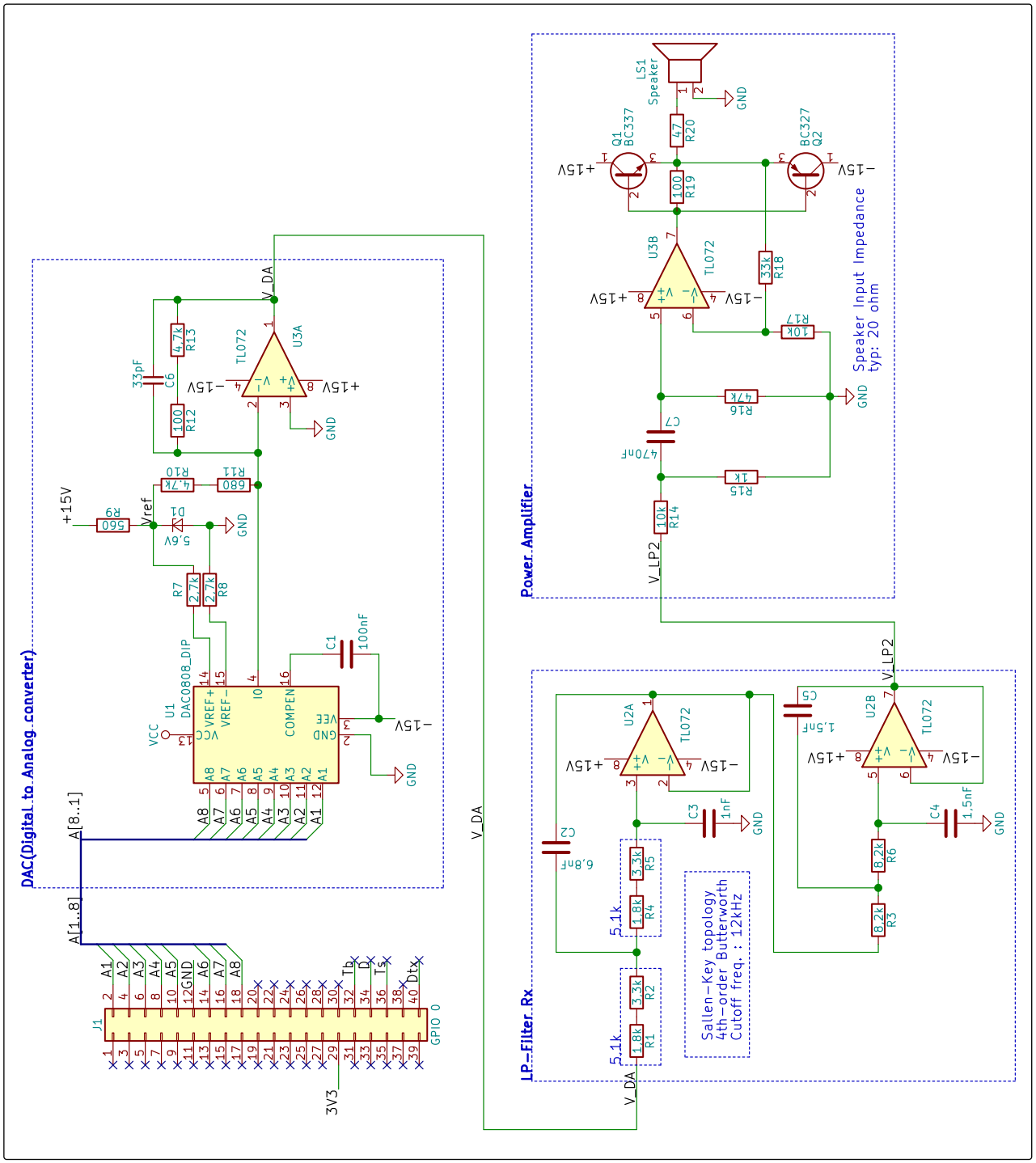

Figure 22: *Serial receiver*

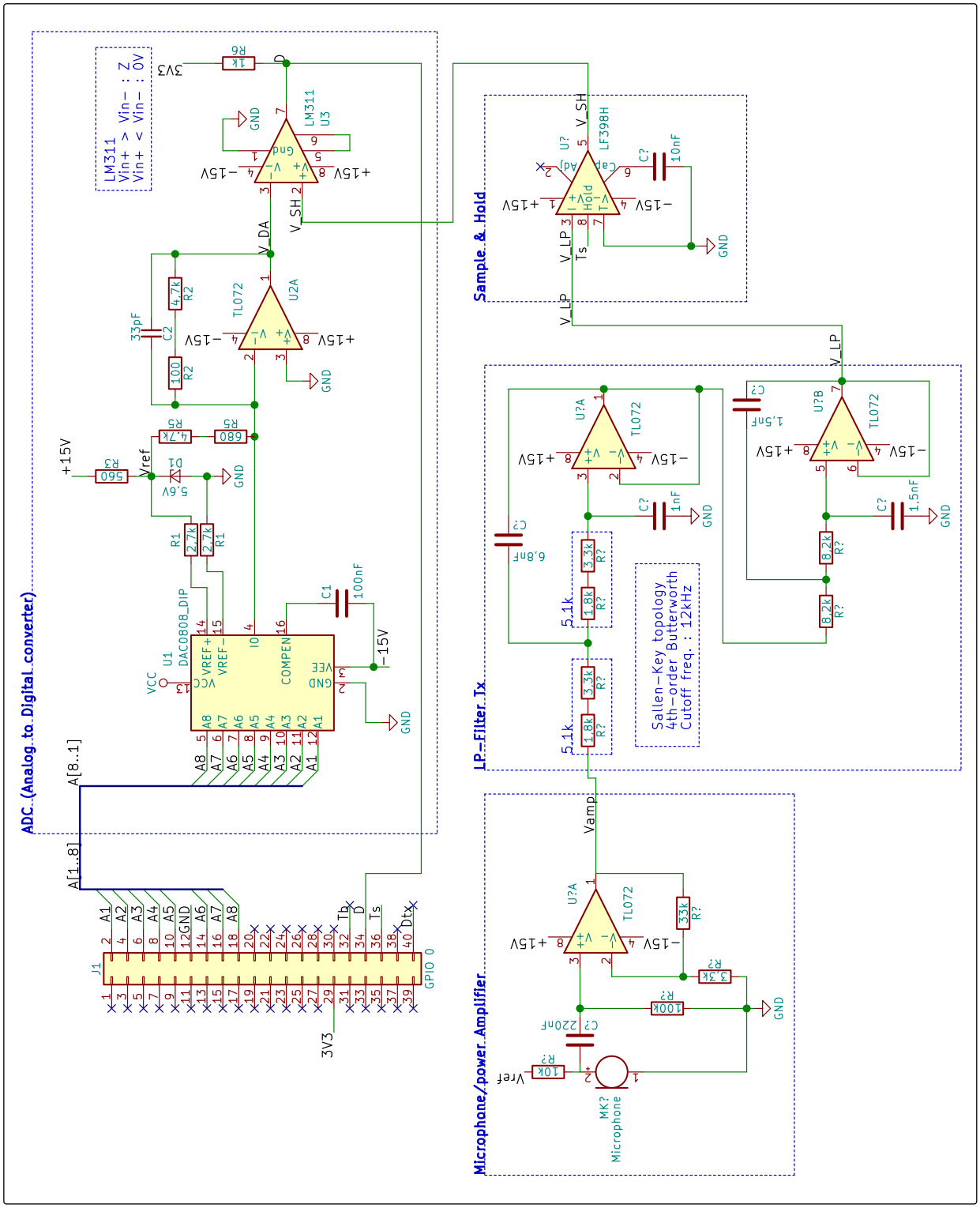

Figure 23: *Serial transmitter*