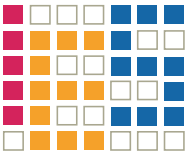
Design and Development of a Custom ASIC Test PCB for Ultrasonic Communication

Course Semester: SS 2023
Supervisor: Dominic Korner
Department: Integrierte Elektronische Systeme (IES) | **Prof. Dr.-Ing. Klaus Hofmann**
Project Seminar Report in the department of Electrical Engineering and Information Technology by
Malte Nilges, 2704362
Date of submission: September 15, 2024

Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Electrical Engineering and Information Technology Department



Contents

1 Introduction

The work presented in this report is part of a project seminar conducted in the Integrated Electronic Systems (IES) lab at the Technical University of Darmstadt. This chapter gives an overview of the purpose and objectives of the project seminar and some physical background knowledge relating to the project.

1.1 Overview

Systems that perceive and evaluate their targeted environment rely on the usage of sensors to acquire the desired data. For many sensing applications, including the monitoring of structures, machinery, the environment, or human health, wireless sensor communication is an essential part of such systems as it is often the only feasible way to share acquired data to other systems or human operators [1].

There are different methods of wireless sensor communication, which differ in frequency, range, data rate and energy consumption. Some examples for commonly used technologies are Bluetooth, ZigBee, WLAN or ultrasonic communication. Common to all is that energy is required in order to communicate. In scenarios where the usage of batteries is not possible, that energy needs to be harvested from the surroundings, e.g. from electrical signals transmitted to the system or from mechanical forces exerted on the system. This method is called energy harvesting or wireless power harvesting (WPH). Depending on the environment, different communication and WPH techniques are suited. Criteria may be the environment, the available space, the density of the transmission medium and the transmission range [2].

The objective of this work is to enable communication and energy transmission using ultrasound, which means sound waves with frequencies greater than the audible frequency range of about $20\,\mathrm{kHz}$. These mechanical vibrations are transported through a transmission medium such as air or metal. Piezoelectric transducers are capable to both transmit and receive ultrasound waves and therefore the method of choice for the targeted functionality. Such components are constructed of materials that change in shape when an electric field is applied (piezoelectric actuator) and generate an electric fields when exposed to a deforming mechanical force (piezoelectric sensor). As this process is reversible, piezoelectric materials can act both as sensors and actuators [3].

In order to perform a two-way communication, the system designed in this work must be able to connect the piezo element to a switched voltage-controlled voltage source (VCVS) in order to apply an electric field resulting in ultrasonic vibrations, as well as receive, amplify and process the electric signals that are induced by the piezo element when exposed to ultrasonic vibrations. The input signal processing needs to 1) convert the amplified signal into a digital code in order to detect its amplitude and 2) determine the frequency with a sufficient accuracy in order to detect changes for frequency modulated signals. The intended solution for this task is the usage of an analog-to-digital converter (ADC) and a comparator. A further requirement is a system for communicating with an external machine in order to receive commands and transmit sampled data.

The implementation of this abstract system is described in chapter 2.

1.2 Piezoelectric effect

The phenomenon of an electric field being generated in a material when it is subjected to mechanical stress or strain is known as the piezoelectric effect. Its counterpart is the inverse piezoelectric effect due to which a mechanical strain or deformation occurs in a material when it is subjected to an electric field [4, 5].

The piezoelectric effect is only exhibited by crystalline materials which either lack a center of symmetry in their atomic structure or exhibit a dipole moment that can be polarized (so-called ferroelectrics). Crystals are materials that have a uniform atomic structure called lattice which extends in all directions of the material. On a macroscopic level, the lattice of a single crystal is visible by the orientation of its flat faces. The counterpart to crystalline structures are amorphous structures, which have no periodicity in their atomic arrangement. In semiconductor technology, the crystalline structures of materials such as silicon are deliberately disrupted by adding impurities, called dopants, in order to create semiconductors such as diodes.

Piezoelectric materials rely on non-centrosymmetric crystalline structures, meaning that the ions are distributed without a center of symmetry. In such materials, the polarization changes as a force such as compression, expansion, tension or torsion, depending on the crystal lattice, is applied. The force has to be directed for a change in polarization, as the lattice is not asymmetrical in every dimension. Inside the material, facing ions may cancel may cancel each other, but the change of the polarization appears as a positive and negative charge on the surfaces generating an electrostatic output voltage. Due to internal leakage paths formed by impurities, the voltage drops after some time[5, 6].

Typical examples for piezoelectric materials include monocrystalline materials like Rochelle salt or quartz, but also polycrystalline ferroelectric ceramics such as lead zirkonate titanate (PZT). Materials of latter category need to be polarized first. This process is performed using a high electric field at temperatures close to the Curie-temperature at which the ferroelectric properties of the material would disappear. The otherwise randomly orientated dipoles align with the electric field forming a permanent dipole and the material becomes piezoelectric [6, 7].

Figure 1.1 shows the piezoelectric effect on a molecular model. As a force is applied to the molecule, the positive and negative gravity centers of the molecules are seperated, generating a dipole and thus an electric field.

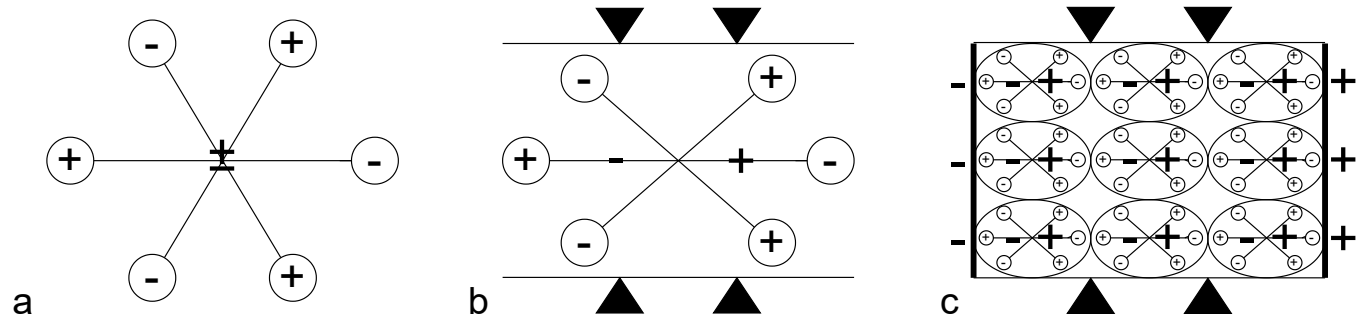


Figure 1.1: Model of the piezoelectric effect: a) unperturbed molecule; b) molecule exposed to force; c) surface of a material exposed to force

Piezo elements have one or multiple resonance frequencies, at which their induced electric signal resulting from a mechanical force is highest and vice versa. The resonance effect is caused by the ability to transfer energy between the storage modes of electric and vibrational energy as described above. At resonance, the damping is lowest, hence the exerted energy is stored and rising with each oscillation. As realistic systems have always some kind of damping e.g. in form of friction, the maximum stored energy is limited, but yet it can be high enough to destroy the material e.g. due to tension caused by movements or due to breakdown voltages. The damping can be described by the quality factor Q, which is defined as the resonance frequency divided by the resonance width, being the bandwidth, over which the power is more than half compared to that of resonance. A high Q means less damping but also a lower relative resonance width, making the system more selective. For optimal energy transmission, the piezo element should be driven near its resonance frequency [5, 8].

2 Implementation

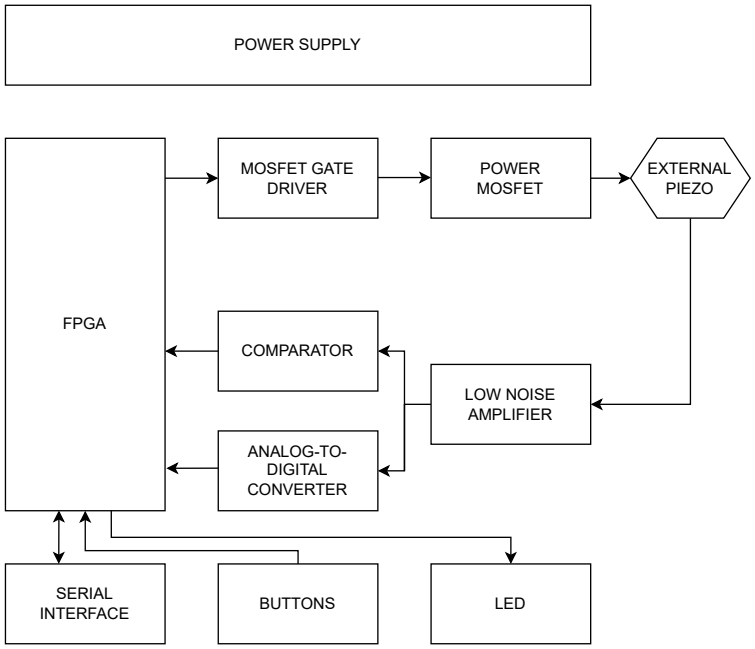


Figure 2.1: System overview as data flow diagram

The test platform for ultrasonic communication was implemented according to the objectives specified in section 1.1. The implementation consists of two parts, the hardware platform, executed as a custom printed circuit board (PCB) design, and the software platform, containing the logic to control the circuit and to send, receive and process data. Figure 2.1 shows a data flow diagram of the resulting system, the controlling of each system part is performed by the field programmable gate array (FPGA).

The following sections describe the designed hardware and software. Please refer to chapter 3 for the results and a discussion of the proposed system.

2.1 Hardware platform

Since a platform for controlling a piezo element was already available, it was used as the basis for all modifications and extensions. The existing hardware platform consisted of a driver stage to use the piezo element as an actuator, an amplifier stage in case of data reception and a connector for a Digilent CMOD S7 FPGA development board to control the driver stage and to adjust the amplifier stage.

This basic structure was retained, but the driver stage was replaced to meet the changed output power requirements and an ADC and comparator stage was added to process and forward received data in digital form. The PCB design was performed with the use of the circuit design tool KiCAD 6.0.

2.1.1 Piezo driver stage

The driver stage is responsible for properly exciting the piezo transducer and was designed with the following specifications in mind:

- $1\,\mathrm{MHz}$ to $2\,\mathrm{MHz}$ switching frequency
- $38\,\mathrm{V}$ input voltage to the piezo transducer
- $100\,\Omega$ equivalent load

The original driver stage was based on a Microchip TC6320, a complementary N- and P-channel metal-oxide-semiconductor-field-effect transistor (MOSFET) pair supporting voltages of $\pm 100\,\mathrm{V}$ and currents of at least $2\,\mathrm{A}$ [9]. Despite being classified as a power transistor pair, the circuit has the disadvantage of a rather high $R_{DS,ON}$ resistance, which, depending on the applied gate-to-source voltage, is at least $7\,\Omega$ in the case of the NMOS and at least $8\,\Omega$ in case of the PMOS and must therefore be cooled accordingly for larger currents. In order to circumvent the thermal limitation of the existing driver stage, it was revised and improved. Two separate onsemi FDD1600N10ALZ N-channel MOSFETs [10] with an $R_{DS,ON}$ resistance of $150\,\mathrm{m\Omega}$ and similar input capacitance C_{iss} of $169\,\mathrm{pF}$ are providing lower conduction losses while maintaining the same switching loss.

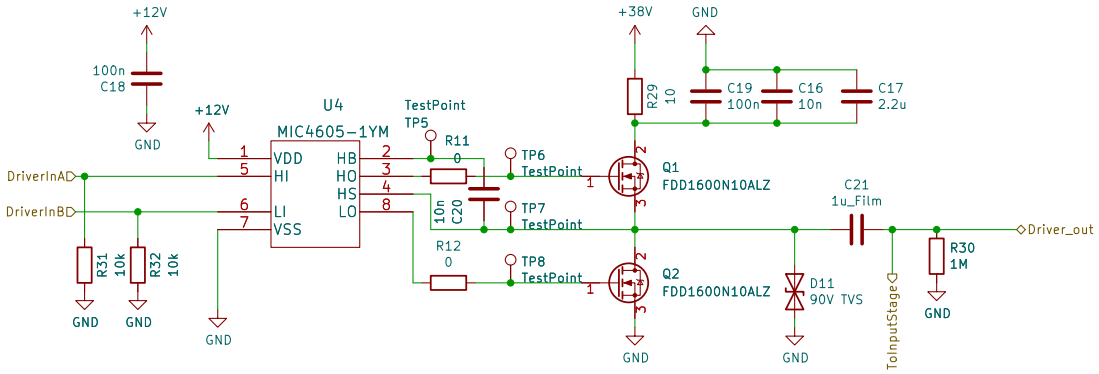


Figure 2.2: Piezo driver stage

The gate driver is a Microchip MIC4604 featuring two outputs and an integrated bootstrap diode for driving N-channel MOSFETs in half-bridge configuration [11]. An external bootstrap capacitor is used to boost the voltage at the high-side gate of the MOSFET. The gate driver is controlled by two separate inputs, the timing to avoid shoot-through, meaning a short circuit from V_{in} to ground due to simultaneously active MOSFETs, is performed by software. In case of data reception, both half-bridge MOSFETs can be switched off in order to avoid a load on the signal source and cross-talk in the received signal.

The output of the signal to the piezo is filtered by a RC high-pass. The coupling capacitor was designed with $1\,\mu\mathrm{F}$, the resistor with $100\,\mathrm{M\Omega}$.

2.1.2 Input stage and amplifier

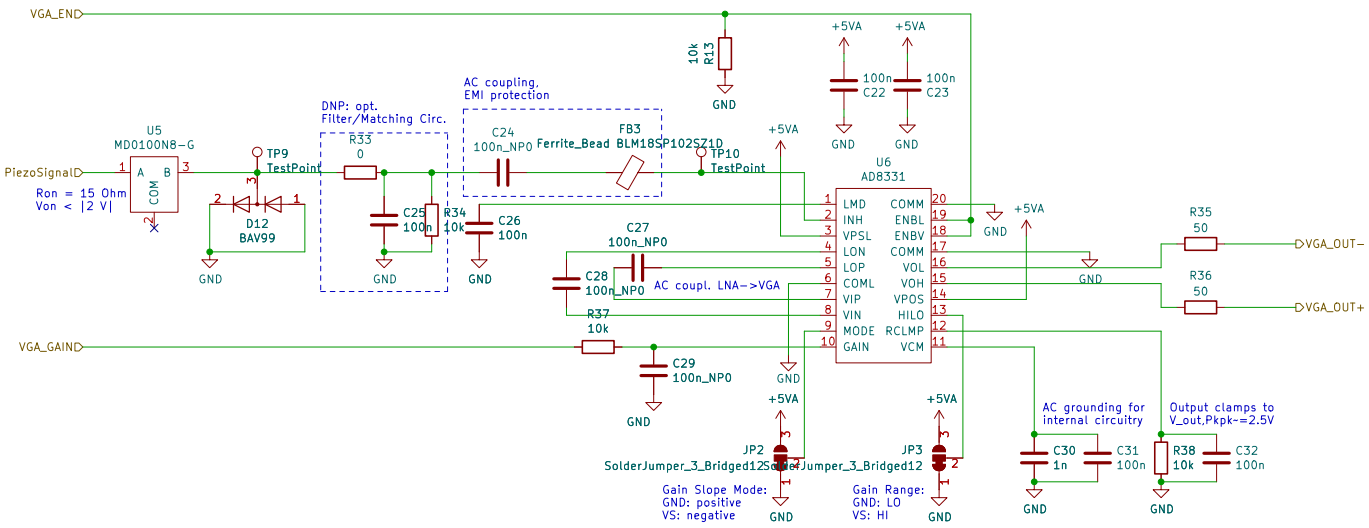


Figure 2.3: Input stage and amplifier schematic

The input stage was largely taken over from the original version. Since the same piezo transducer is used for transmitting and receiving and thus the transmit path is electrically connected to the receive path, a Microchip MD0100 T/R switch was used which has been explicitly designed for ultrasonic applications [12]. The switch protects the low noise amplifier (LNA) from high input voltages that would occur in the event of a transmit signal. Once the voltage between input and output exceeds $\pm 2\,\mathrm{V}$, the switch turns off, limiting the current to $200\,\mu\mathrm{A}$. The switching time is $20\,\mathrm{ns}$, for faster transients and to limit the LNA input, anti-parallel Schottky diodes with short reverse recovery times t_{rr} are used in the form of the Vishay BAV99 integrated circuit [13].

In series to the LNA input is an additional passive filter network which can be modified to adjust the frequency components to be amplified, although it is not placed in order to avoid additional noise from being amplified. A coupling capacitor and a ferrite provide for the filtering of DC components and high-frequency interference signals, respectively.

The amplifier itself is an Analog Devices AD8331 [14]. This circuit connects a LNA, which has a fixed gain of $19\,\mathrm{dB}$, to a variable gain amplifier (VGA) with an adjustable gain between $-4.5\,\mathrm{dB}$ and $43.5\,\mathrm{dB}$ in LO gain mode and between $7.5\,\mathrm{dB}$ and $55.5\,\mathrm{dB}$ in HI gain mode. The $3\,\mathrm{dB}$ bandwidth is $120\,\mathrm{MHz}$ and is thus well suited for amplification of high frequency signals. The gain is set by the gain pin and a 4-bit digital-to-analog converter (DAC), implemented as a resistor network, provides the intended voltage level between $0\,\mathrm{V}$ and $1\,\mathrm{V}$. The differential peak-to-peak output is limited to $2.5\,\mathrm{V}$ by the Rclmp pin to protect downstream components and is terminated with $50\,\Omega$ resistors.

The amplified signal can then either be output via the BNC connectors or, alternatively, processed further in digital form using the analog-to-digital converter or the comparator.

2.1.3 Analog-to-digital converter

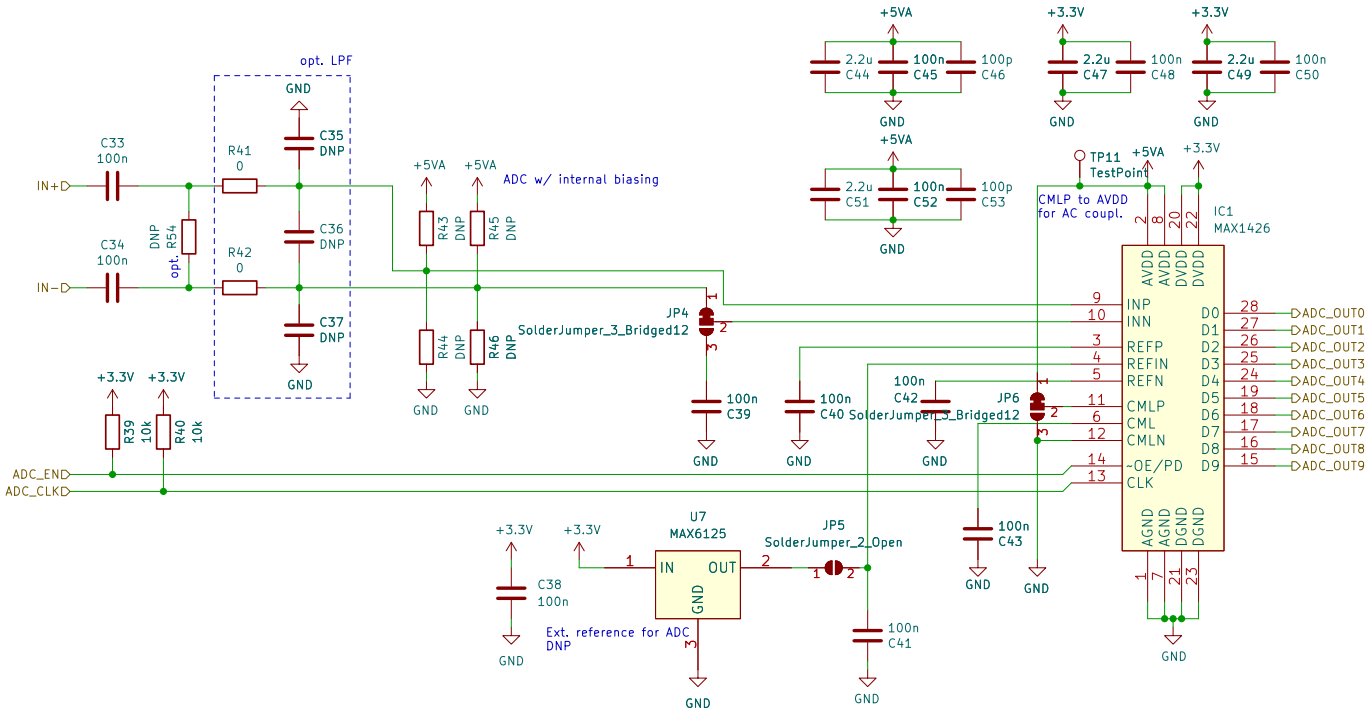


Figure 2.4: ADC schematic

Analog Devices' MAX1426 analog-to-digital converter is a 10 bit, 10 megasamples per second pipeline ADC [15]. It supports a differential signal that is superimposed in the case of AC coupling using an integrated common-mode voltage. This bias voltage is $2.25\,\mathrm{V}$ at the CML pin, which is derived from an internal or external voltage reference of $2.5\,\mathrm{V}$. The reference voltages required for comparison are $3.25\,\mathrm{V}$ (REFP pin) and $1.25\,\mathrm{V}$ (REFN pin). Thus, the input range is $\pm 2\,\mathrm{V}$. Alternatively, the bias voltage and reference voltages can be specified by applying appropriate voltage levels to the pins themselves, although this is not implemented in the board design.

The digital output signal is provided via a parallel interface in two's complement. Because of the separate supply voltages for the analog and digital sections, the output level is set to $3.3\,\mathrm{V}$, eliminating the need to convert the logic level externally. The latency between input and output signal is 5.5 clock cycles.

2.1.4 Comparator

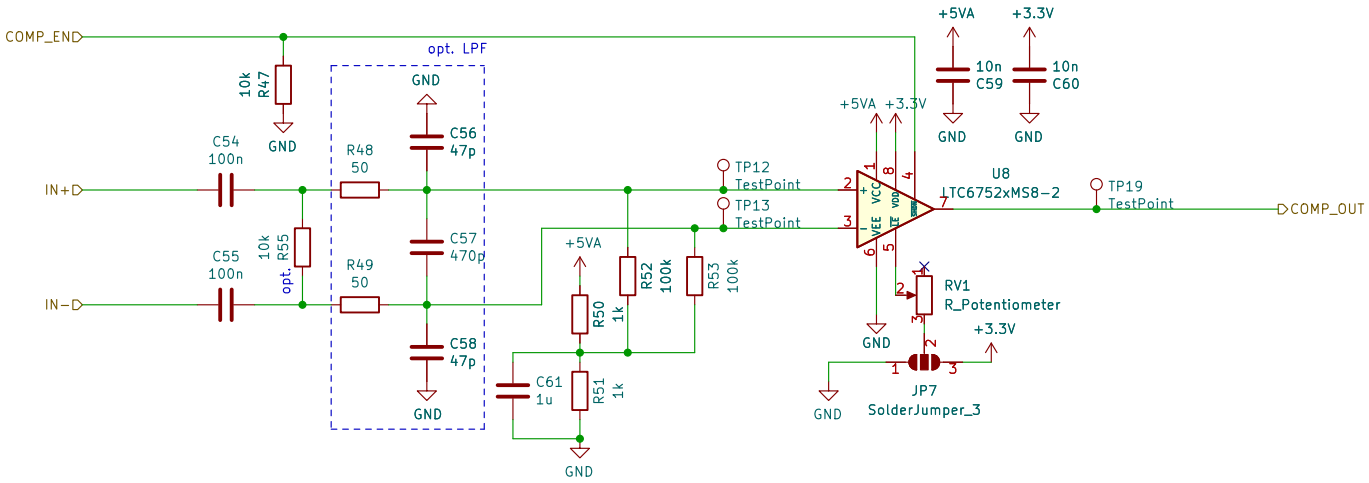


Figure 2.5: Comparator schematic

The comparator is an Analog Devices LTC6752 featuring toggle rates of up to $280\,\mathrm{MHz}$ and low propagation delay as well as low rise and fall times in the range of a few nanoseconds [16]. It is supposed to enable the possibility of transmitting data with frequency modulation by counting the number of the respective FPGA clock cycles in order to determine the transmission of a '0' or a '1'. The device features differential inputs and separate supply rails for the input and output stage. After the AC-coupling of the signal coming from the VGA, the differential signal is filtered with a passive band-pass filter which suppresses noise from unwanted frequencies. The $3\,\mathrm{dB}$ corner frequencies are calculated as follows:

$$f_{3dB,CM}^{HPF} = \frac{1}{2\pi C_{in} R_{CM}} \approx \frac{1}{2\pi \cdot 100\,\mathrm{nF} \cdot 100\,\mathrm{k\Omega}} = 15.92\,\mathrm{Hz}$$

$$f_{3dB,CM}^{LPF} = \frac{1}{2\pi R_{in} C_{CM}} \approx \frac{1}{2\pi \cdot 50\,\Omega \cdot 47\,\mathrm{pF}} = 67.73\,\mathrm{MHz}$$

$$f_{3dB,DM}^{HPF} = \frac{1}{2\pi \cdot C_{in}\left(\frac{1}{2}R_{DM} \| R_{CM}\right)} \approx \frac{1}{2\pi \cdot 100\,\mathrm{nF} \cdot (5\,\mathrm{k\Omega} \| 100\,\mathrm{k\Omega})} = 334.23\,\mathrm{Hz}$$

$$f_{3dB,DM}^{LPF} = \frac{1}{2\pi \cdot R_{in}\left(2C_{DM} + C_{CM}\right)} \approx \frac{1}{2\pi \cdot 50\,\Omega \cdot (940\,\mathrm{pF} + 47\,\mathrm{pF})} = 3.23\,\mathrm{MHz}$$

After filtering, the AC signal is biased to a DC operating point of $2.5\,\mathrm{V}$, generated by a voltage divider of the $5\,\mathrm{V}$ analog supply rail [17].

In order to further minimize the effect of noise at the cost of latency, the comparator provides an adjustable hysteresis, meaning different tripping points for a positive and negative input voltage difference in order to change the output. The internal hysteresis is set by using an external resistor between the input of the $\overline{\mathrm{LE}}$/HYST-Pin and GND. The default value is $5\,\mathrm{mV}$ and increases as the voltage of the pin is being decreased from its default open loop value of $1.25\,\mathrm{V}$. By pulling the $\overline{\mathrm{LE}}$/HYST-Pin to VCC the hysteresis could also be eliminated. As we are dealing with amplified and comparatively slow signals, the hysteresis is set to around $20\,\mathrm{mV}$ using a potentiometer in order to get a steady signal at the output when there is no substantial difference voltage at the input, i.e. in the case of no signal.

2.1.5 FPGA and I/O

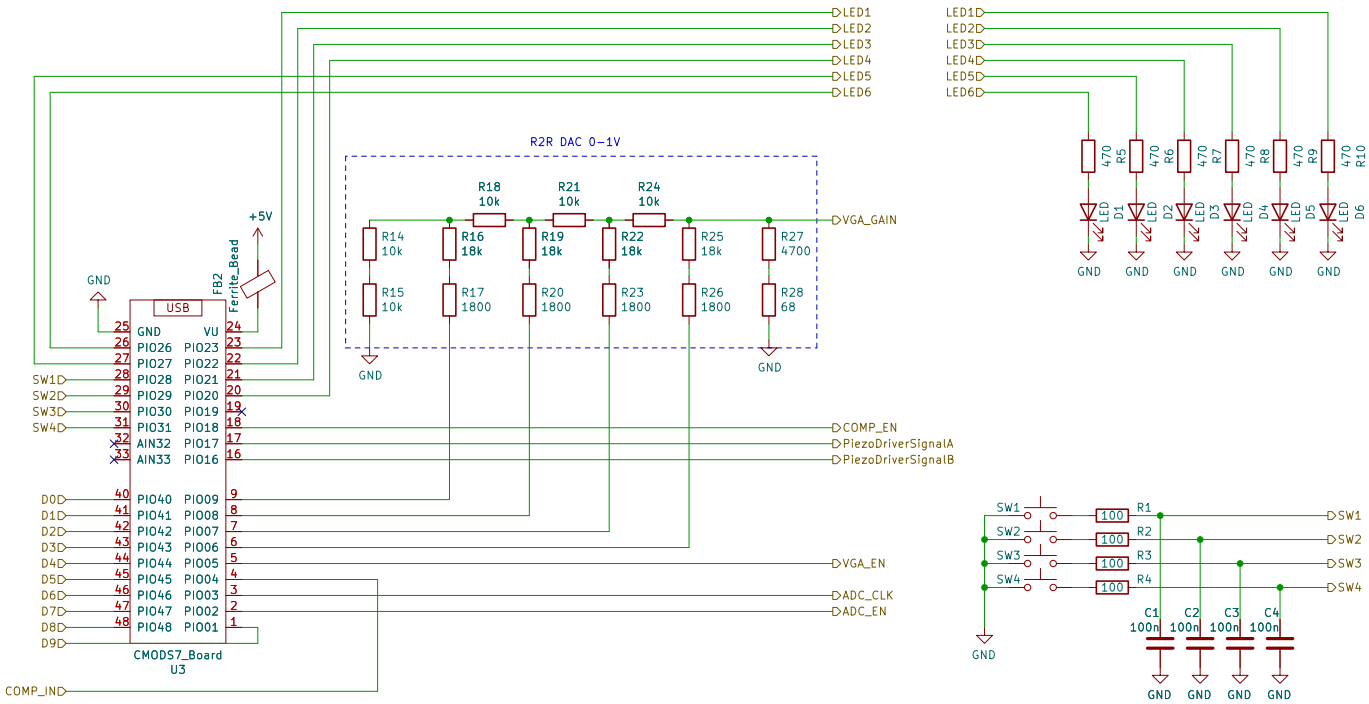


Figure 2.6: FPGA and I/O schematic

In order to reduce the individual component count on the PCB and to enable easier replacements, a breadboard-able FPGA module in form of the CMOD S7 was chosen to provide the logic functionality for the communication platform [18]. It features a Xilinx Spartan7 XC7S25 FPGA on a 48-pin dual in-line package (DIP) form factor board driven by a $12\,\mathrm{MHz}$ external clock and a FTDI FT2232HQ USB-UART bridge for communication with a host computer. The FPGA module requires a $5\,\mathrm{V}$ supply voltage, which is either provided by an USB connection or using the $5\,\mathrm{V}$ digital supply rail, which is decoupled and filtered in order to reduce effects on the analog rail sourced from the same low-dropout voltage regulator (LDO).

Of the 36 pins of the module, 32 are usable as digital inputs or outputs, not counting the pins of the additional PMOD header on the board. The pins are connected to the FPGA via $240\,\Omega$ series resistors, limiting the switching speed to $25\,\mathrm{MHz}$, which is sufficient for driving and polling all circuits on the board. Besides the control of the platform over serial communication, some basic functionality should be accessible without the need of an external computer, hence four switches have been placed on the board in addition to the two switches included on the CMOD S7. These external buttons are pulled to VDD internally and connect the pins to GND in case of a button press. The debouncing of a button press is performed in logic, but the board provides the option to replace the arbitrarily chosen RC network in order to perform the debouncing via hardware. For basic output, six light emitting diodes (LEDs) have been placed on the board, which together with the LEDs of the module sums up to a total of ten single color LEDs plus one RGB LED.

The VGA requires a voltage between $0\,\mathrm{V}$ and $1\,\mathrm{V}$ in order to control the gain of the differential output stage. This is achieved by a 4 bit R-2R-Ladder DAC with a voltage divider at its output, giving the option to control the gain from logic if needed. The DAC features 16 steps with a step size around $66.6\,\mathrm{mV}$, with pin PIO9

controlling the least significant bit (LSB) and PIO6 controlling the most significant bit (MSB). The resistor values have been reused from the original design and simulated in LTSpice for verification.

Besides the input/output (I/O) and the 4 bit DAC, the FPGA provides the enable signals of the VGA, the ADC and the comparator, the clock for the ADC and the high-side and low-side signals for the piezo driver stage and receives the parallel 10 bit output signals from the ADC and the 1 bit comparator output.

2.1.6 Power supply

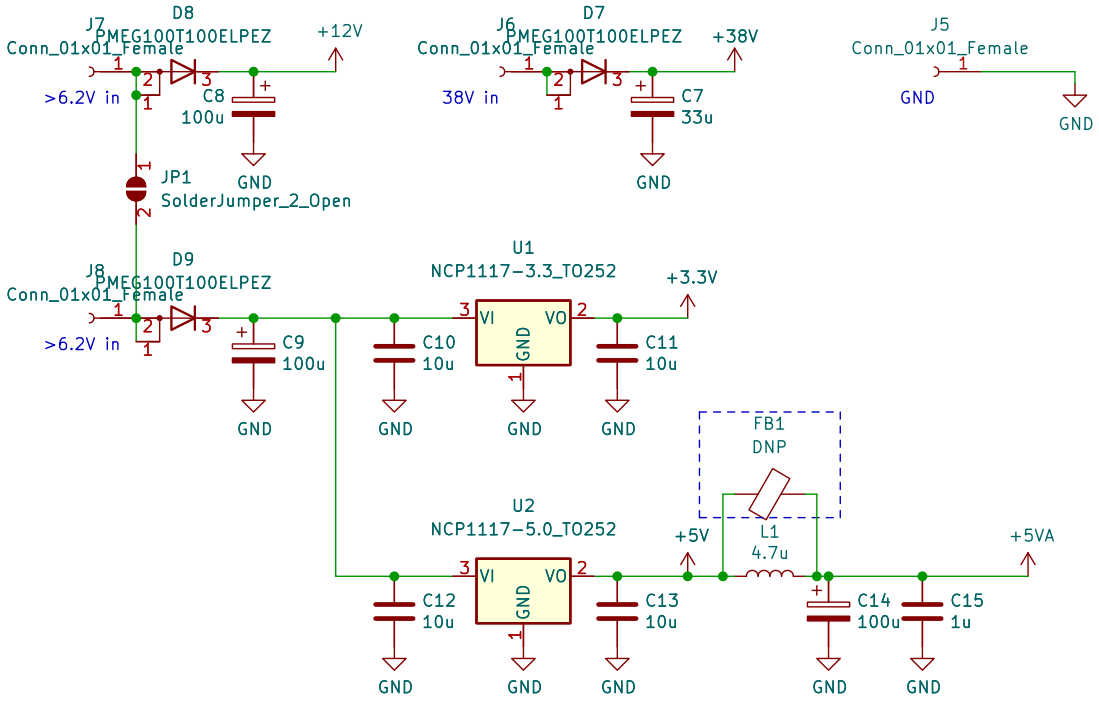


Figure 2.7: Power supply schematic

The power stage features three different supply voltage inputs in addition to a common ground connection. All inputs are protected against reverse polarity by nexperia PMEG100 general purpose Schottky barrier rectifier with a forward voltage drop of $750\,\mathrm{mV}$ [19]. The first input supplies the two onsemi NCP1117-3.3 and NCP1117-5 LDOs, providing stable $3.3\,\mathrm{V}$ and $5\,\mathrm{V}$ references respectively [20]. In order to compensate the voltage drop of the reverse polarity protection diode and to meet the specifications of the $5\,\mathrm{V}$ LDO, the input voltage at the connector should be greater than $6.2\,\mathrm{V}$. According to the recommended application, the LDOs are decoupled with $10\,\mu\mathrm{F}$ ceramic capacitors. As the $5\,\mathrm{V}$ LDO has to supply digital logic which can lead to increased noise due to high switching frequencies, the analog rail is filtered using an LC low-pass filter in addition to the usage of a ferrite bead connected to the VU pin of the FPGA module. The $3.3\,\mathrm{V}$ rail is for digital signals only. The other two voltage inputs supply the gate driver, which requires a gate drive supply voltage between $5.5\,\mathrm{V}$ and $16\,\mathrm{V}$, depending on the desired gate voltage and the resulting $R_{DS,ON}$ resistance, as well as the driving voltage for the piezo transducers, which is specified to be around $38\,\mathrm{V}$. The gate driver voltage input can be shorted to the LDO voltage input, if only two DC power supplies are desired and the thermal budget of the LDOs is not exceeded.

2.1.7 PCB design

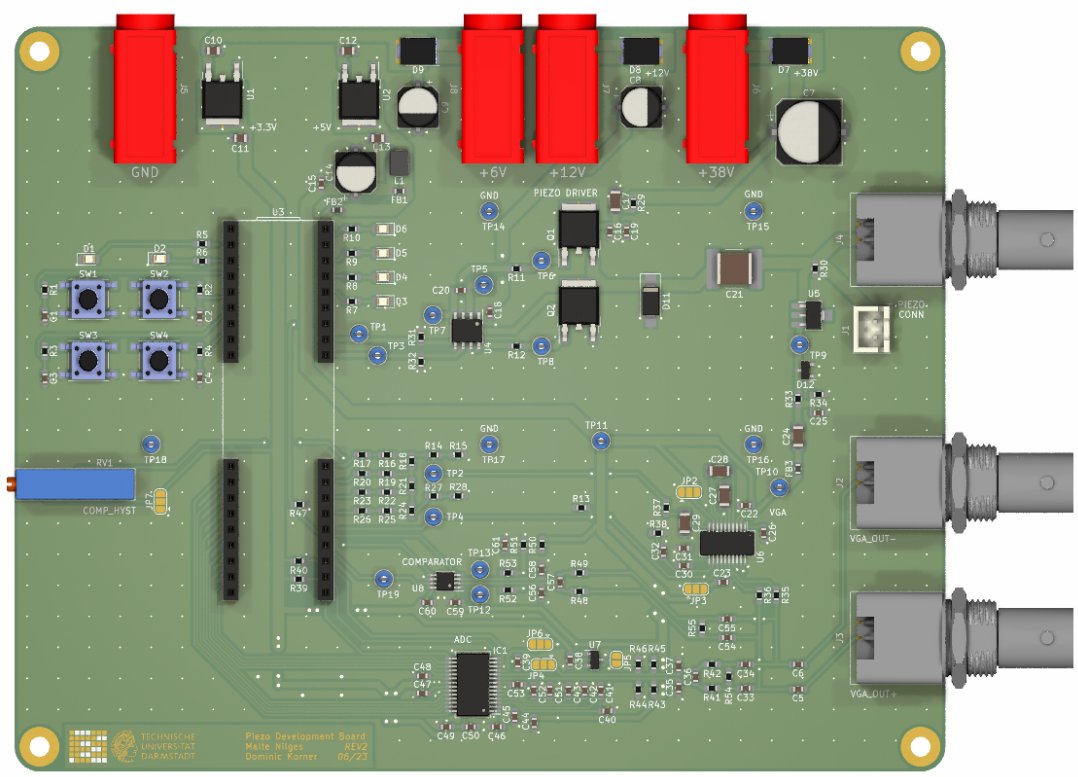


Figure 2.8: PCB Rendering

The PCB layout is based on a simple two layer board with dimensions of 150 mm x 120 mm, executed in a horizontal layout. As the signals on the board are on the lower MHz region and the component size and density do not require a high trace density, a two layer design approach was chosen, as it is the most efficient choice in terms of design time, production time and cost. The bottom layer acts as a ground plane with little breaks for routing in order to create a low impedance return path and to avoid large loop inductance which may lead to noise negatively effecting the AC performance of the board [21]. The top layer is used for component placement and signal routing with a densly stitched copper pour to the ground plane which avoids the risk of bending due to uneven copper distribution and may improve radiation of high speed digital signals.

The power section was placed on the upper section of the board, with the common ground connector on the left, and the other DC voltage connectors placed to the right alongside their respective decoupling capacitors and LDOs. The LDOs are stitched to small copper pours on the bottom layer to improve thermal dissipation. The 48-pin DIP-sized CMOD S7 has only 36 pins, hence the footprint has been replaced with two 18-pin DIP footprints in order to increase the routing area and to maximize the ground pour. The buttons, LEDs and the DAC are placed directly to the sides of the CMOD S7 module. The piezo driver stage, the input stage and the ADC and comparator stage are placed to the right of the CMOD S7 module. Where specified, routing and placement of decoupling capacitors has been executed in adherence to the guidelines in the data sheets, which is the case for the gate driver, the VGA and the ADC. The piezo connectors as well as the amplified differential signal outputs are placed on the right side of the board.

2.2 Software platform

The software has been completely rewritten, as the the previous platform only consisted of driving the piezo, without sampling and data transfer mechanisms to a host computer over the serial interface. The code described in the following sections is written in Verilog hardware description language (HDL) and synthesized with Xilinx Vivado 2022.2 in order to deploy the logic on the Xilinx XC7S25 FPGA.

2.2.1 Serial communication

The serial communication relies on two interfaces, the universal asynchronous receiver/transmitter (UART) interface for converting a bitstream to bytes for incoming data and vice versa for outgoing data within the modules uart_rx and uart_tx, as well as a worker module interface for centralized command decoding and transmission arbitration within the modules serial_rx_handler and serial_tx_handler. Common definitions for worker modules are contained in the file serial_defines.hv.

uart_rx and uart_tx

For the communication to the host computer, the FTDI FT2232HQ USB-UART bridge is being used. As the FPGA is supposed to receive and transmit data according to the UART protocol over two pins (RX and TX), the first step was to create an interface to send and receive bytes within a data frame. As there are existing solutions for FPGA, the respective code was taken from the reference implementation in [22].

The module for receiving data from host, called uart_rx has a clock signal (*i_Clock*) and the signal from the RX pin (*i_Rx_Serial*) as inputs and offers a valid signal (*o_Rx_DV*) and the received byte (*o_Rx_Byte*) as outputs. The module parameter *CLKS_PER_BIT* determines, how many clock cycles each transferred bit is applied before continuing with the next bit, thus defining the required baud rate for a data transfer. This oversampling technique is the working principle by which a reliable data transfer is made possible without a synchronizing clock between the transmitter and receiver [23].

The state machine consists of four states: *s_IDLE*, *s_RX_START_BIT*, *s_RX_DATA_BITS*, *s_RX_STOP_BIT* and *s_CLEANUP*. The incoming bits from the RX pin are double-registered before being processed in order to avoid metastability. Starting from the state *s_IDLE*, once a start bit ('0') has been detected, the state machine proceeds to the *s_RX_START_BIT* state, where the start bit is resampled at the middle of the *CLKS_PER_BIT* cycle by setting a counter. If the bit is still low, the next state is entered and the data bits are sampled likewise at the center of each respective cycle until the byte is filled. After a high stop bit is received, the valid signal is applied for one cycle until it is reset in the *s_CLEANUP* state. The state machine returns into its idle state.

The uart_tx module receives a byte and a valid signal as inputs and offers the serial signal for the TX pin as its output together with a active and done signal for signaling with the module sending the byte. The states of the module match these of the uart_rx module, but instead of sampling the data at the center of a cycle with the length defined in the *CLKS_PER_BIT* cycle, it outputs the start bit, the data bits beginning with the LSB and the finish bit, all for the duration of aforementioned parameter. The active signal is applied beginning with the reception of a valid byte for the duration of the transmission, the done signal is applied for one cycle at the end of the transmission.

serial_rx_handler

The serial_rx_handler module has a UART bitstream (*uart_rx_stream*) as input and passes the decoded input in the form of destination (*rx_dst*), command (*rx_cmd*), value (*rx_val*) together with a busy and finish signal (*rx_busy/rx_fin*) to the worker modules.

In order to decode the bitstream coming from the RX pin, it is first forwarded to the uart_rx module. The received bytes are then processed within a finite state machine (FSM), implementing the states *STATE_DST*, *STATE_CMD*, *STATE_VAL* and *STATE_CLR*. Every input must adhere to the pattern <DESTINATION>:<COMMAND> <VALUE>, which reflects the procedure of the state machine. Within the project, this FSM is the only one implementing the two-process state machine style with one process implemented as combinatorial and one process as synchronous logic. This served as a trial balloon, but seemed more complicated and error-prone compared to the single process FSM and was therefore not pursued further. The code in listing 2.1 shows an example for a decoding state based on *STATE_DST*. If a new byte is available, the *dst_buf* register is left-shifted 8 bits in order to fill bits 7 to 0 with the new data. If the new byte is a delimiter character, the state machine proceeds to the next state or to *STATE_CLR*. The used delimiters are ':' (colon) as a delimiter for the destination, ' ' (space) as a delimiter for the command and line-feed (usually enter) as a delimiter for the value. If a line-feed is received in another state, the state machine proceeds to *STATE_CLR* and resets the registers in order to receive a new input.

```
76  STATE_DST: begin
77     if (valid) begin
78        if (data == STRING_DELIM_DST) begin
79           nextstate = STATE_CMD;
80        end
81        else if (data == STRING_DELIM_FIN) begin
82           nextstate = STATE_CLR;
83        end
84        else begin
85           dst_buf_next = (dst_buf << 8);
86           dst_buf_next[7:0] = data;
87        end
88     end
89  end
```

Listing 2.1: serial_rx_handler decoding state machine

The separately stored strings for destination and command are compared to a list of strings and in case of a match coded to a corresponding number, see listing 2.2. The *val_buf* register contains the value as ASCII string, which is a binary coded decimal (BCD) for numbers with preceding bits, hence the binary value can be extracted by multiplying each decimal number with the respective power of ten. The *rx_fin* is applied if the state transitions from *STATE_VAL* to *STATE_CLR*.

```
126 assign rx_dst_wire = (dst_buf == STRING_DST_VGA ) ? DESTINATION_VGA :
...                                                      ...
131                                                      DESTINATION_NONE ;
```

Listing 2.2: String comparison and destination encoding

serial_tx_handler

The serial_tx_handler module acts as an arbiter for the transmission of UART data. For this reason, it receives a byte-sized input together with a valid signal for each module, that needs to initiate serial communication (e.g. *vga_data* and *vga_data_valid*). For signaling it offers a ready bit (*vga_data_ready*), indicating whether new data can be sent. The bitstream received from the instantiated *uart_tx* module is the output for the TX pin (*uart_tx_stream*).

The state machine is a priority encoder with a defined timeout, see listing 2.3. In *STATE_IDLE*, the state can be switched to that of a worker in case it is offering data indicated by its **_valid* signal. The output of the VGA module has the highest priority, followed by the comparator, the ADC and the COMM modules in descending order. The FSM returns to *STATE_IDLE* only, if a timeout counter reaches the values specified in *TIMEOUT_CYCLES_INACTIVE*. The counter increments unless the uart_tx module indicates that it is active, in which case the counter resets to 0. The data and valid signal passed to the uart_tx module is determined by a multiplexer controlled by the state of the FSM.

```
97  case (state)
98     STATE_IDLE: begin
99        if (vga_data_valid)
              state <= STATE_VGA;
100       else if (comp_data_valid)
              state <= STATE_COMP;
101       else if (adc_data_valid)
              state <= STATE_ADC;
102       else if (comm_data_valid)
              state <= STATE_COMM;
103       else
              state <= STATE_IDLE;
104    end
105    default: begin
106       if (counter == TIMEOUT_CYCLES_INACTIVE)
              begin
107           state <= STATE_IDLE;
108       end
109    end
110 endcase
```

Listing 2.3: serial_tx_handler priority encoder state machine

2.2.2 VGA

The vga_driver module contains the logic for enabling the VGA, controlling its gain and returning the status. Due to its low complexity, it serves as a comprehensive example for the communication and control scheme that is also applied to the other modules.

vga_driver

The inputs of the module are *power_signal*, *gain_inc_signal* and *gain_dec_signal*, which are applied for one clock cycle at the push of their respective button and has the enable signal (*vga_enable*) and the DAC code (*dac_value*) as outputs, together with signals for driving four LEDs indicating the gain. Like the other worker modules, it also interfaces with the serial interface as described in section 2.2.1. The parameters *DEFAULT_POWER* and *DEFAULT_GAIN* serve to adjust the default values of gain and power state of the VGA. Each of these parameters have secondary **_ASCII* parameters which should contain the default values in ASCII format.

Central to the module are the single-process FSMs. The input handler FSM reacts to inputs, either by button presses or by serial data, executes output logic and passes status messages to the output handler FSM, which initiates the serial communication.

Listing 2.4 shows the Verilog code for the input handling part of the process. Input handling is performed exclusively with *if...else* statements, meaning the dismissal of all but one input in the unlikely case of concurrent inputs in one clock cycle. This is mainly done in order to prevent multiple instances of the ASCII encoding logic. The if-branch of an applied *power_signal* toggles the *vga_enable* output register and stores the resulting value in converted ASCII format in the *ascii_val_power* register for future usage in status messages. In order to reduce code duplication, the local variable *recv_value* is introduced. As this reg type is always write-before-read, it is not inferred as a register, but seen as a wire instead.

The last if-case of the input handler is the serial listener. If a *rx_fin* is applied and the *rx_dst* input matches the destination code of the module defined in the serial_defines.hv file, the FSM executes the logic for the passed command and value. The code snippet shows the case for *COMMAND_POWER* and *COMMAND_STATUS*. Like

the button press, *COMMAND_POWER* changes the output signal and stores the corresponding ASCII code. The difference is that the output is not toggled, but set to the LSB of *rx_val*. Additionally, the change is returned over UART by storing the return message left-justified (by zero-padding) in the *write_buffer* register, setting the number of bytes to transmit with the *tx_count* register and applying the *tx_start* register for one clock cycle. The transmission handler of the FSM performs the transmission. The *COMMAND_STATUS* state returns all adjustable outputs of the module. The *write_buffer* register has to be sufficiently sized in order to fit the return message, so code changes in the return logic usually require checking the local parameter *SZ_BUF* which determines said size.

```
87  if (power_signal == 1) begin
88     recv_value  = ~vga_enable;
89     vga_enable  <= recv_value;
90     ascii_val_power <= bin2ascii10(recv_value);
91  end
... ...
102 else if (rx_fin && rx_dst == DESTINATION_VGA) begin
103    case (rx_cmd)
104      // VGA:POWER [0/1]: enables/disables vga.
105      COMMAND_POWER: begin
106        recv_value  = rx_val[0];
107        recv_value_ascii = bin2ascii10(recv_value);
108        vga_enable   <= recv_value;
109        ascii_val_power <= recv_value_ascii;
110
111        write_buffer <= {ASCII_TXT_POWER, recv_value_ascii, ASCII_LF,
112                         {SZ_BUF-80{1'b0}}};
113        tx_count    <= 10;
114        tx_start    <= 1;
115      end
116      // VGA:STATUS [<any number>]: return status message.
117       COMMAND_STATUS: begin
118        write_buffer <= {ASCII_TXT_POWER , ascii_val_power , ASCII_LF,
119                        ASCII_TXT_GAIN  , ascii_val_gain , ASCII_LF,
120                        {SZ_BUF-144{1'b0}}};
121        tx_start <= 1;
122        tx_count <= 18;
123       end
... ...    ...
150    endcase
151 end
```

Listing 2.4: RX handler of vga_driver FSM

The transmission handler of the process is a FSM with a default *TX_IDLE* state and a *TX_SEND* state. The *TX_IDLE* state checks for a an applied *tx_start* signal and proceeds to the *TX_SEND* state, which is shown in listing 2.5. If there are bytes available, the *tx_data* register is assigned to the eight MSBs of the *write_buffer* and *tx_valid* is set to high. These registers are passed to the serial interface and only changed if the interface returns *tx_ready*. In this case, the *write_buffer* is left-shifted by eight bits and the *tx_count* register decremented by one. If *tx_count* reaches zero, the state is changed back to *TX_IDLE* and the *tx_valid* is unset.

```
160 TX_SEND: begin
161   if (tx_count > 0) begin
```

```
162       tx_valid   <= #1 1'b1;
163       tx_data    <= #1 write_buffer[SZ_BUF-1:SZ_BUF-8];
164       if (tx_ready) begin
165         write_buffer <= #1 write_buffer << 8;
166         tx_count   <= #1 tx_count - 1;
167       end
168     end
169     else begin
170       tx_valid <= #1 1'b0;
171       state_tx <= #1 TX_IDLE;
172     end
173   end
```

Listing 2.5: TX_SEND state of vga_driver FSM

2.2.3 Comparator

The comparator outputs a logic '1' to the FPGA in case of a positive differential voltage and a logic '0' in case of a negative differential voltage. This data is used to determine the absolute frequency of the piezo signal or its change. In order to obtain this data, the FPGA has to process the input at high frequencies and pass it at a lower frequency over the UART interface. Therefore the comparator_driver module implements functionality for generating a secondary clock, for processing the sampled data, for storing the processed data in the block random access memory (BRAM) of the FPGA and stream it from said BRAM to the serial interface for transmission to a host computer. The code is reused to a large extent in the ADC module as the procedure is very similar except for the actual processing of the data.

comparator_driver

The module comparator_driver has, besides the serial interface, the signals *comp_in* and *trigger_in* as inputs and the signal *comp_en* as output. The parameters of the module are *DEFAULT_POWER*, *DEFAULT_TRIG*, *DEFAULT_MAXSMP* and *DEFAULT_WIDTH*. For input and transmission handling, the module uses the same FSM pattern as described in section 2.2.2, with the addition of data streaming for transmission. This data streaming is initiated, if the output first in first out (FIFO) indicates available data using the *outfifo_vld* register. If data is available, the transmission FSM switches from *TX_IDLE* to *TX_HEAD1*. The states *TX_HEAD1* to *TX_HEAD5* and *TX_TAIL1* to *TX_TAIL3* wrap the streamed data inside a frame of the following format:

- Byte 1: 8'h80
- Byte 2: 8'h7F
- Byte 3: 8'h43 (ASCII character 'C') indicating the streaming source
- Byte 4: *freq_state* register indicating the sampling frequency
- Byte 5: *width* register indicating the width in bits of each sample
- Byte 6 to N-3: *sample_stream* register (FIFO output)
- Byte N-2: 8'h7F
- Byte N-1: 8'h80

- Byte N: 8'h0A (ASCII line-feed character).

For sampling the incoming data at high frequencies, the module generates a secondary clock using Xilinx MMCME2_BASE primitive [24]. An instantiation of this block design controls the mixed-mode clock manager (MMCM) in one of the three clock management tiles (CMTs) available on the FPGA [25]. The MMCM serves as a frequency synthesizer, jitter filter and clock deskew. For the purpose of generating a seperate clock, only the frequency synthesis is relevant for this module. Listing 2.6 shows the respective Verilog code. The parameters of the MMCME2_BASE module define the input clock period, the multiplier for the feedback clock, which is applied for all output clocks and the divider for the first output clock *CLKOUT0*. The values are chosen to match the input clock of $12\,\mathrm{MHz}$ and to achieve an output clock of $50\,\mathrm{MHz}$. The resulting clock is buffered and made available in form of the *clk_comp* register.

```
131 MMCME2_BASE #( // Xilinx HDL Language Template, version 2021.2
132    .BANDWIDTH("OPTIMIZED"), // OPTIMIZED, HIGH, LOW
133    .CLKFBOUT_MULT_F (62.5), // Multiply value for all CLKOUT (2.000-64.000).
134    .CLKFBOUT_PHASE (0.0),  // Phase offset in degrees of CLKFB,
          (-360.000-360.000).
135    .CLKIN1_PERIOD (83.333), // Input clock period in ns to ps resolution (i.e.
          33.333 is 30 MHz).
136    .CLKOUT0_DIVIDE_F (15.000),
...    ...
143 ) mmcme2_comp_inst (
144    .CLKOUT0       (clk_comp_unbuf),  // 1-bit output: CLKOUT0
...    ...
151    .CLKFBOUT      (clk_feedback_unbuf), // 1-bit output: Feedback clock
152    .CLKFBOUTB     (),                // 1-bit output: Inverted CLKFBOUT
153    .LOCKED        (),                // 1-bit output: LOCK
154    .CLKIN1        (clkin_buf),       // 1-bit input: Input clock
155    .PWRDWN        (1'b0),            // 1-bit input: Power-down
156    .RST           (1'b0),            // 1-bit input: Reset
157    .CLKFBIN       (clk_feedback_buf) // 1-bit input: Feedback clock
158 );
```

Listing 2.6: MMCME2_BASE instance of comparator_driver

To note is that many control signals are synchronous to the $12\,\mathrm{MHz}$ system clock, therefore they are synchronized to the faster sampling clock using the Xilinx XPM_CDC_* macros [24] before being used in the *clk_comp* clock domain. The XPM_CDC_* modules are intended for clock domain crossing (CDC) purposes and provide different strategies to synchronize signals depending on the type, the bus size and the duration of the signal. Signals that are synchronized to the comparator clock are denoted with **_comp* within this module.

The sampling FSM runs with *clk_comp* and consists of three states: *SAMPLE_WAIT*, *SAMPLE_RUN* and *SAMPLE_RST* as seen in listing 2.8. Its purpose is to count the number of clock cycles that the *comp_in* input remains stable after being triggered. The state *SAMPLE_WAIT* waits until a *force_single* signal is applied or an external trigger (*trigger_in*) is detected while triggering is enabled, as determined by the *trigger* register. The register *cycle_counter* is set to 1 and the state changes to *SAMPLE_RUN*. After triggering, the *trigger* register is unset in the input handling process of the module as seen in listing 2.7.

```
525 if (!bram_empty) begin // detect triggering on !bram_empty signal (same clock)
526   trigger   <= 0;
527   config_rdy   <= 1'b1;
528   ascii_val_trig  <= 8'h30;
```

```
529 end
```

Listing 2.7: Trigger reset in RX handler of comparator_driver

The *SAMPLE_RUN* state increments *cycle_counter* by 1 in each clock cycle until the input is changed compared to the previous clock cycle. If this is the case, the input is passed to the input FIFO in order to be stored in the BRAM afterwards. The sampling continues until the BRAM is full or the *sample_count* register reaches 0 after being initialized to *max_samples* and decremented for every FIFO entry. The state *SAMPLE_RST* waits until the transmission logic applied a reset signal after the completed transmission in order to return to *STATE_WAIT*.

```
525 case (state_sample)
526   // Wait state: trigger on force signal or when threshold is exceeded
527   SAMPLE_WAIT: begin
528     if (force_single_comp ||
529       trigger_in_comp && trigger_comp) begin
530         cycle_counter <= 1;
531         state_sample <= SAMPLE_RUN;
532     end
533   end
534   // Run state: write samples to infifo and keep triggered until sample_count is
        reached or bram is full
535   SAMPLE_RUN: begin
536     cycle_counter  <= cycle_counter + 1;
537     if (sim_switch_comp && (sim_comp_in != previous_comp) ||
538       !sim_switch_comp && (comp_in_synced != previous_comp)) begin
539         sample_in  <= cycle_counter;
540         sample_in_vld <= 1'b1;
541         cycle_counter  <= 1;
542         sample_count <= sample_count - 1;
543     end
544     if (sample_count == 0 || bram_full) begin
545       state_sample <= SAMPLE_RST;
546     end
547   end
...    ...
555 endcase
```

Listing 2.8: Sampling FSM of comparator_driver

An additional feature is the test of the sampling and FIFO logic using a simulated input. The sampling logic switches between simulated and real input using the *sim_switch* register, which is set and unset by the respective command sent over UART. The input simulation is performed in a separate process, switching the simulated input from '0' to '1' and vice versa after a timeout, which is incremented after each switch.

The cycle counts determined in the sampling FSM are stored in the BRAM before being transmitted. The XPM_FIFO_ASYNC macro from Xilinx is used to instantiate an asynchronous FIFO using the BRAM of the FPGA [24]. It supports the usage of different clock signals for data input and output, thus integrating CDC functionality. The depth and data read and write width of the FIFO are defined as parameters of the macro, with the write depth being limited to values that are a power of two and the data width limited to aspect ratios of 1:1, 1:2, 1:4 and 1:8 for either read or write width. Depending on the write width of the FIFO, the available BRAM resources can be more or less efficiently used. This is especially relevant because these resources are shared with the FIFO of the adc_driver module.

In order to be more flexible in these terms, a design approach was chosen, in which the results are first aggregated to match the width of the BRAM FIFO and then passed to said BRAM FIFO from where they are retrieved and split up into single bytes for transmission. This process is illustrated in fig. 2.9. The aggregation and splitting logic is implemented in the modules fifo_aggregate and fifo_x2byte respectively and described in section 2.2.6.

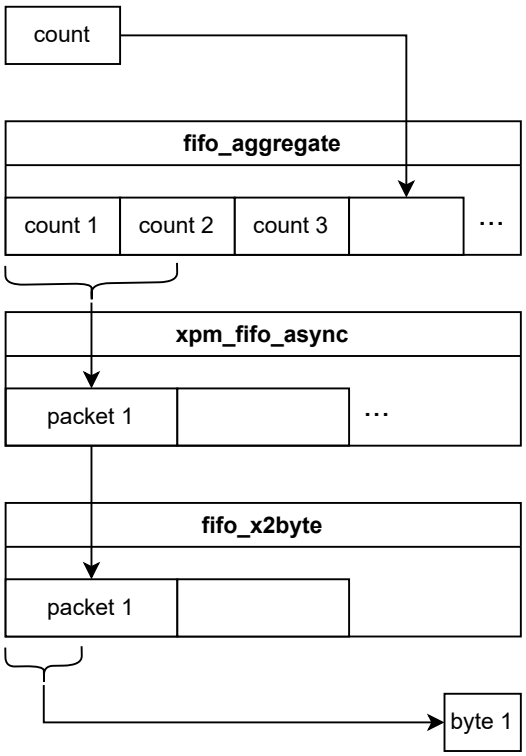


Figure 2.9: FIFO chain

2.2.4 ADC

The ADC converts the received signal into a 10-bit digital output coded as two's complement. The sampling and conversion rate is determined by the clock provided to the ADC, the output latency is 5.5 clock cycles. As the functionality needed for controlling, sampling and transmission is very similar to that of the comparator_driver shown in the previous section, the following description of the module adc_driver presents only the significant differences between the modules.

adc_driver

The module adc_driver has the 10-bit wide signed input (*adc_in*) for the parallel output of the ADC and outputs *adc_en* and *clk_adc* for enabling and clocking the ADC as well as a *trigger_out* signal, which is used by the comparator_driver module. The parameters of the module are *DEFAULT_POWER*, *DEFAULT_TRIG*, *DEFAULT_THOLD*, *DEFAULT_MAXSMP*, *DEFAULT_WIDTH* and *DEFAULT_FREQ* along with its respective **_ASCII* representations.

Unlike the comparator_driver, this module uses a variable clock as output to the FPGA and for sampling. This functionality allows the user to change the clock frequency in order to either increase the amount of stored data before the BRAM FIFO is full or to increase the sampling frequency. In order to achieve this, a module named adc_clkgen is instantiated which receives a signal *SSTEP* in order to initiate a frequency change, a 5 bit wide *STATE* signal representing the desired frequency, an optional *RST* signal and the base clock *CLKIN*, from which a clock is synthesized and returned as *CLK_ADC* together with a *LOCKED_OUT* signal indicating whether the clock is stable.

```
108 reg [5:0] freq_state  = DEFAULT_FREQSTATE;
109
110 adc_clkgen u_adc_clkgen (
111   .SSTEP             ( freq_update ),
112   .STATE             ( freq_state ),
113   .RST               ( 1'b0      ),
114   .CLKIN             ( clk       ),
115
116   .LOCKED_OUT        ( freq_locked ),
117   .CLK_ADC           ( clk_adc   )
118 );
```

Listing 2.9: adc_clkgen instantiation in adc_driver

The *STEP* input is wired to the *freq_state* register of the adc_driver module. Frequencies are selectable in $250\,\mathrm{kHz}$ steps, the lowest being $250\,\mathrm{kHz}$ for the state 0 and the highest being $10\,\mathrm{MHz}$ for state 39. The clock

synthesis is performed by an instantiation of Xilinx MMCME2_ADV module, which controls one MMCM of the FPGA as done by the MMCME2_BASE module shown in section 2.2.3, but with the option to access its reconfiguration registers by the means of additional input and output signals in order to reconfigure the MMCM during runtime. For a correct interfacing and register access, Xilinx provides the module template mmcme2_drp along with a header file, which were adjusted to define the frequencies for reconfiguration.

The input handler FSM of adc_driver accepts a command setting the frequency with the value passed as the desired frequency in kilohertz, e.g. 250. In order to avoid code duplication or division logic, an expanding for-loop has been employed in the respective *COMMAND_FREQ* state to compare the number against a possible *freq_state* as seen in listing 2.10

```
276 COMMAND_FREQ: begin
277   recv_value  = rx_val;
278   recv_value_ascii = 40'h494E56; // "INV"
279
280   for (ii = 1; ii <= 40; ii = ii + 1) begin
281     if (recv_value == (ii * 250)) begin
282       recv_value_ascii = bin2ascii10000(recv_value);
283
284       freq_state    <= ii - 1;
285       freq_update   <= 1'b1;
286       ascii_val_freq  <= recv_value_ascii;
287     end
288   end
...    ...
294 end
```

Listing 2.10: COMMAND_FREQ state of adc_driver RX FSM

The sampling FSM handles the incoming data in every cycle of set frequency. The FSM consists of two states: *SAMPLE_RUN* and *SAMPLE_RST*. In the *SAMPLE_RUN* state, the incoming data is sampled if a triggering is forced manually using the *force_single* register, the (simulated) input exceeds the value of the *trigger_threshold* register while triggering is enabled (*trigger* register) or if the sampling was already triggered and is not yet finished as indicated by the *triggered* register. The sample is shifted such, that the in register *width* specified number of bits are written to the *sample_in* register which feeds the input FIFO. While the input FIFO takes inputs of the width *MAX_SAMPLE_WIDTH*, only an adjustable number defined by said *width* register are stored.

If the BRAM FIFO is full or *sample_count* reached 0, the sampling is stopped and the FSM continues to the *SAMPLE_RST* state, which waits until the transmission logic applies a reset signal. As that reset signal is set for only one cycle in the faster system clock domain, it is stretched before being synchronized to the *clk_adc* using the module pulse_stretcher so that the pulse is not missed. The resulting signal is named *reset_w_extd_adc*.

```
477 // Run state: trigger on force signal or when threshold is exceeded
478 //         write samples to infifo and keep triggered until sample_count is
       reached or bram is full
479 SAMPLE_RUN: begin
480   if (force_single_adc ||
481     (trigger_adc && adc_in > trigger_threshold_adc && !sim_switch_adc) ||
482     (trigger_adc && sim_adc_in > trigger_threshold_adc && sim_switch_adc) ||
483     triggered) begin
484       if (sample_count > 0 && !bram_full) begin
```

```
485           if (sim_switch_adc)
486             sample_in  <= sim_adc_in >> (MAX_SAMPLE_WIDTH - width_adc);
487           else
488             sample_in  <= adc_in >> (MAX_SAMPLE_WIDTH - width_adc);
489           sample_in_vld <= 1'b1;
490           triggered   <= 1'b1;
491           sample_count <= sample_count - 1;
492         end
493         else begin
494           sample_in_vld <= 1'b0;
495           triggered   <= 1'b0;
496           sample_count <= 1'b0;
497           state_sample <= SAMPLE_RST;
498         end
499     end
500     else begin
501         sample_in_vld  <= 1'b0;
502     end
503 end
```

Listing 2.11: SAMPLE_RUN state of adc_driver sampling FSM

The adc_driver module also employs logic for outputting a trigger signal. This output *trigger_out* is always '1' if *adc_in* exceeds the threshold and is synchronous to *clk_adc*.

```
531 always @(posedge clk_adc) begin
532   if (adc_in >= trigger_threshold_adc && !sim_switch_adc ||
533     sim_adc_in >= trigger_threshold_adc && sim_switch_adc)
534       trigger_out_adc <= 1;
535   else
536       trigger_out_adc <= 0;
537 end
```

Listing 2.12: trigger_out process of adc_driver

The simulation logic emulates a input which is incremented by 1 each *clk_adc* clock cycle.

2.2.5 Piezo driver

The logic for driving the piezo is defined within the module piezo_driver which is instantiated and controlled by its instantiating module comm_protocol. The comm_protocol receives commands from the host or button inputs and directs the piezo_driver to switch the low and high input of the gate driver and consequently the half-bridge driving the piezo. The transmission of the digital code is performed using on-off keying (OOK) modulation scheme, which represents digital data by a defined duration of presence or absence of a carrier wave [26].

comm_protocol

The module comm_protocol has a single input *button_start* and outputs a signal *comm_fin* and the signals for switching the gate driver inputs (*piezodriver_hi* and *piezodriver_lo*). Additionally it communicates with

the serial_interface module to receive and transmit data. The parameters of the module mainly set the default pulsing durations for charging (*DEFAULT_CHP*), a '0' bit (*DEFAULT_0L*), a '1' bit (*DEFAULT_1L*) and the break durations in microseconds before starting a transmission (*DEFAULT_STL*) and in between the pulses (*DEFAULT_BRL*). The parameter *DEFAULT_DATA* sets the 4 default bits to be transmitted and *DEFAULT_CLKDIV* sets the default clock divider to create the frequency with which the piezo is being switched. As for the other modules, the default values can be changed using the UART interface.

As mentioned before, the module instantiates piezo_driver in order to determine the pulse frequency and the number of pulses the piezo is supposed to generate. The *piezo_start_cfg* wire combines the signals *piezo_clkdiv*, *piezo_numpulses* and *piezo_start* and passes it to the module together with the *piezo_start_cfg_rdy* signal. The module returns whether it received the config and excecuted the pulsing operation to *piezo_start_cfg_rcv* and *piezo_fin* respectively. The implementation of the module is described further below. The delay module takes *delay_us* and *delay_start* as inputs and returns '1' to *delay_fin* if the internal counter reaches the value specified in *delay_us*. This counter is only incremented every 12 system clock cycles, which equals one microsecond.

```
78  reg [11:0] piezo_numpulses = 0;
79  reg [8:0] piezo_clkdiv  = DEFAULT_CLKDIV;
80  wire [21:0] piezo_start_cfg = {piezo_clkdiv,
        piezo_numpulses, piezo_start};
81
82  //wire piezo_start_100, piezo_fin_100;
83  wire piezo_start_cfg_rcv, piezo_fin, delay_fin;
84
85  piezo_driver m_piezo_driver (
86    .clk           (clk),
87    .start_cfg     (piezo_start_cfg),
88    .start_cfg_rdy (piezo_start_cfg_rdy),
89    .start_cfg_rcv (piezo_start_cfg_rcv),
90    .fin           (piezo_fin),
91    .piezodriver_lo (piezodriver_lo),
92    .piezodriver_hi (piezodriver_hi)
93  );
94
95  delay m_delay (
96    .clk_12        (clk),
97    .delay_us      (delay_us),
98    .start         (delay_start),
99    .fin           (delay_fin)
100 );
```

Listing 2.13: piezo_driver and delay instantiation in comm_protocol

The main part of the comm_protocol is the communication FSM, which defines the communication pattern. The FSM starts with state 0, sensing for *comm_start* which is either enabled by button or by its respective command. if the register is high, *comm_fin* is unset and the FSM continues to the next state. State 1 sets *piezo_numpulses* to the *Charge_Pulses* register and sets *piezo_start* to high. State 2 unsets these signals if peizo_driver received the signals. States 3 and 4 set and unset *delay_start*. This pattern continues with the transmission of the defined pattern of '0' and '1' bits and the defined break in between until all 4 bits have been transmitted.

An additional testing FSM has been implemented which simply repeats the tranmsission of the *Charge_Pulses* without breaks in between, resulting in a constant transmission. The test state is either initiated with the respective serial command or with *button_start* if the module was synthesized with a *COMM_TEST* flag defined within the module.

piezo_driver

The piezo_driver module synthesizes a $250\,\mathrm{MHz}$ clock with the Xilinx MMCME2_BASE module, which is divided down to a desired frequency using the value defined in *piezo_clkdiv* of comm_protocol. To achieve this, the received *start_cfg* is synchronized to the *clk_250* clock. The clock and state control FSM senses for

the *start* register in state *IDLE* and continues to state *BUSY* if it is set. State *BUSY* increments *clk_counter* until *counter_compare_value* is reached, which is set to the value of *piezo_clkdiv*. If it is reached, the FSM is in the state *BUSYPULSE* for one cycle, where the *pulse_counter* register is incremented by one and which either continues with state *IDLE* if the *pulse_counter* reached *numpulses* or otherwise returns to state *BUSY*.

The MOSFET output FSM senses in which state the clock and state control FSM is and cycles though its own states *state_out* accordingly. In *state_out IDLE*, both outputs *out0* and *out1* are set to '0'. If the other *state* changes to *BUSY*, *state_out* continues to *OUT1_WAIT*. This wait state sets both output signals to '0' and is responsible for providing the delay in order to avoid shoot-throughs as described in section 2.1.1. If *dly1* reached 0 after decrementing in each clock cycle, the FSM proceeds to *state_out OUT1*, in which *out1* is set to '1', as seen in listing 2.14. This changes only after a *BUSYPULSE* as the *state_out OUT0_WAIT* is entered thereafter and both output signals are set to '0'. The *state_out OUT0* acts in the same manner as *OUT1* and is therefore not further described. Independant of the current *state_out*, the FSM switches to *state_out IDLE* if *state* is *IDLE*.

```
110 OUT1_WAIT : begin
111    dly0 <= DLY;
112    out0 <= 0;
113
114    dly1 <= dly1 - 1;
115    out1 <= 0;
116    if (dly1 == 0)
117       state_out <= OUT1;
118 end
119 OUT1 : begin
120    out1 <= 1;
121    if (state == BUSYPULSE)
122       state_out <= OUT0_WAIT;
123 end
```

Listing 2.14: Snippet of MOSFET output FSM of piezo_driver

The outputs *out0* and *out1* are assigned to *piezodriver_lo* and *piezodriver_hi* respectively.

2.2.6 Helpers

Multiple modules mentioned in the previous sections instantiated the modules fifo_aggregate, fifo_x2byte and/or mmcme2_drp. As they are more complex, the former two are described within this section for a better understanding. Please refer to the in-file documentation for changing mmcme2_drp.

fifo_aggregate

The module fifo_aggregate has the purpose to store an input of adjustable width in order to aggregate it and return it as an output of higher width. For that purpose it has *rst*, *input_width* and *wr_en* as inputs and outputs *data_out* and *full*. The module has the parameters *MAX_INPUT_WIDTH* and *OUTPUT_WIDTH_BYTES* which define the maximum input width and the actual output width respectively.

As the output width may not be a perfect multiple of the input, the modules shift register width is the output width plus the maximum input width. The shift register is shifted by a variable amount of bits each time a *wr_en* is applied.

If the current written bit count *counter* plus *input_width* does not exceed *OUTPUT_WIDTH*, *current_shift_amount* is set to *input_width* + *overflow*. This *overflow* register differs from 0 only if *OUTPUT_WIDTH* is exceeded during a write to the FIFO. In the case that *counter* plus *input_width* reaches or exceeds *OUTPUT_WIDTH*, the *current_shift_amount* is set to the number of remaining bits until it is full and the number of remaining bits is stored in *current_overflow* in order to be shifted in the next clock cycle. Additionally *full* is set to '1' indicating a valid output.

The shift register right shifts according to *current_shift_amount* and inserts the arriving bits according to *input_width* and *current_overflow* to the left of the existing data.

fifo_x2byte

The module fifo_x2byte has the purpose to split data input with the width of multiple bytes to a single byte output. It receives *rst*, *rd_en*, *wr_en* and *data_in* as inputs and returns *data_out* and the status registers *full* and *empty*. The width of the input is defined by the module parameter *INPUT_WIDTH_BYTES*, the parameter *FIFO_DEPTH_INPUT* defines the number of stored inputs.

The module employs a common FIFO pattern with seperate read and write pointers. The memory is an 8-bit vector array with a depth of *MAX_NUM_BYTES* (= *INPUT_WIDTH_BYTES * FIFO_DEPTH_INPUT*). If an input is stored, *wr_ptr* is either increased by *INPUT_WIDTH_BYTES* or its MSB is inverted and the other bits set to '0' if the *wr_ptr* would reach *MAX_NUM_BYTES* otherwise. This MSB enables checking, whether the FIFO is full (if the write and read pointers have a different MSB) or empty (otherwise). The *full* flag is checked before storing an input.

If *rd_en* is set to '1' and the FIFO is not empty, the read pointer is either incremented by 1, or its MSB is inverted and the other bits are set to '0', if it would reach *MAX_NUM_BYTES*, analog to the behaviour of the *wr_ptr*.

The output *data_out* is assigned to *memory[rd_addr]*, with *rd_addr* being *rd_ptr* without its MSB, making the FIFO first-word-fall-through (FWFT), as the output is available without latency.

3 Results & Discussion

This chapter evaluates the implementation in consideration of the objectives laid out in section 1.1. This includes mainly the actual ultrasonic communication but also extends to the parts of software and hardware enabling the functionality. For the transmission tests, a SONOSCAN RMC 2.25 piezoelectric transducer (A) with a resonance frequency of around $2.09\,\mathrm{MHz}$ has been used, which is tightly pressed against a steel beam in an angle of 60° using a mount, magnets and ultrasound gel. As one piezo element failed during a thermal stress test, the reception tests used two not further specified piezoelectric transducers (B) with a resonance frequency of around $121\,\mathrm{kHz}$. These are put in a metal enclosure in a 90° angle. All test articles can be seen in fig. 3.1.

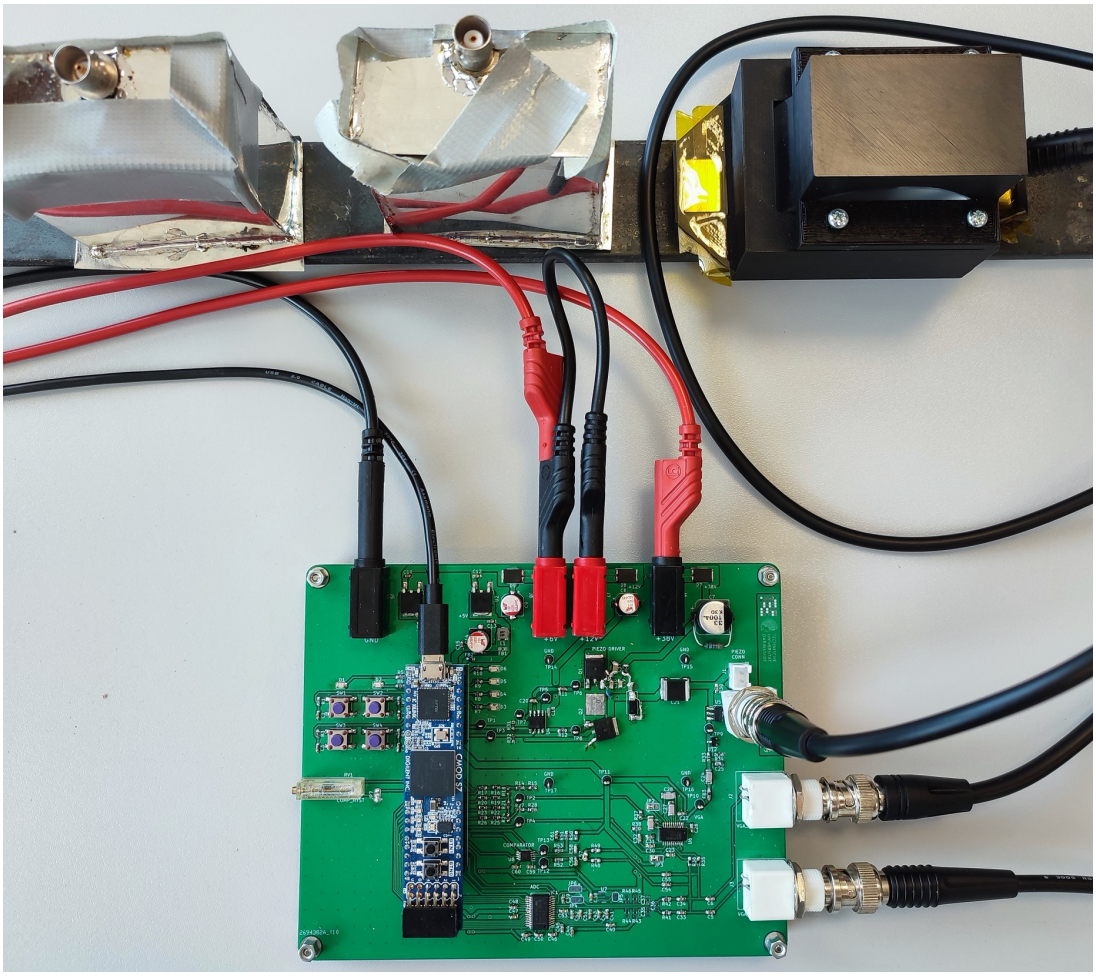


Figure 3.1: All test articles: PCB, two unspecified piezos (A) in a metal enclosure, single SONOSCAN piezo (B)

3.1 Ultrasonic transmission

The ultrasonic transmission is in terms of hardware enabled by the switched voltage-controlled voltage source (VCVS) consisting of the MOSFET gate driver and the MOSFET half-bridge and in terms of software by the logic defined in the Verilog module comm_protocol with its instance of piezo_driver. To decrease ringing for the targeted transducer (A), two snubber capacitors with $C_{snub} = 15\,\mathrm{pF}$ have been soldered between drain and source of both half-bridge MOSFETs according to the methodology shown in [27] and as verified in tests. This dampens the otherwise present ringing overshoots with frequencies of around $150\,\mathrm{MHz}$.

The default parameters have been set such, that a the bit pattern '0101' should be transmitted with a clock divider value of 200. This should result in a $625\,\mathrm{kHz}$ switching frequency for the piezo element.

Figure 3.2 shows the signals applied to the gate driver and the unterminated signal at the BNC connector output to the piezo transducer. The frequency matches the targeted $625\,\mathrm{kHz}$. The transmission procedure is performed as desired with first transmitting the charging pulses, then waiting the defined start timeout and finally sending the four bits consecutively with the defined break timeouts in between. The bit pattern matches the pattern defined in the module. The timeouts are defined in microseconds and are by default $500\,\mu\mathrm{s}$ after the charging pulse and $100\,\mu\mathrm{s}$ after each bit. The pulses are for testing a count of 100 for charging, 100 for a '0' and 200 for a '1'. The measurements show, that the the count is correct as $625\,\mathrm{MHz} \cdot 320\,\mu\mathrm{s} = 200$ cycles. As can be seen in fig. 3.3, the signal is not substantially delayed and the pulse width is uniformly half the clocking cycle.

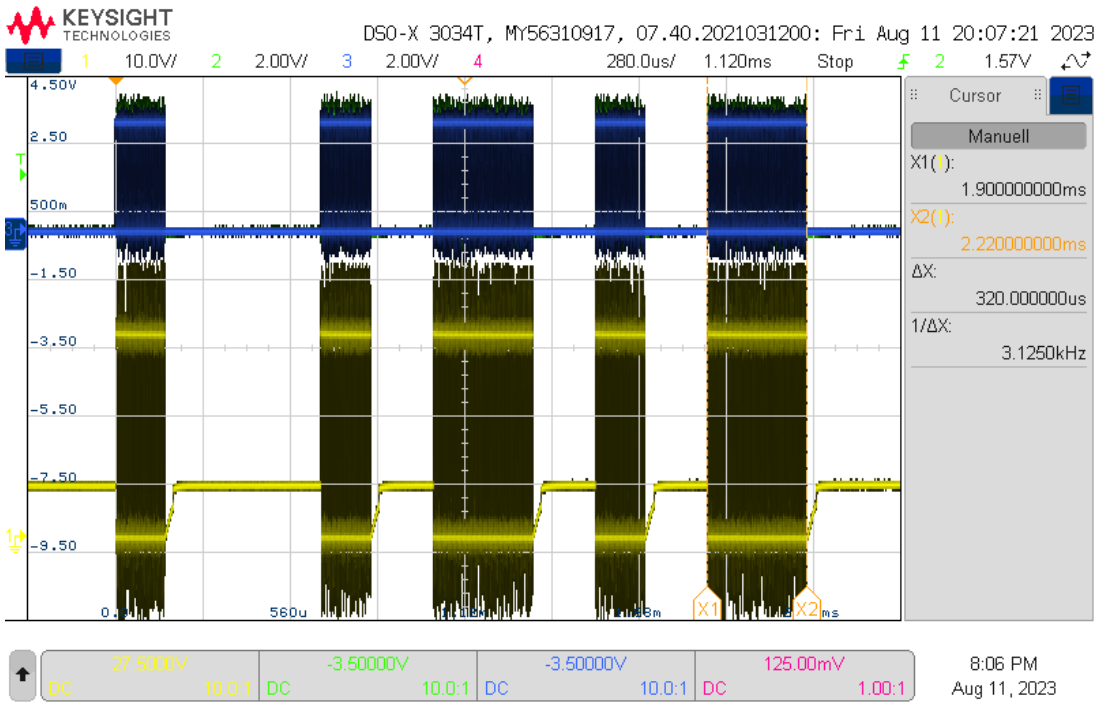


Figure 3.2: Packet waveform of unloaded transmission; channel 1: output of the MOSFET half-bridge, channels 2 & 3: low-side and high-side signals from the FPGA

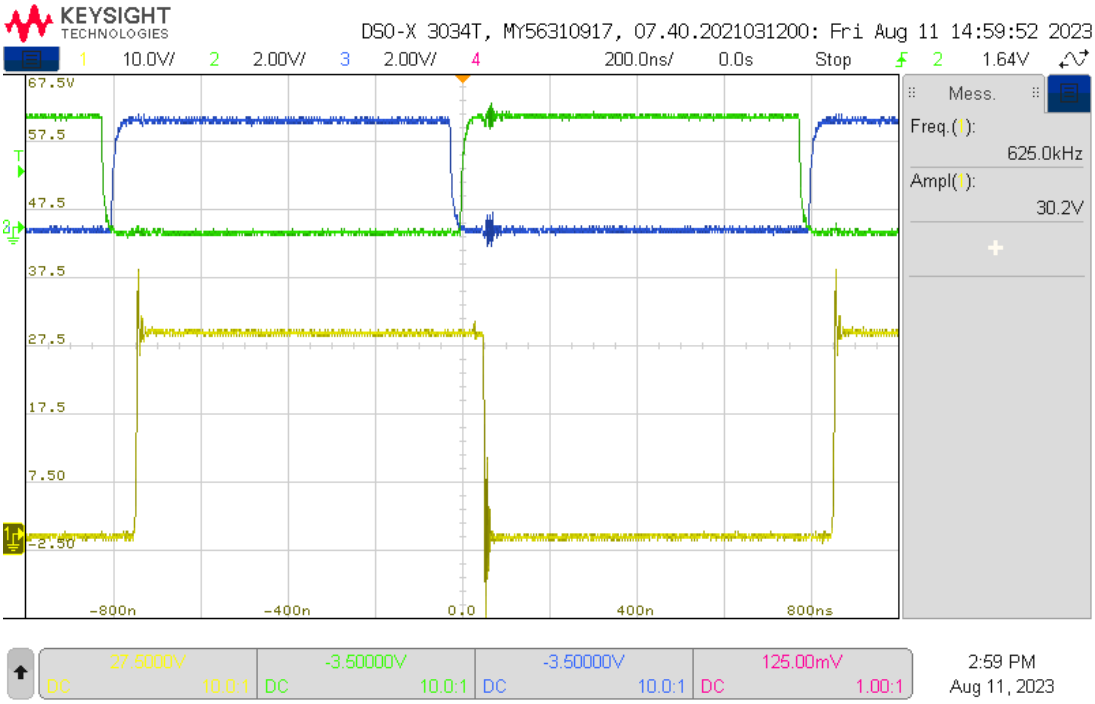


Figure 3.3: Detailed waveform of unloaded transmission; channel 1: output of the MOSFET half-bridge, channels 2 & 3: low-side and high-side signals from the FPGA

For the next test, the piezo element (A) has been connected to the BNC connector. The clock divider has been set to 60 for a pulsing frequency of around $2\,\mathrm{MHz}$. Reception could be verified using its counterpart before its destruction, although images can't be provided.

Instead, piezo transucers (B) were used at a transmission frequency of $313.3\,\mathrm{kHz}$, as the dominant resonance frequency of $121\,\mathrm{kHz}$ can't be achieved with the boundaries of the clock divider. Figure 3.4 shows the received signal at the secondary piezo element connected to the oscilloscope. Its Fourier transform reveals that the power is highest at around $313\,\mathrm{kHz}$ confirming a successful transmission.

As can be seen, the requirements for data transmission are met. Commands passed to the comm_protocol module are interpreted correctly and the frequency is changed as desired. The PCB is able to drive a piezo transducer with frequencies over $2\,\mathrm{MHz}$ with a sustained average current of $400\,\mathrm{mA}$ as verified in a stress test. The tests were able to show the communication pattern which was defined by the original version of the code. Due to the addition of an adjustable clock divider, transmission using frequency modulation is also feasible with a low effort for code rewrite.

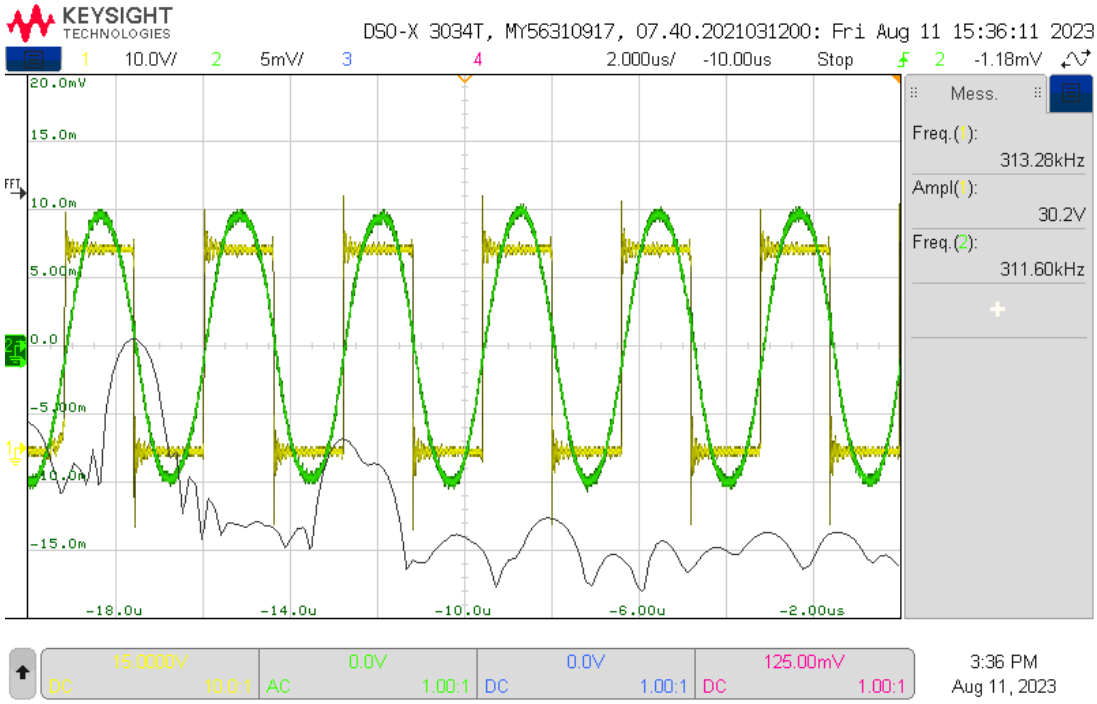


Figure 3.4: Waveform of transmission with a piezo (B) as load; channel 1: output of the MOSFET half-bridge, channel 2: signal generated by receiving piezo

3.2 Ultrasonic reception

The reception and processing of ultrasonic signals is in terms of hardware mainly enabled by the LNA, the ADC and the comparator and in terms of software by the modules vga_driver, adc_driver and comp_driver along its instantiated modules.

For the first test, a signal generator was connected to the BNC connector and set up to generate a sine wave with a frequency of $2\,\mathrm{MHz}$ and an amplitude of $10\,\mathrm{mV}$. Figure 3.5 shows the differential output of the VGA. The output is taken from the BNC connectors connected to the oscilloscope with enabled AC coupling. The gain of the VGA is set to ’1111’ using the buttons. As seen, the amplified signal is not shifted compared to the source. This would not be a concern however, as there is no need to be in phase with the source, either way the digitization with the ADC shifts the signal due to its 5.5 system clock cycle latency. The amplification matches the expected $43\,\mathrm{dB}$ of the VGA (LO gain mode) and the signal keeps its shape. One potential issue that became apparent during later ADC testing is that the gain increases linearly on a logarithmic scale. That means a binary gain setting of 8 amplifies the signal by a factor of 10 ($20\,\mathrm{dB}$) whereas a setting of 14 gives a factor of nearly 100 ($40\,\mathrm{dB}$). Depending on the signal range this can be desired, but it should be considered. Because of this, a more precise DAC would greatly enhance the gain setting.

For a more realistic testing, the signal generator has been connected to the secondary piezo element (B) and set up to generate a sine wave at the resonance frequency of $121\,\mathrm{kHz}$ with an amplitude of $5\,\mathrm{V}$ for the following tests. The primary piezo (B) is connected to the BNC connector of the PCB.

Testing the comparator, fig. 3.6 shows the amplified differential signal as well as the output signal of the comparator, sampled by the oscilloscope using a passive probe. The comparator output follows the received

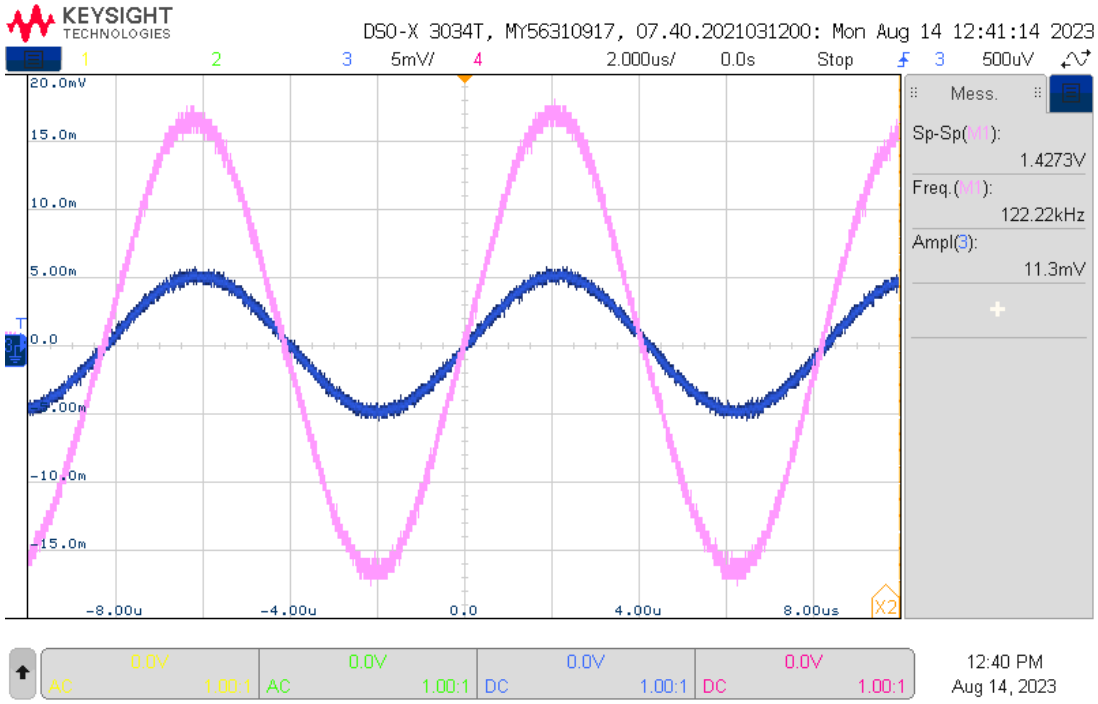


Figure 3.5: Waveform of VGA output; channel 3: output of the signal generator, pink waveform: differential VGA_OUT

and amplified signal closely. A small phase shift exists which is mainly due to the hysteresis setting, but the duration of the positive and negative half wave matches the pulse width of the comparator output.

The comparator_driver module samples the comparator output at $50\,\mathrm{MHz}$, counts the number of cycles for a signal switch and streams the data to an external computer. The left column of table 3.1a shows an excerpt of the output of the comparator after the bitstream has been converted to a readable format using a self-written Python script. As the values only show the clock cycles for a half-wave, two values have to be added in order to get the actual number of cycles for a complete signal period. The formula $f_{signal} = \frac{50\,\mathrm{MHz}}{num\_cycles_{lo\_hi} + num\_cycles_{hi\_lo}}$ yields $f_{signal} = 120.985\,\mathrm{kHz}$ averaged over the shown excerpt, matching the frequency set at the function generator. The right column of table 3.1a shows an excerpt when the clock is increased by $1\,\mathrm{kHz}$, showing a difference of 4 cycles on average for a signal period.

121 kHz	122 kHz
207	205
207	205
206	204
207	205
206	205
207	206
207	204
206	205
207	205
206	205
207	205
207	205

(a) comparator

121 kHz
-224
-213
-188
-152
-106
-54
2
57
109
153
187
208

(b) ADC

Table 3.1: Sampling excerpts

The ADC testing was performed by triggering on a threshold, with a sampling frequency of $3\,\mathrm{MHz}$, a sampling width of 10 bits and a sampling length of 40000 samples. As the output clamping level has been set such, that $V_{pp,diff} = 2.5\,\mathrm{V}$, the ADC currently does not make use of its full $\pm 4\,\mathrm{V}$ input range and is thus limited in resolution. In order to maximize the signal-to-noise ratio (SNR), the $10\,\mathrm{k\Omega}$ R_{clmp} resistor (R38) should be removed, as the differential peak-to-peak voltage is clamped to $V_{pp} = 4.5\,\mathrm{V}$ by the VGA anyway, eliminating the need for additional protection for the ADC and comparator.

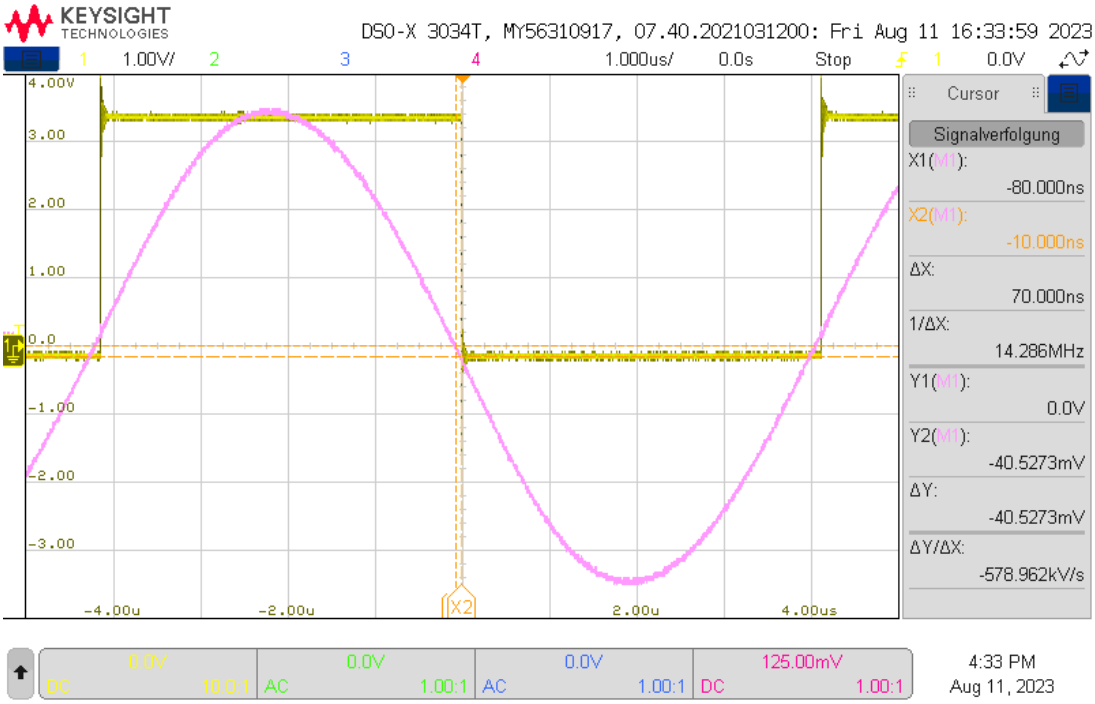


Figure 3.6: Waveform of comparator output; channel 1: output of the comparator, pink waveform: differential VGA_OUT

Table 3.1b shows an excerpt of the sampled integer values for a $121\,\mathrm{kHz}$ signal received from the piezo (B) and amplified to a differential peak-to-peak voltage of approximately $V_{pp,diff} = 1.75\,\mathrm{V}$, measured using the VGA_OUT+ and VGA_OUT- connectors. This should lead to sample values in the range of $\pm\frac{1.75\,\mathrm{V}}{4\,\mathrm{V}} * 1024/2 = \pm 224$, which can be confirmed by the received results. Figure 3.7 shows the sinusoidal signal reconstructed from the samples, while fig. 3.8 shows the respective Fourier transform.

3.3 PCB design

The PCB design uses a standard two layer stackup and was manufactured by JLCPCB. Before manufacturing, electrical rule checking (ERC) as well as design rule checking (DRC) was performed in order to verify the pin connections of the schematic as well as the design rules and schematic parity of the PCB layout. The only major issue within the first design was an incorrect pin order of the half-bridge MOSFETs. For testing purposes, this issue could be fixed manually by some sanding and the use of a single flying wire. The schematic and layout shown in chapter 2 show a fixed second iteration of the PCB schematic and layout. Other minor improvements of the second iteration are the increased footprint size of the LDO input and output capacitors, the addition of a BNC connector footprint and more footprints for the possibility of a more complex passive filter network at the ADC and comparator. A third iteration with additional footprints for an optional R-C snubber, further improved footprint sizes and capacitor discharge resistors has been uploaded to the project, but could not be reflected in the schematics of the report.

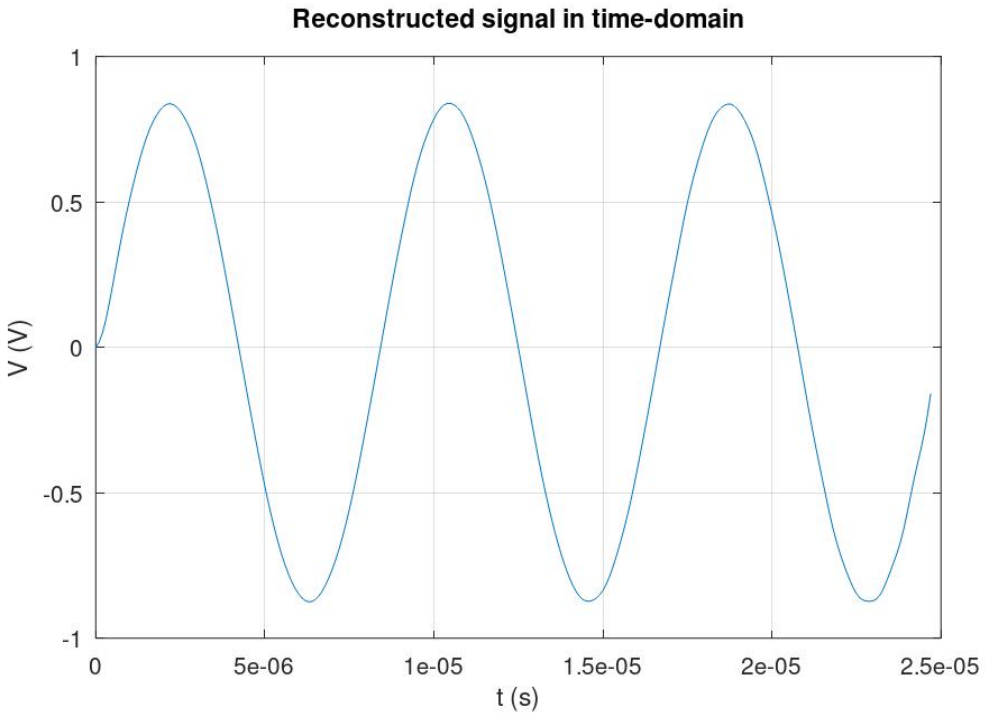


Figure 3.7: Reconstructed signal in time domain

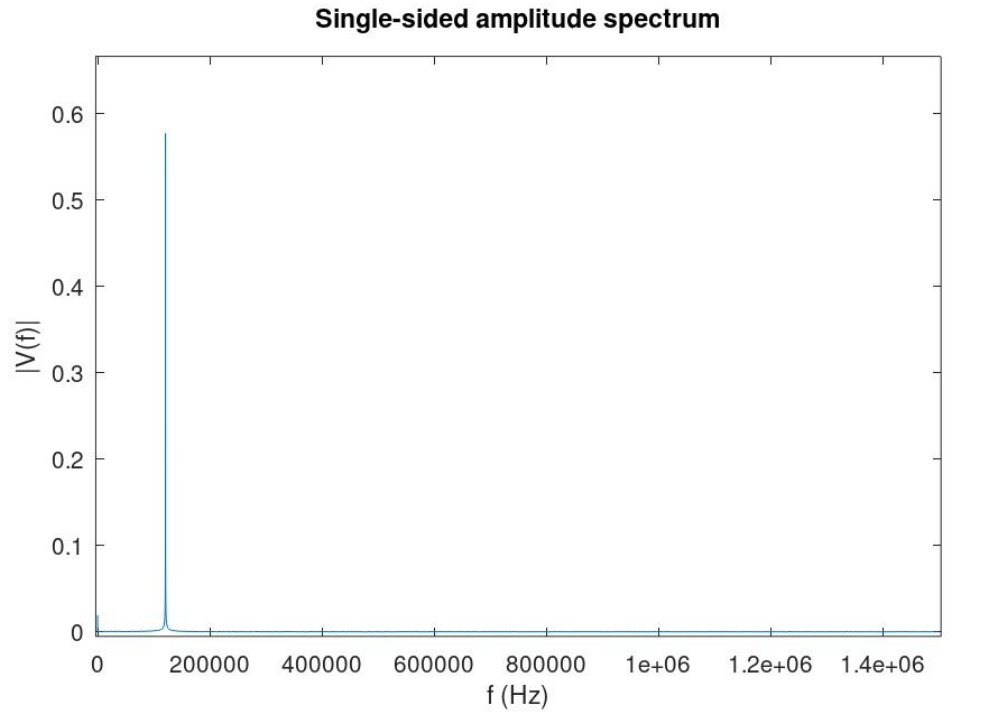


Figure 3.8: Fourier transform of sampled signal

3.4 FPGA logic

The FPGA logic was written in Verilog using Xilinx macros for clock generation, clock domain crossing and BRAM instantiation. All modules have been tested with the use of behavioral testbench simulations in Vivado and by examining the behavior when deployed to the FPGA and used for ultrasonic communication.

For easy usage without a host computer, the latest version of the implemented bitstream has been exported as a bin file and stored on the flash memory of the FPGA, such that it is loaded when powered on.

Serial communication

Most of the functionality is initiated by commands sent over the serial interface. Listing 3.1 shows the available commands as defined in serial_defines.hv. The serial interface works robustly and offers a convenient command structure. In order to add additional commands or to interface additional modules, only few changes in code are needed.

```
1  // command structure: "DESTINATION:COMMAND VALUE\n"
2  // e.g.               "VGA:GAIN 1\n"
3
4  /* Available Commands:
5  // VGA:POWER [0/1]: enables/disables vga.
6  // VGA:STATUS [<any number>]: return status message.
7  // VGA:GAIN [0..15]: sets the vga gain.
8  // VGA:RESET [<any number>]: resets signals
9
10 // COMM:START [<any number>]: starts the communication protocol with defined
       settings and data.
11 // COMM:TEST [0/1]: enables/disables the communication test state.
12 // COMM:STATUS [<any number>]: return status message.
13 // COMM:SETDTA [0..15]: sets the transmission data.
14 // COMM:SETCHP [0..4095]: sets the amount of charge pulses.
15 // COMM:SETSTL [0..65535]: sets the start delay in microseconds.
16 // COMM:SET0L [0..4095]: sets the amount of pulses for a 'zero' bit.
17 // COMM:SET1L [0..4095]: sets the amount of pulses for a 'one' bit.
18 // COMM:SETBRL [0..65535]: sets the break delay in microseconds.
19 // COMM:CLKDIV [0..511]: sets the piezo clock divider value.
20 // COMP:RESET [<any number>]: resets signals
21
22 // ADC:POWER [0/1]: enables/disables adc.
23 // ADC:STATUS [<any number>]: return status message.
24 // ADC:TRIG [0/1]: enable triggering according to threshold.
25 // ADC:THOLD [0..1023]: set threshold. IMPORTANT: signed number -> values >511
       are negative.
26 // ADC:MAXSMP [0..65535]: set maximum amount of samples. '0' equals maximum
       samples (18'd262.143).
27 // ADC:FORCE [<any number>]: force immediate triggering.
28 // ADC:FREQ [250..10000]: sets sampling frequency in kHz if input is valid. steps
       of 250 kHz.
29 // ADC:WIDTH [0..10]: sets sampling width.
30 // ADC:RESET [<any number>]: resets signals (TODO) and applies reset (emptiing
       fifos)
31 // ADC:SIM [0/1]: use simulation data instead of adc_in pins input
```

```
32
33 // COMP:POWER [0/1]: enables/disables comp.
34 // COMP:STATUS [<any number>]: return status message.
35 // COMP:MAXSMP [0..65535]: set maximum amount of samples. '0' equals maximum
       samples (18'd262.143).
36 // COMP:TRIG [0/1]: enable triggering according to adc threshold.
37 // COMP:FORCE [<any number>]: force immediate triggering.
38 // COMP:FREQ [25/50/100/200/300/400]: sets sampling frequency in MHz if input is
       valid. CURRENTLY REMOVED.
39 // COMP:WIDTH [0..12]: sets maximum counting width (ensure that the cycle count
       fits, otherwise overflows may occur)
40 // COMP:RESET [<any number>]: resets signals (TODO) and applies reset (emptiing
       fifos)
41 // COMP:SIM [0/1]: use simulation data instead of comp_in pins input
42 */
```

Listing 3.1: Command strings as defined in serials.hv

One drawback of the current UART implementation is the use of the $12\,\mathrm{MHz}$ system clock for sampling the incoming bitstream. As the signal is inherently asynchronous, the signal has to be oversampled in order to avoid sampling at the transitions which would likely result in bit errors. Currently, the uart_rx and uart_tx modules are set to receive or transmit a bit every 5 clock cycles, which leads to a effective bit rate of 2.4 Mbit/s, which has to be set in the terminal of choice, e.g. HTerm. As for each transferred byte there is one start and one stop bit, the effective data transfer rate is 1.92 Mbit/s.

Possible solutions for a higher bit rate would be to increase the system clock which may result in timing violations in other parts of the circuit or to use a separate clock which comes at the downside of dealing with clock domain crossing. More advanced FPGA modules such as the micro-nova Mercury 2 [28] also use FTDI FT2232H as an USB interface, but connect to its synchronous interfaces, which enables the use of synchronous communication with data rates of up to 200 Mbit/s. Using such a module would have eliminated the need for a large BRAM buffer and facilitated the transfer due to simple FIFO interfacing.

Ultrasonic transmission

The ultrasonic transmission logic located in the modules comm_protocol and piezo_driver enables a convenient data transfer. It currently employs one communication scheme with a fixed 4-bit transmission using OOK modulation as described in section 2.2.5. This modulation scheme enables simple demodulation logic by only counting the duration of signal presence or absence. The downside is the need of multiple carrier wave periods for a robust communication, making this scheme rather low speed.

The carrier frequency can be adjusted using the clock divider, that reduces the $250\,\mathrm{MHz}$ base frequency by waiting for a counter to reach the defined number. For a desired frequency, the clock divider can be calculated as follows:

$$f_{carrier} = \frac{1}{2} \cdot \frac{250\,\mathrm{MHz}}{clk\_div} \iff clk\_div = \frac{1}{2} \cdot \frac{250\,\mathrm{MHz}}{f_{carrier}}$$

The base frequency was set to $250\,\mathrm{MHz}$ as it is the maximum frequency at which all timing requirements can be met. In order to meet the timing, the FSM was stripped down as much as possible in order to achieve short combinatorial logic paths and low signal fan-out.

The approach of a clock divider was chosen in order to make the frequency adjustable and provide solutions for different transmission strategies. The step size of the transmission frequency is about $30\,\mathrm{kHz}$ at a frequency

of $2\,\mathrm{MHz}$, or 1.5%. While a separate MMCM could synthesize clocks more accurately, it takes many clock cycles to be reconfigured during runtime, making it unusable for a fast frequency-modulated communication in addition to being complex in code and thus not easily modifiable.

VGA

The logic for the vga_driver module is very simple as it only turns the VGA on and off and sets its gain. This digital setting of the amplification may be used however in order to enable feedback by the ADC.

ADC

The ADC logic provides all necessary functions to adjust the sampling in terms of sampling frequency, sampling width and sampling duration. It provides means to be triggered at a threshold level or to be force triggered. Also, it provides a *trigger_out* signal for triggering the comparator.

Sampled data is sent along with the information about its source, the frequency and the data width to the host computer where it can be processed further. At a maximum of 10 Msamples/s and a 10-bit output, the maximum bit rate needed for a transmission without buffering would be 100 Mbit/s.

The FPGA has a total of 1.62 Mbit of BRAM of which around 786 kbit are currently allocated for the ADC samples. With buffering that means $t_{sampling} = \frac{0.786 Mbit}{(100-1.92) Mbit/s} = 8.01\,\mathrm{ms}$. As can be seen, the data streaming makes only a relatively small difference of around $150\,\mu\mathrm{s}$ in regards of the maximum sampling duration. However, a decrease of the sampling frequency and the sample width can drastically increase the sampling duration, e.g. a halved sampling frequency yields a doubled sampling duration. Furthermore, a larger portion of the BRAM can be allocated to the ADC, as the comparator transmits much less data.

The adjustment of sampling frequencies and sampling width makes the module very adaptable in terms of input frequencies and sampling duration. The maximum input signal frequency is around $5\,\mathrm{MHz}$ following the Nyquist-Shannon theorem [29], which makes a wide array of piezo transducers usable with the designed system.

Comparator

The comparator logic shares most parts with the ADC logic. It provides similar reconfiguration options as the ADC module in terms of setting the sampling duration and the data width with the difference, that the sampling frequency is set to a fixed $50\,\mathrm{MHz}$.

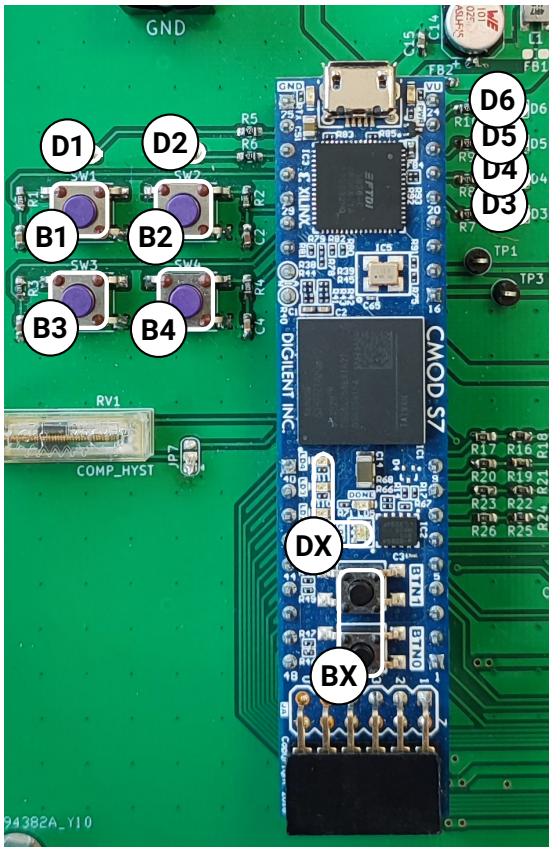
The data width needs to be set to a sufficient value in order not to overflow. For example, an incoming $2\,\mathrm{MHz}$ signal switches every 12.5 cycles of the $50\,\mathrm{MHz}$ sampling clock from '0' to '1' or vice versa, suggesting a sample width of at least 5 bits. The frequency can be calculated by using $f_{signal} = \frac{50\,\mathrm{MHz}}{num\_cycles_{lo\_hi} + num\_cycles_{hi\_lo}}$. With the sampling frequency of $50\,\mathrm{MHz}$, the detectable change in frequency around $2\,\mathrm{MHz}$ is about $80\,\mathrm{kHz}$. Increasing the sampling frequency and thus reducing the detectable frequency step size would require omitting the variable bit width in the fifo_aggregate module, as the high number of possibilities and as such high fan-out of the signals does not allow for a substantial increase in clock frequency as-is. This change is relatively easy feasible as it would greatly reduce the complexity of the logic.

A strongly suggested improvement, that should be added to the logic is a timeout and overflow detection. This could be easily achieved by adding following statement to the comp_driver module:

```
536 if (cycle_counter == 0) state_sample <= SAMPLE_RST; // cycle_counter is never 0
      except when overflow occured
```

Due to time constraints, this suggested addition could not yet be implemented and tested.

User I/O



- B1: Starts a predefined data or test transmission (depending on Verilog flag)
- B2: Enables/disables the VGA.
- B3: Decreases the gain by 1.
- B4: Increases the gain by 1.
- BX: Unassigned push buttons.
- D1: Indicates a running communication.
- D2: Indicates the status of the VGA (on/off).
- D3: Bit 0 (LSB) of the VGA gain.
- D4: Bit 1 of the VGA gain.
- D5: Bit 2 of the VGA gain.
- D6: Bit 3 (MSB) of the VGA gain.
- DX: Unassigned LEDs.

Figure 3.9: Overview of I/O

The PCB and FPGA offer some basic I/O for human operators, namely push buttons and LEDs. While not all have been assigned to a function, the buttons and LEDs enable the transmission of the predefined data and setting up the VGA to output the amplified differential signal over the BNC connectors, as seen in section 3.4. This enables all key functions which are not dependent on a host computer.

4 Conclusion

This report shows the design and development of a test PCB for ultrasonic communication. Its intended use is the support and testing of a custom ASIC which is powered and operated by ultrasonic communication using piezoelectric transducers.

The resulting implementation features data transmission using a simple OOK encoding and data reception, sampling and forwarding with the use of an amplifier, a comparator, an ADC and a FPGA as its logic core. The platform can be operated in its most basic functionality with its push button and LED interface and offers high configurability and sampling features when paired with a host computer. The configurability makes it suitable for interfacing with a variety of piezo elements, only limited by a maximum reception frequency of $5\,\mathrm{MHz}$ in the case of the ADC, for long sampling durations in the region of up to a few dozen milliseconds and different communication mechanisms. The Verilog code for the FPGA logic was designed in order to be as modular as possible in terms of host computer communication and to offer a modifiable basis for different transmission and sampling strategies.

Further work may be directed at implementing these strategies and enhancing the platform through higher sampling frequencies for the comparator and ADC and by increasing the transmission speeds to the host computer if needed. Suggestions for such changes were stated in the previous chapter.

Bibliography

[1] W. Dargie and C. Poellabauer, *Fundamentals of Wireless Sensor Networks: Theory and Practice* (Wiley Series on Wireless Communications and Mobile Computing). Chichester, West Sussex, U.K. ; Hoboken, NJ: Wiley, 2010, 311 pp., ISBN: 978-0-470-99765-9.

[2] C. R. Valenta and G. D. Durgin, "Harvesting Wireless Power: Survey of Energy-Harvester Conversion Efficiency in Far-Field, Wireless Power Transfer Systems," *IEEE Microwave Magazine*, vol. 15, no. 4, pp. 108–120, Jun. 2014, ISSN: 1557-9581. DOI: 10.1109/MMM.2014.2309499.

[3] A. A. Vives, Ed., *Piezoelectric Transducers and Applications*. Berlin, Heidelberg: Springer, 2008, ISBN: 978-3-540-77507-2 978-3-540-77508-9. DOI: 10.1007/978-3-540-77508-9.

[4] X. Wang, "Piezoelectric and Piezotronic Effects in Energy Harvesting and Conversion," in *Nanotechnology for the Energy Challenge*, John Wiley & Sons, Ltd, 2013, pp. 89–132, ISBN: 978-3-527-66510-5. DOI: 10.1002/9783527665105.ch4.

[5] A. Arnau and D. Soares, "Fundamentals of Piezoelectricity," in *Piezoelectric Transducers and Applications*, A. A. Vives, Ed., Berlin, Heidelberg: Springer, 2008, pp. 1–38, ISBN: 978-3-540-77508-9. DOI: 10.1007/978-3-540-77508-9_1.

[6] M. Griner. "Piezoelectric materials," ZfP - TUM Wiki. (Jul. 2016), [Online]. Available: https://wiki.tum.de/display/zfp/Piezoelectric+materials (visited on 08/09/2023).

[7] H. Kim, Y. Tadesse, and S. Priya, "Piezoelectric Energy Harvesting," in *Energy Harvesting Technologies*, S. Priya and D. J. Inman, Eds., Boston, MA: Springer US, 2009, pp. 3–39, ISBN: 978-0-387-76464-1. DOI: 10.1007/978-0-387-76464-1_1.

[8] R. Lucklum, D. Soares, and K. Kanazawa, "Models for Resonant Sensors," in *Piezoelectric Transducers and Applications*, A. A. Vives, Ed., Berlin, Heidelberg: Springer, 2008, pp. 63–96, ISBN: 978-3-540-77508-9. DOI: 10.1007/978-3-540-77508-9_3.

[9] Microchip Technology Inc. "N-Channel and P-Channel Enhancement-Mode MOSFET Pair." TC6320 datasheet, (2017), [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/20005697A.pdf (visited on 08/07/2023).

[10] ON Semiconductor Corporation. "N-Channel PowerTrench® MOSFET." FDD1600N10ALZ datasheet, (2014), [Online]. Available: https://www.onsemi.com/download/data-sheet/pdf/fdd1600n10alz-d.pdf (visited on 08/07/2023).

[11] Microchip Technology Inc. "85V Half-Bridge MOSFET Driver with up to 16V Programmable Gate Drive." MIC4604 datasheet, (2018), [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/20005852A.pdf (visited on 08/07/2023).

[12] Microchip Technology Inc. "Single/Dual-Channel High-Voltage Protection T/R Switch." MD0100 datasheet, (2018), [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/MD0100-Single-and-Dual-ChannelHigh-Voltage-Protection-TR-Switch-Data-Sheet-20005738A.pdf (visited on 08/07/2023).

[13] Vishay Semiconductor. “Small Signal Switching Diode, Dual in Series.” BAV99 datasheet, (2018), [Online]. Available: https://www.vishay.com/docs/85718/bav99.pdf (visited on 08/07/2023).

[14] Analog Devices. “Ultralow Noise VGAs with Preamplifier and Programmable RIN.” AD8331 datasheet, (2016), [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/AD8331_8332_8334.pdf (visited on 08/07/2023).

[15] Analog Devices. “10-Bit, 10Msps ADC.” in collab. with MAX1426 datasheet. (2011), [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/MAX1426.pdf (visited on 08/07/2023).

[16] Analog Devices. “280MHz, 2.9ns Comparator Family with Rail-to-Rail Inputs and CMOS Outputs.” LTC6752 datasheet, (2014), [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/6752fc.pdf (visited on 08/07/2023).

[17] Analog Devices. “Biasing and Decoupling Op Amps in Single Supply Applications.” AN581 application note, in collab. with C. Kitchin. (), [Online]. Available: https://www.analog.com/media/en/technical-documentation/application-notes/AN-581.pdf?doc=AD7699.pdf (visited on 08/07/2023).

[18] Digilent. “Breadboardable Spartan-7 FPGA Module.” CMOD S7 overview, (), [Online]. Available: https://digilent.com/reference/programmable-logic/cmod-s7/start (visited on 08/07/2023).

[19] nexperia. “100 V, 10 A low leakage current Trench MEGA Schottky barrier rectifier.” PMEG100T100ELPE datasheet, (2020), [Online]. Available: https://assets.nexperia.com/documents/data-sheet/PMEG100T100ELPE.pdf (visited on 08/07/2023).

[20] ON Semiconductor Corporation. “1.0 A Low-Dropout Positive Fixed and Adjustable Voltage Regulators.” NCP1117 datasheet, (2017), [Online]. Available: https://www.onsemi.com/pdf/datasheet/ncp1117-d.pdf (visited on 08/07/2023).

[21] M. Mardiguian, *Controlling Radiated Emissions by Design*. Cham: Springer International Publishing, 2014, ISBN: 978-3-319-04770-6 978-3-319-04771-3. DOI: 10.1007/978-3-319-04771-3.

[22] Nandland. “UART in VHDL and Verilog for an FPGA,” Nandland. (Jun. 9, 2022), [Online]. Available: https://nandland.com/uart-serial-port-module/ (visited on 08/07/2023).

[23] Maxim Integrated. “Determining Clock Accuracy Requirements for UART Communications.” AN2141 tutorial, (Aug. 7, 2003), [Online]. Available: https://pdfserv.maximintegrated.com/en/an/AN2141.pdf (visited on 08/03/2023).

[24] Xilinx. “Vivado Design Suite 7 Series FPGA and Zynq 7000 SoC Libraries Guide.” UG953 user guide, (2023), [Online]. Available: https://docs.xilinx.com/r/en-US/ug953-vivado-7series-libraries (visited on 08/07/2023).

[25] Xilinx. “7 Series FPGAs Data Sheet: Overview.” DS180 data sheet, (2020).

[26] Maxim Integrated. “I’m OOK. You’re OOK?” AN4439 application note, (Apr. 8, 2009), [Online]. Available: https://www.analog.com/media/en/technical-documentation/tech-articles/im-ook-youre-ook.pdf (visited on 08/07/2023).

[27] John Betten. “Power Tips: Calculate an R-C snubber in seven steps - Power management - Technical articles - TI E2E support forums.” (May 5, 2016), [Online]. Available: https://e2e.ti.com/blogs_/b/powerhouse/posts/calculate-an-r-c-snubber-in-seven-steps (visited on 08/08/2023).

[28] micronova. “Xilinx Artix-7 FPGA development board.” Mercury 2 overview, (2023), [Online]. Available: https://www.micro-nova.com/mercury-2 (visited on 08/07/2023).

[29] A. Jerri, “The Shannon sampling theorem—Its various extensions and applications: A tutorial review,” *Proceedings of the IEEE*, vol. 65, no. 11, pp. 1565–1596, Nov. 1977, ISSN: 1558-2256. DOI: 10.1109/PROC.1977.10771.

List of Figures

Listings