DIPLOMARBEIT

FPGA-basiertes RISC-V-Computersystem: YARM

Höhere Technische Bundeslehr- und Versuchsanstalt Anichstraße

Abteilung

Elektronik und technische Informatik

Ausgeführt im Schuljahr 2019/20 von:

Armin Brauns 5AHEL
Daniel Plank 5BHEL

Betreuer/Betreuerin:

Dipl.-Ing. Christoph Schönherr

Projektpartner: IT-Syndikat, Verein zur Förderung des freien Zugangs zu technischer Fort- und Weiterbildung jeglicher Art, Hackerspace Innsbruck

Ansprechpartner: Ing. David Oberhollenzer B.Sc.

Innsbruck, am 29. März 2020

Abgabevermerk:

Datum:

Betreuer/in:

Gendererklärung

Aus Gründen der besseren Lesbarkeit wird in dieser Diplomarbeit die Sprachform des generischen Maskulinums angewendet. Es wird an dieser Stelle darauf hingewiesen, dass die ausschließliche Verwendung der männlichen Form geschlechtsunabhängig verstanden werden soll.

Kurzfassung/Abstract

Diese Diplomarbeit beschäftigt sich mit der Arbeitsweise von Prozessoren und Prozessorperipherie in moderner und traditioneller Form. Sie versucht anschaulich den Aufbau eines Computersystems in Hard- und Software zu veranschaulichen sowie diesen zu erklären. Dafür wurde auf einem XILINX FPGA ein RISC-V32I Prozessor in VHDL implementiert sowie diverse Parallelbus gebundene Hardwareperipherie entwickelt und gebaut. Als Harwareperipherie wurde ein 8-Bit 2-Kanal DAC und eine serielle Schnittstelle mit TIA-/EIA-232 Pegeln gewählt. Der Prozessor implementiert das RISC-V32I base instruction set. Aufgrund der starken Verwendung von Englisch im Software- und Hardwarebereich wurde diese Diplomarbeit in Englisch verfasst, was ebenfalls die Lesbarkeit erhöhen soll. Die entstandene Dokumentation soll für Menschen mit einem grundlegenden Verständnis von Elektronik sowie der Hardware-Beschreibungssprache VHDL verständlich sein.

This diploma thesis deals with the operation of processors and their corresponding peripherials in modern and traditional forms. It attempts to illustrate the structure of a computersystem in hard- and software. To reach this goal a RISC-V32I processor has been implemented in VHDL on a XILINX FPGA as well as some peripherials bound to the parallel bus. These peripherials include a 2-channel 8-bit Digital to analog converter as well as a TIA-/EIA-232 compliant serial interface. Due to the common use of english in the hardware and software engineering field this thesis was written in english, which should enhance readability as well. The written documentation should be understandable for everyone with a basic understanding of electronics as well as the hardware description language VHDL.

Result

The project was fully implemented with all functionality originally targeted. The system has been tested and verified and all example code have been documented and tested as running. Implementations in hardware were made in open-source programs and the RISC-V processor can compile using an open source toolchain. The completed project can be found on the USB stick which accompanies this thesis, or in the git repositories at `https://git.it-syndikat.org/tyrolyean/dipl.git` and `https://gitlab.com/YARM-project/`. The completed hardware peripherials can be seen in figure i

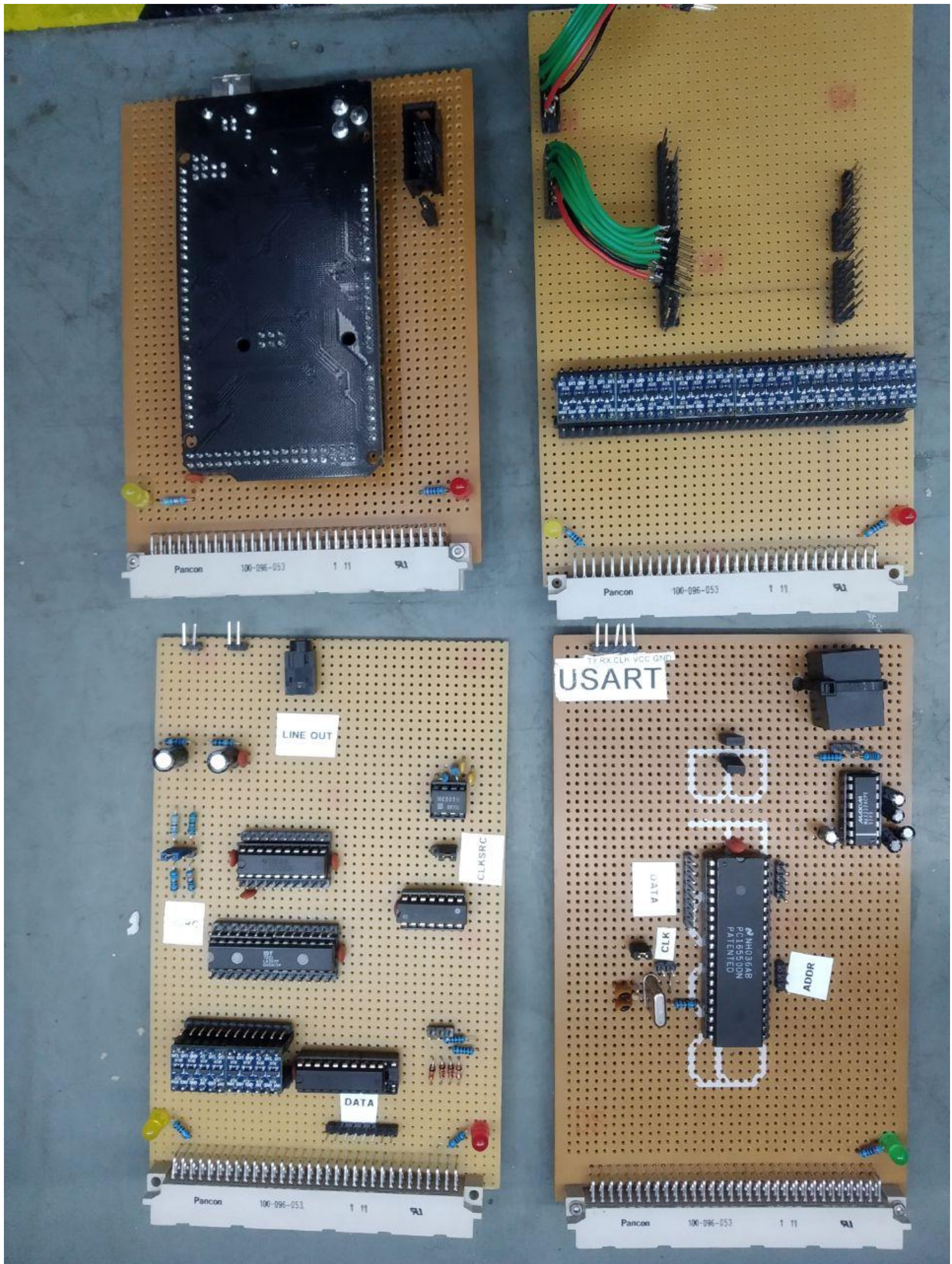

Figure i: An overview of the hardware peripherials

Contents

Gendererklärung i
Kurzfassung/Abstract ii
Result iii
1 Task description 1
1.1 Hardware 1
2 Hardware peripherials 2
2.1 Parallel bus 2
2.1.1 Address Bus 3
2.2 Data Bus 4
2.3 Control Bus 4
2.3.1 Master Reset 4
2.3.2 Write Not 4
2.3.3 Read Not 4
2.3.4 Module Select 1 and 2 Not 5
2.4 Von Neumann Archtiecture 5
2.5 Testing and Measurement 6
2.5.1 Measurements 6
2.5.2 Testing 7
2.6 Backplane 9
2.6.1 Termination resistors 9
2.7 Case 10
2.8 Serial Console 12
2.8.1 16550 UART 12
2.8.2 MAX-232 13
2.8.3 Schematics 13
2.8.4 Demonstration Software 17
2.8.5 Final Module 21
2.9 Audio Digital-Analog-Converter 23
2.9.1 TLC 7528 Dual R2R Ladder DAC 23
2.9.2 IDT7201 CMOS FIFO Buffer 24
2.9.3 Theory verfication 24
2.9.4 Schematics 25
2.9.5 DAC Module Read 28
2.9.6 Demonstration Software 28
3 Addressing DACA and DACB 32
3.0.1 Final Module 33

3.1 FPGA to Hardware interface ... 34
3.1.1 Measurement error ... 37
3.1.2 Final Module ... 37

4 Textadventure ... **39**
4.1 General Implementation details ... 39
4.1.1 General definitions and pinout of the AVR ... 39
4.1.2 Read and Write routines ... 41
4.1.3 UART and DAC update polling ... 41
4.1.4 Program execution path ... 42
4.2 DAC sound generation ... 44
4.2.1 DAC modes ... 44
4.2.2 Tones and Tracks ... 48
4.2.3 Track switching ... 52
4.3 User command interpretation ... 52
4.3.1 Command structure and parsing ... 52
4.3.2 Command parameters ... 54
4.4 Gameplay ... 55
4.4.1 Memory constraints ... 57
4.4.2 Story ... 57
4.4.3 Recursion ... 57
4.4.4 Computer State Machine ... 57

I A short introduction to VHDL **60**

5 Prerequisites ... **60**

6 Creating a design ... **60**

7 Simulating a design ... **62**

8 Synthesizing a design ... **63**

II Meta **64**

9 History ... **64**

10 Tooling ... **66**
10.1 Vendor Tools ... 66
10.2 Free Software Tools ... 66

11 Peripherals ... **67**
11.1 UART ... 67
11.2 DVI graphics ... 67

11.2.1 VGA timing 68
11.2.2 Text renderer 69
11.2.3 TMDS encoder 70
11.3 Ethernet 70
11.4 WS2812 driver 70
11.5 DRAM 72
11.6 External Bus 72
12 Testing **72**
12.1 RISC-V Compliance Tests 72

III The Core **73**

13 Overview **74**
14 Control **74**
15 Decoder **75**
16 Registers **76**
17 Arithmetic and Logic Unit (ALU) **76**
18 Control and Status Registers (CSR) **77**
19 Memory Arbiter **77**
20 Exception Control **78**
21 Erklärung der Eigenständigkeit der Arbeit **80**
I List of Figures ..... **I**
II List of Tables ..... **II**
III Listings ..... **II**
Anhang ..... **VI**

1 Task description

1.1 Hardware

Due to the recurring questions in the environment of the Hackerspace Innsbruck about the internal workings of a computer system and the lack of material to demonstrate these, hardware should be developed for educational purposes. This hardware should not be too complex to understand but still demonstrate basic tasks of a computer system. The targeted computing tasks are human interface device controllers, under which a **D**igital to **A**nalog **C**onverter[A] and a serial console with TIA-/EIA-232 compliant voltage levels were chosen. For these peripherials schematics and a working implementation in the hardware building style of the hackerspace should be built. All nescessary hardware will be provided by the Hackerspace. If possible already present hardware should be used, if impossible new one will be ordered. All schematics should, where possible be written in open-source software such as Kicad or GNU-EDA.

If possible software-examples should be written as well, though the complexity of these are coupled to the time left to spend on the project. Software should be written in C, the coding convention is left to the implementer.

[A] From now on reffered to simply as DAC

2 Hardware peripherials

2.1 Parallel bus

The core part of the hardware is the interface between the microprocessor and the hardware peripherials. This bus is delivering data in parallel and is therefore named the “parallel bus“. This bus has 3 different sub-parts:

1. The address bus
2. The data bus
3. The control bus

This split is common in many computer architectures and bus systems used by various microprocessor manufacturers. In figure ii the layout of the Atari Parallel Bus Interface is shown as used on the Atari 800XL.

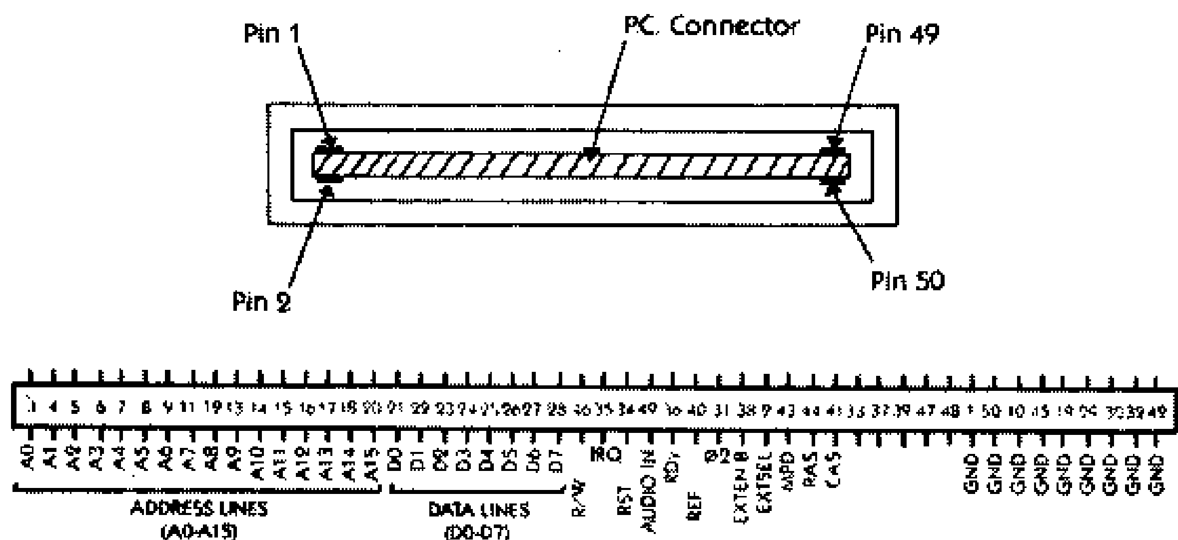

Figure 4.
Parallel Bus Pinout

Figure ii: Atari PBI Pinout;Source: `https://www.atarimagazines.com`

System Bus In some architectures the backbone parallel bus consisting of data- address- and control bus is called the system bus. The system bus even has its own

wikipedia article [B] and the picture seen in figure iii, which has been taken from this wikipedia article, even shows the exact same parts. However the origin of this term could not be determined and its use was the most common when describing the interface between the fabric of the CPU with external parts via this interface on a motherboard, which ran on system clock speed and was synchronized with the processor. The term parallel bus was chosen for this thesis because the bus runs on an independant clock speed and only interacts with the processor asynchronous to its clock. The term front side bus would be more fitting but not used because of its affiliation with intel products.

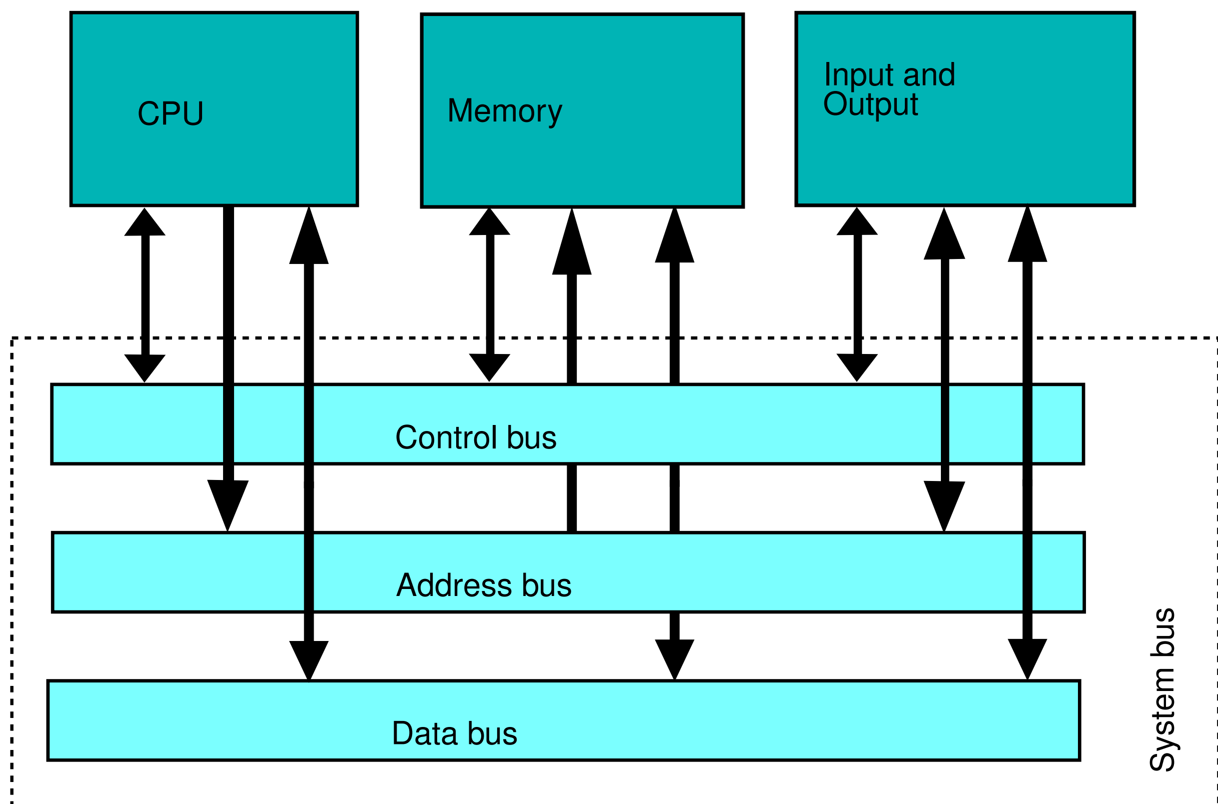

Figure iii: System bus structural diagram; Source: https://en.wikipedia.org/

2.1.1 Address Bus

The address bus contains the nescessary data lines for addressing the individual registers of the Serial connection and the UART. On any modern system this bus is from 16 to 64 bits wide. For our implementation the bus size was chosen to be 8 bit, which is multiple times the amount of needed address space, but is the smallest addressable unit on most microcontroller architectures and therefore easy to program with. The address bus is unidirectional.

[B] https://en.wikipedia.org/wiki/System_bus

2.2 Data Bus

The data bus contains the actual data to be stored to and read from registers. The data bus is, as well on most systems a multiple of 16 bits wide, but for the same reasons as the data bus, was shrunk down in our case to 8 bits. The data bus is bidirectional.

2.3 Control Bus

Control bus is a term which referes to any control lines (such as read and write lines or clock lines) which are neither address nor data bus. The control bus in our case is 5 bits wide and consists of:

- MR ... Master Reset
- $\neg WR$ ... Write Not
- $\neg RD$ ... Read Not
- $\neg MS1$ ... Module Select 1 Not
- $\neg MS2$ ... Module Select 2 Not

2.3.1 Master Reset

A high level on the MR lane signals to the peripherials that a reset of all registers and states should occure. This is needed for the serial console and the DAC.

2.3.2 Write Not

A low level on the $\neg WR$ lane signals the corresponding modules that the data on the data bus should be written to the register on the address specified from the address bus.

2.3.3 Read Not

A low level on the $\neg RD$ lane signals the corresponding modules that the data from the register specified by the address on the address bus should be written to the data bus.

2.3.4 Module Select 1 and 2 Not

A low level on one of these lines signals the corresponding module that the data on address data and the control lines is meant for it.

2.4 Von Neumann Archtiecture

The term “von Neumann architecture“ referrs to a type of computer architecture which referres to almost any modern computer system. It describes the in this thesis used Human input and output parts and the general workings of modern processors with the ALU[C] or the CA [D] as well as means to interface with its operator[1].

In his thesis “First Draft of a Report on the EDVAC“ he writes about human input:

“Once these instructions are given to the device, it must be able to carry them out completely and without any need for further intelligent human intervention. At the end of the required operations the device must record the results again in one of the forms referred to above.“[1, p.7]

This can be applied to the hardware implemented in this thesis, as well as other general computing systems. The EDVAC, which his thesis referres to, was a computer developed for military purposes. Much like the EDVAC, the CPU in this thesis is responsible for arithemtic operations and code interpetation. The peripherials are what is referred to as the input and output devices in his report. Though the for examples used ATMega2650 utilizes a harvard architecture “In order to maximize performance and parallellism“[2, p.11] the more general descriptions of computational operations still apply to this thesis. The differences between a harvard architecture and a von neumann architecture are shown in figure iv

[C] ALU...arithmetic logic unit
[D] CA...Central Arithmetic Part

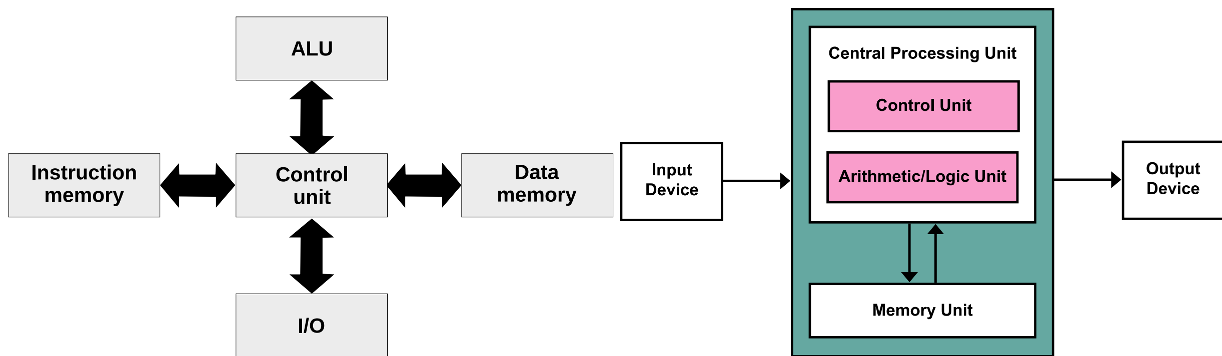

Figure iv: Harvard(left) vs Von-Neumann architecture(right);
Source: `https://en.wikipedia.org/`

2.5 Testing and Measurement

For functional testing and verification of implementation goals measurements needed to be performed in various different ways and testing software was required.

2.5.1 Measurements

Measurements were performed, if not noted otherwise, with the Analog Discovery 2 from Digilent as it has 16bit digital I/O Pins as well a a Waveform generator and 2 differential oszilloscope inputs[3]. These were enough for all nescessary measurements. Though due to the size and construction of the device, which can be seen in figure v, errors were encountered while performing the measurements. These are noted on occurance.

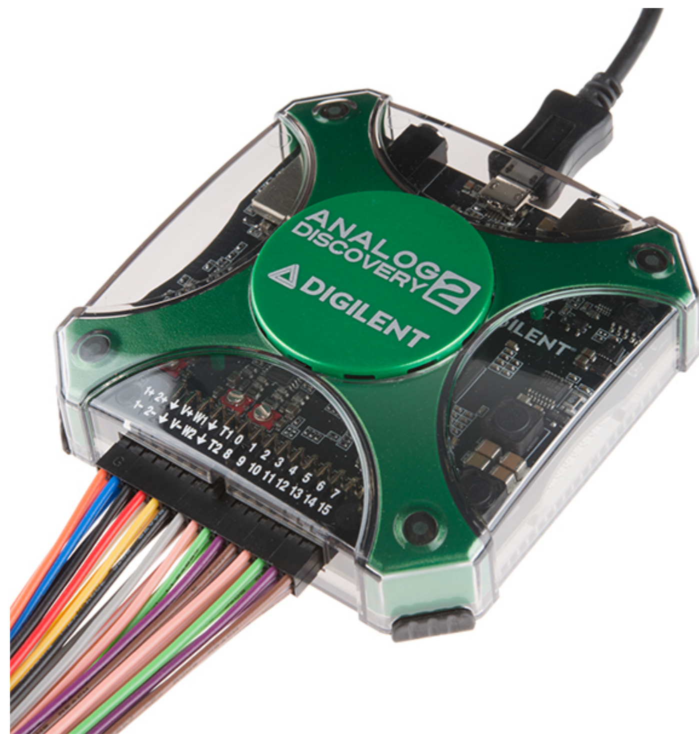

Figure v: Digilent Analog Discovery 2;Source: `https://www.sparkfun.com/`

2.5.2 Testing

All testing was performed with an Atmel ATMega2560 due to it's large amount of I/O pins, 5V I/O, which is the more common voltage level on CMOS peripherials, way of addressing pins (8 at a time) and availability. [2] All testing software was written for this ATMega and compiled using the avr-gcc from the GNU-Project.

To fully test the developed modules on the backplane a seprate module for the ATMega was developed, which can be seen in figure vi. The ATMega is beneath the the black PCB [E] in the center, which is an Arduino™Mega. The Arduino™is, for all indends and purposes, only a breakout of the ATMega 2560 and has only been used in that way. No parts of the Arduino™IDE or other parts of the Arduino™software suite have been used, as they consume too much memory and the abstraction models used are not compatiable with building processor peripherials.

[E]Printed circuit board

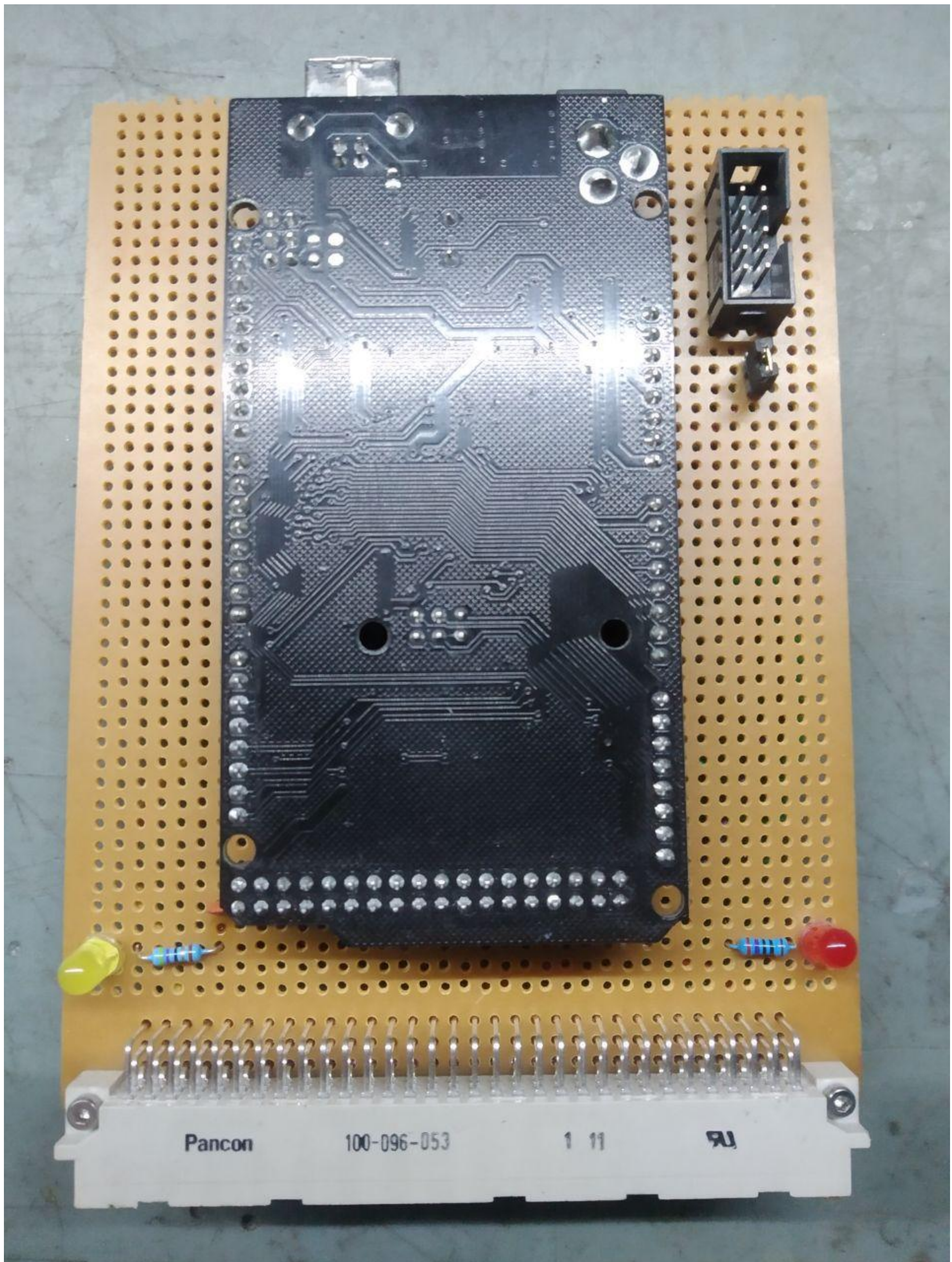

Figure vi: The ATMega 2560 module for the backplane

2.6 Backplane

To connect the modules to the microprocessor, many pins need to be connected straight through. For this purpose a backplane was chosen where DIN41612 connectors can be used. These connectors were chosen for their large pin count (96 pins) and their availability. The backplane connects all 96-pins straight through. With the 6 outer left and right pins connected for VCC and ground as can be seen in figure vii.

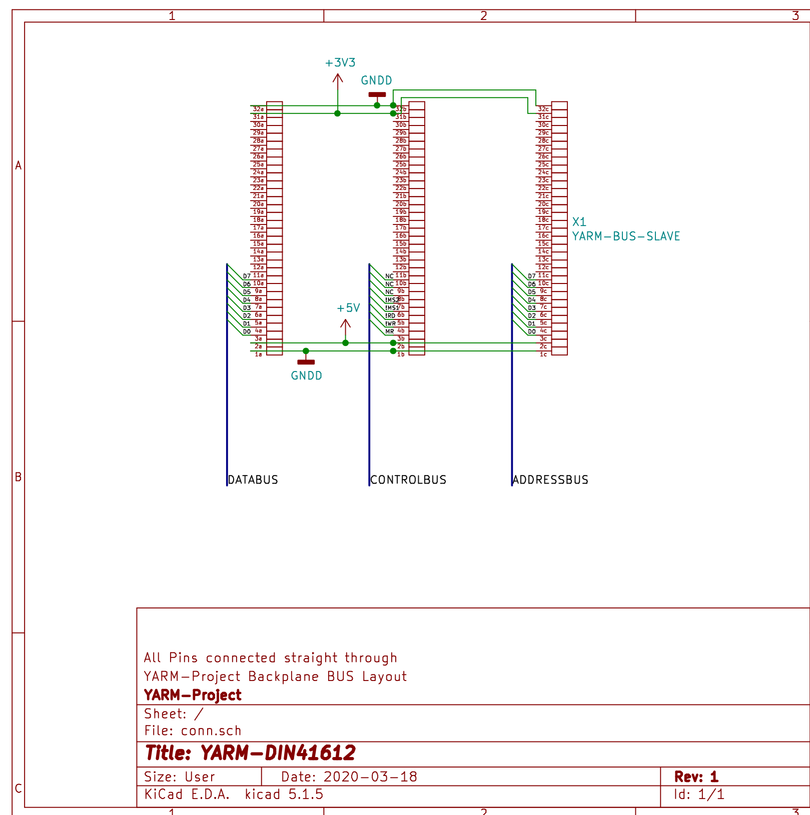

Figure vii: Layout of the DIN41612 Connectors on the Backplane

2.6.1 Termination resistors

In constrast to other systems using this backplane no termination resistors were used. This makes the bus more prone to refelctions, however these were not a problem during development with the maximum transmission rate of 1MHz, as can be seen in the sample recording in figure viii

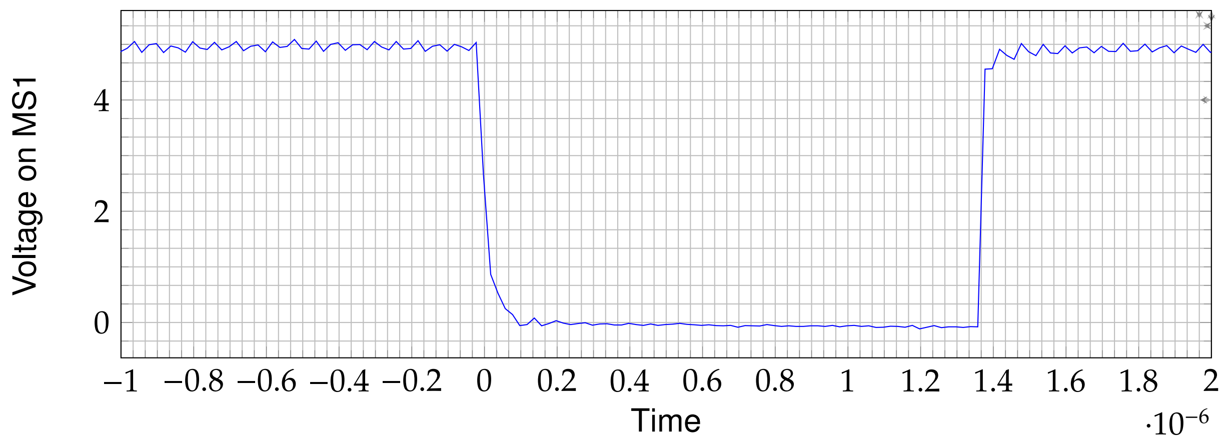

Figure viii: Measurement at around 1MHz bus clock on MS1

The ripple seen in figure viii is most likely due to the sample rate of the Oszilloscope, which is around 10Mhz after an average filter has been applied. The measurement was performed on the finished project with all cards installed.

2.7 Case

The case for the backplane was provided by the hackerspace and is meant for installation in a rack. The case is meant for installation of cards in the EUROCARD format, therefore all modules were built by this formfactor.

Figure ix: The case with installed backplane

2.8 Serial Console

One core part of any computer systems is it's way to get human input. On older systems, and even today on server machines, this is done via a serial console. On this serial console characters are transmitted in serial, which means bit by bit over the same line. The voltage levels used in these systems vary from 5V to 3.3V or +-10V. The most common standard for these voltage levels is the former RS-232[F] or as it should be called now, TIA-[G]/EIA-[H]232.[4] Voltage-levels ,as per TIA-/EIA- standard, are not practical to handle over short distances however, so other voltages are used on most interface chips and need to be converted.

2.8.1 16550 UART

The 16550 UART[I] is a very common interface chip for serial communications. It produces 5V logic levels as output on TX and needs the same as input on RX. Though common for a UART, these voltage levels need to be converted to TIA-/EIA-232 levels for a more common interface.

The 16550 UART is in it's core a 16450 UART, but has been given a FIFO [J] buffer. It needs three address lines, and 8 data lines, which can be seen in figure x

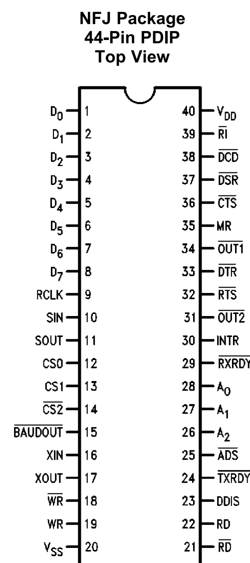

Figure x: PC-16550D Pinout[5]

[F] RS... Recommended Standard
[G] TIA...Telecommunications Industry Association
[H] EIA.. Electronic Industries Alliance
[I] Uinversal Asynchronous Receiver and Transmitter
[J] First-In First-Out

In figure x the most important lanes are the SIN and SOUT lanes, as they contain the serial data to and from the 16550 UART.

2.8.2 MAX-232

To convert the voltage levels of the 16550 UART to levels compliant with TIA-/EIA-232 levels the MAX-232 is used. It has two transmitters and two receivers and generates the needed voltage levels via an internal voltage pump[6].

2.8.3 Schematics

Based on the descriptions in the datasheets, the schematic in figure xi was developed.

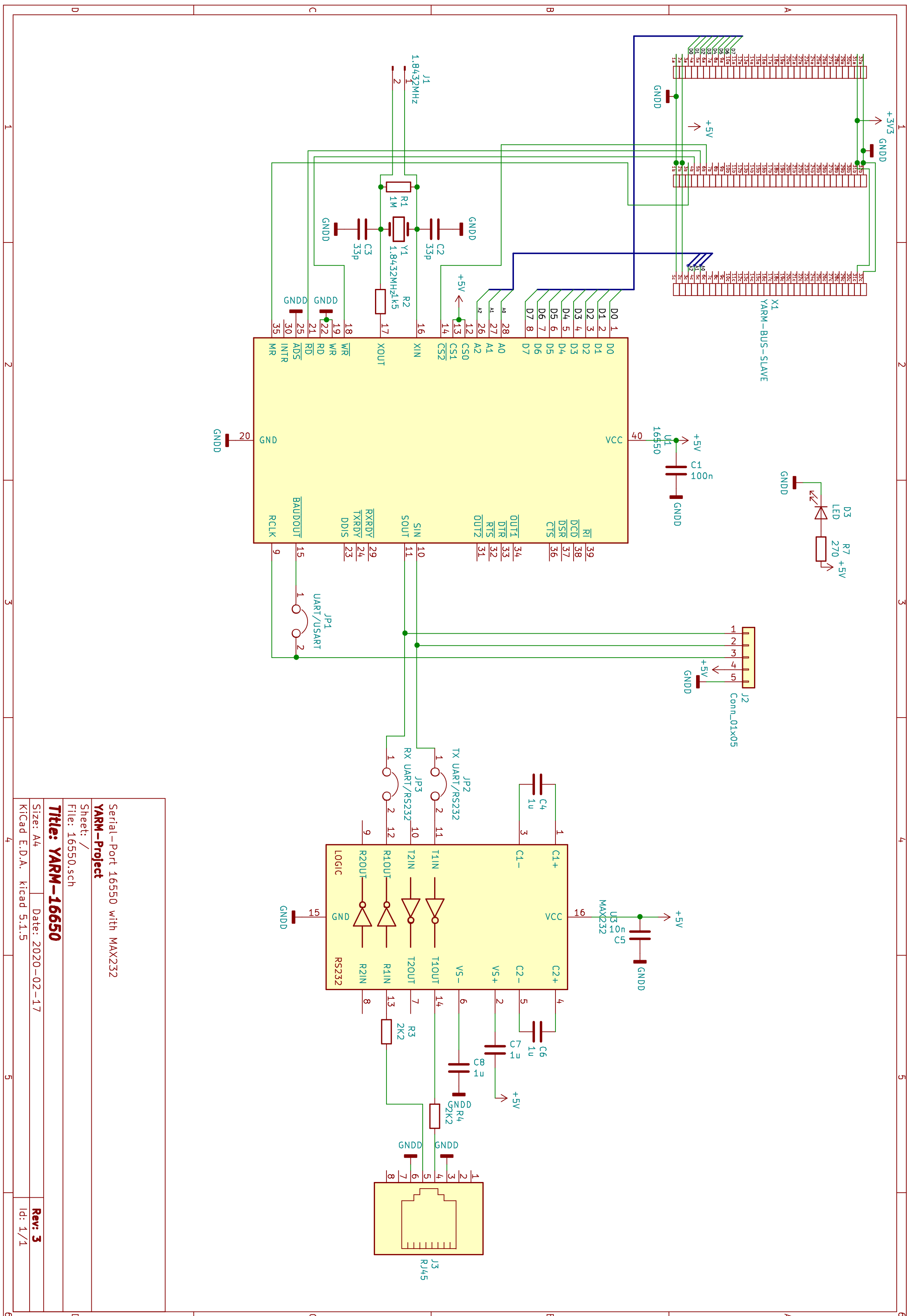

Figure xi: The schematic of the UART Module

Element Description The quartz oszillator Y1 is the clock source for the Baud Rate generation and was chosen with 1.8432 MHz for availability reasons and because it is the lowest ozillator from which all common baud rates can still be derived [5]. Resistors R1 and R2 are for stability and functionality of the Oszillator nescessary as per datasheet. The resulting frequency can be measured via J1 as can be seen in figure xii. C1 is used to stabilize the voltage for the 16550 UART and is common practice. Via JP1 the UART can be transformed into a USRT, where the receiver is synchronized to the transmitter via a clock line. This mode has, however, not been tested, and the clock needs to be 16 times the receiver clock rate[5]. The final output of the 16550 UART can be used and measured via J2, as shown in figure xiii . Before the UART on J2 can be used however, the Jumpers JP2 and JP3 need to be removed, as otherwise the MAX-232 will short out with the incoming signal. Capacitors C4, C6, C7 and C8 are for the voltage pump as defined in the datasheet[6]. R4 and R5 have been suggested by the supervisor in order to avoid damage to the MAX-232. The RJ-45 plug is used to transmit the TIA-/EIA-232 signal, rather than the more common D-SUB connector, because the RJ-45 connector fits on a 2.54mm grid. The Pinout of the RJ-45 plug can be seen in figure xiv. C5 has the same functionality for the MAX-232 as the C1 has to the 16550-UART.

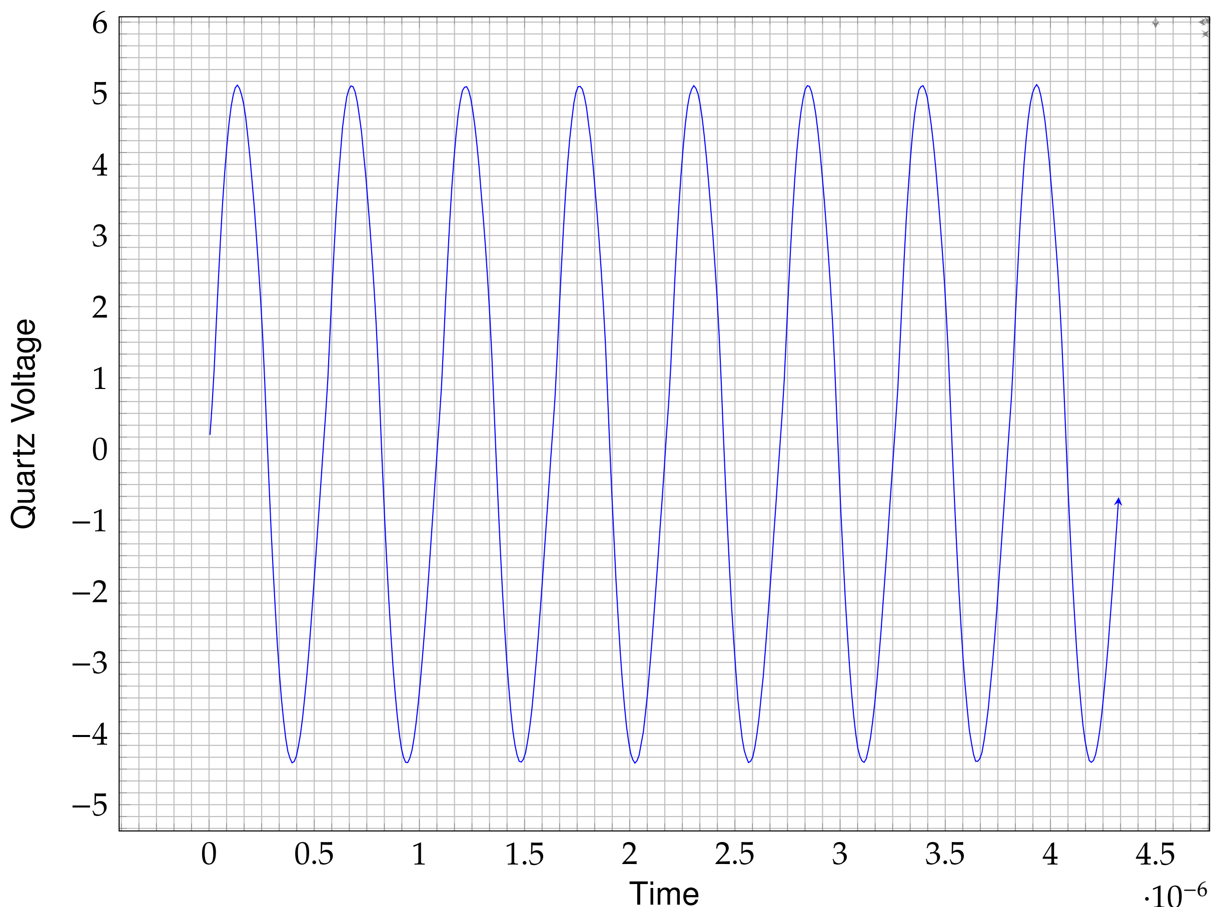

Figure xii: Measurement of the 1.8432 MHz Output on J1

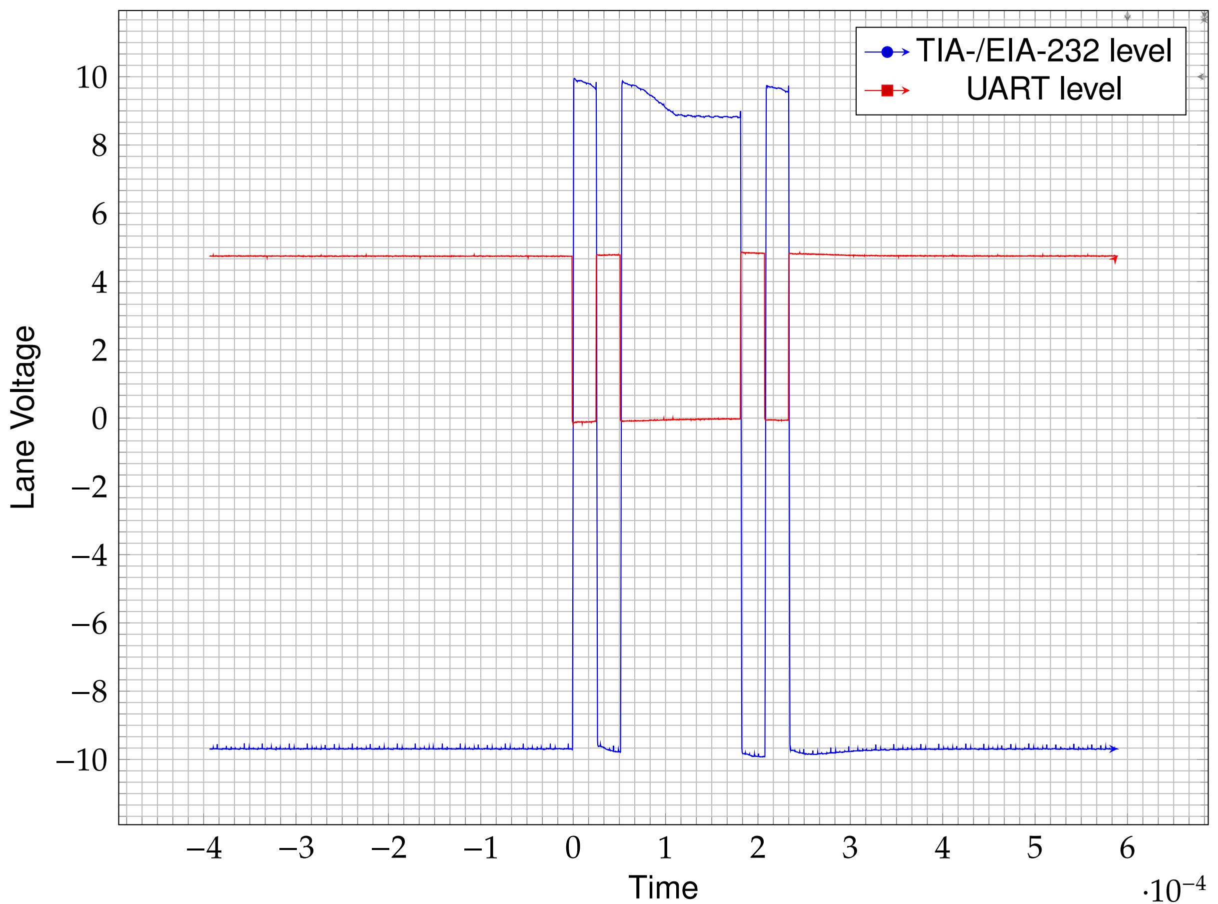

Figure xiii: Measurement of a character transmission before and after MAX-232

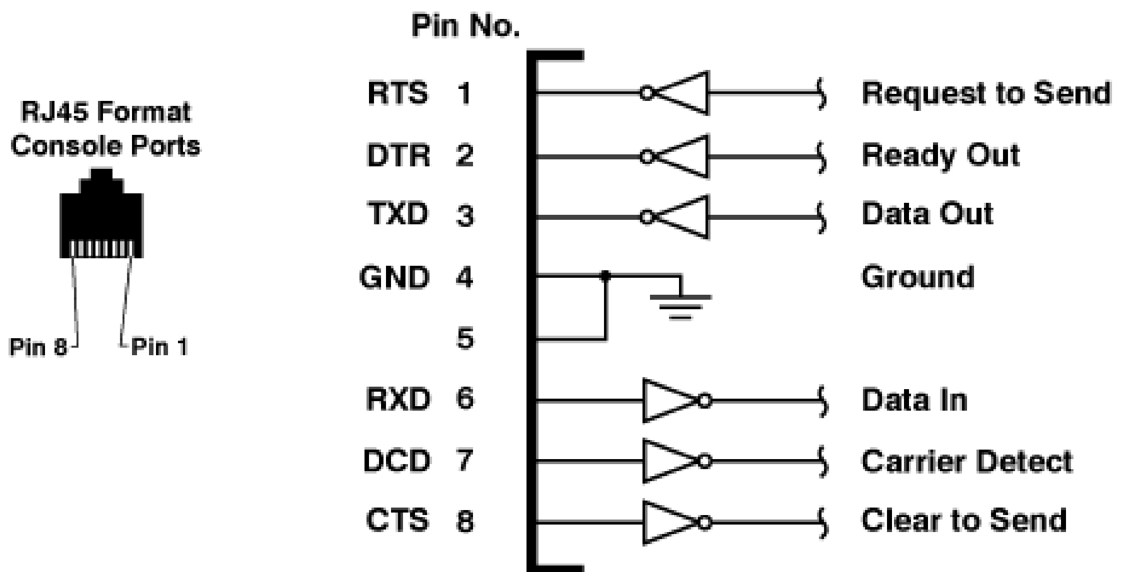

Figure xiv: Pinout of the RJ-45 Plug; Src: `https://www.wti.com/`

2.8.4 Demonstration Software

To demonstrate the functionality and prove that the schematic has no underlying error, a program which regularly transmits a character was written as well as a simple echo program, which transmits all received characters. Both programs transmit 8 bit characters without parity at 38400 Baud. The output for program one can be seen in figure xiii and the output for program two in figure xv.

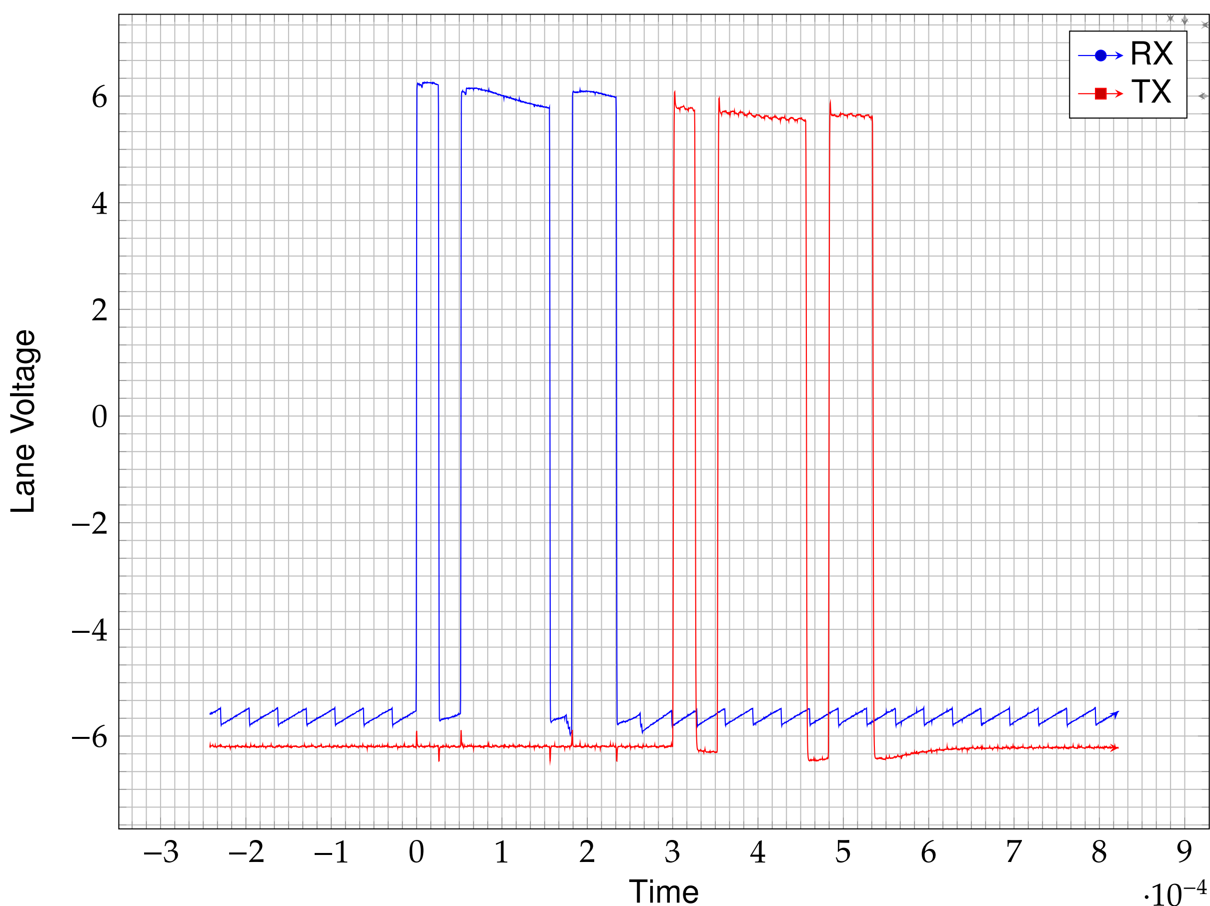

Figure xv: Measurement of a character echo

Transmit code The transmit code regularly transmits the letter capital A via the 16550 UART. Before it can do this it needs to perform some initialisations. The functions shown in listing I are the read and write routines for accessing the 16550 UART. These routines also apply to the echo code.

```
#define F_CPU 16000000UL

#include <stdint.h>
#include <util/delay.h>

#define BUS_HOLD_US 1
```

```

/* Shift values inside the PORTL Register */
#define WR_SHIFT 1
#define RD_SHIFT 2
#define MR_SHIFT 0
#define CS_SHIFT 3
#define CS_ADC_SHIFT 4

/* Registers in the 16550 UART */

#define UART_REG_DLLS   0
#define UART_REG_DLMS   1
#define UART_REG_TXRX   0
#define UART_REG_IER    1
#define UART_REG_IIR    2
#define UART_REG_LCR    3
#define UART_REG_MCR    4
#define UART_REG_LSR    5
#define UART_REG_MSR    6
#define UART_REG_SCR    7

void set_addr(uint8_t addr){

    PORTK = addr;
    return;
}

void write_to_16550(uint8_t addr, uint8_t data){


    set_addr(addr);
    DDRF = 0xFF;
    PORTL &= ~(1<<WR_SHIFT);
    PORTF = data;
    PORTL &= ~(1<<CS_SHIFT);

    _delay_us(BUS_HOLD_US);

    PORTL |= 1<<CS_SHIFT;
    set_addr(0x00);
    PORTL |= 1<<WR_SHIFT;
    PORTF = 0x00;
    return;
}

uint8_t read_from_16550(uint8_t addr){

```

```
    uint8_t data = 0x00;
    set_addr(addr);
    DDRF = 0x00;
    PORTF = 0x00;
    PORTL &= ~(1<<RD_SHIFT);
    PORTL &= ~(1<<CS_SHIFT);
    _delay_us(BUS_HOLD_US);
    data = PINF;
    PORTL |= 1<<CS_SHIFT;
    set_addr(0x00);
    PORTL |= 1<<RD_SHIFT;
    DDRF = 0xFF;
    PORTF = 0x00;
    _delay_us(BUS_HOLD_US); /*Wait for the data and signal lanes to become
   stable*/
    return data;
}
```

Listing I: Read and write routines for the 16550 UART

To write to the 16550 UART, you need to perform some setup tasks. After startup, it requires a MR for at least 5ţs[5]. The baud rate divisor latch needs to be set to the specified divisor for the desired baud rate, and the character width and parity control needs to be set. The MR signal is beeing generated by the AVR on bootup. To access the divisor latch, the divisor latch access bit needs to be set and after setting up the baud rate divisor latch, it nees to be cleared to allow a regular transmission. This process can be seen in listing II

```
int main(){

    /* Disable interrupts during initialisation phase */
    cli();

    /* Setup Data Direction Registers and populate with sane default
       values */
    DDRF = 0xFF; /* Data Bus */
    DDRK = 0xFF; /* Address Bus */
    DDRL = 0xFF; /* Control Bus */
    PORTF = 0x00;
    PORTK = 0x00;
    PORTL = 0x00;

    /* Cleanly reset the 16550 uart */
    PORTL |= (1<<WR_SHIFT);
    PORTL |= (1<<RD_SHIFT);
    PORTL |= (1<<CS_SHIFT);
```

```c
    PORTL |= (1<<MR_SHIFT);
    _delay_us(100);
    PORTL &= ~(1<<MR_SHIFT);
    _delay_us(1000);

    sei();

    for(;;){
        write_to_16550(UART_REG_LCR,0x83);
        write_to_16550(UART_REG_DLLS,0x03);
        write_to_16550(UART_REG_DLMS,0x00);
        write_to_16550(UART_REG_LCR,0x03);
        write_to_16550(UART_REG_TXRX,'A');
        _delay_us(10000);
        }

        return 0;
}
```

Listing II: 16550 INIT routines and single char transmission

The output of this code on the address, data and control bus as well as on the SOUT lane of the 16550 UART can be seen in figure xvi

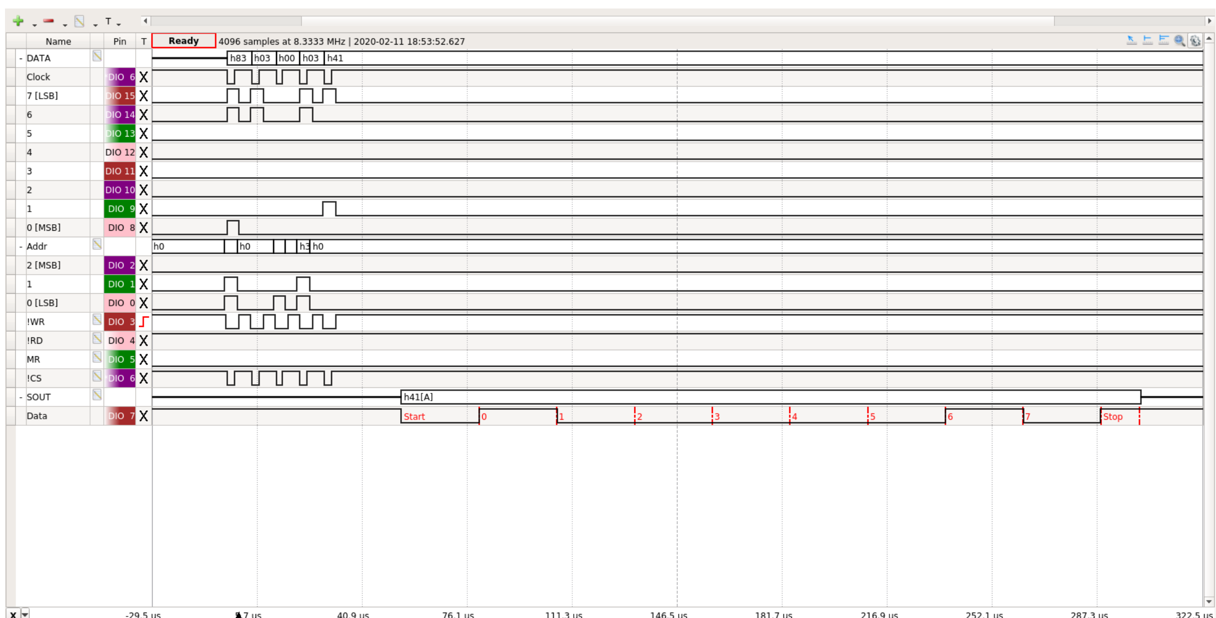

Figure xvi: Transmission of character A via the 16550 UART

Echo code The echo code permanently polls the 16550 UART wether a character has been received, and if yes, reads it from the receiver holding register and writes it back to the tx holding register. The output of this code can be seen in figure xv. The

initialisation is practically the same as for the transmission code, as well as the read and write routines in listing I.

```
int main(){

    /* Disable interrupts during initialisation phase */
    cli();

    /* Setup Data Direction Registers and populate with sane default
       values */
    DDRF = 0xFF; /* Data Bus */
    DDRK = 0xFF; /* Address Bus */
    DDRL = 0xFF; /* Control Bus */

    /* Cleanly reset the 16550 uart */
    PORTL |= (1<<WR_SHIFT);
    PORTL |= (1<<RD_SHIFT);
    PORTL |= (1<<CS_SHIFT);
    PORTL |= (1<<CS_ADC_SHIFT);
    PORTL |= (1<<MR_SHIFT);
    _delay_us(100);
    PORTL &= ~(1<<MR_SHIFT);
    _delay_us(1000);

    write_to_16550(UART_REG_LCR,0x83);
    write_to_16550(UART_REG_DLLS,0x03);
    write_to_16550(UART_REG_DLMS,0x00);
    write_to_16550(UART_REG_LCR,0x03);
    for(;;){
        if(read_from_16550(UART_REG_LSR) & 0x01){
            write_to_16550(UART_REG_TXRX,
                read_from_16550(UART_REG_TXRX));
        }
    }

        return 0;
}
```

Listing III: 16550 character echo

2.8.5 Final Module

The final module can be seen in figure xvii with the pc16550 UART in the center and the MAX-232 above.

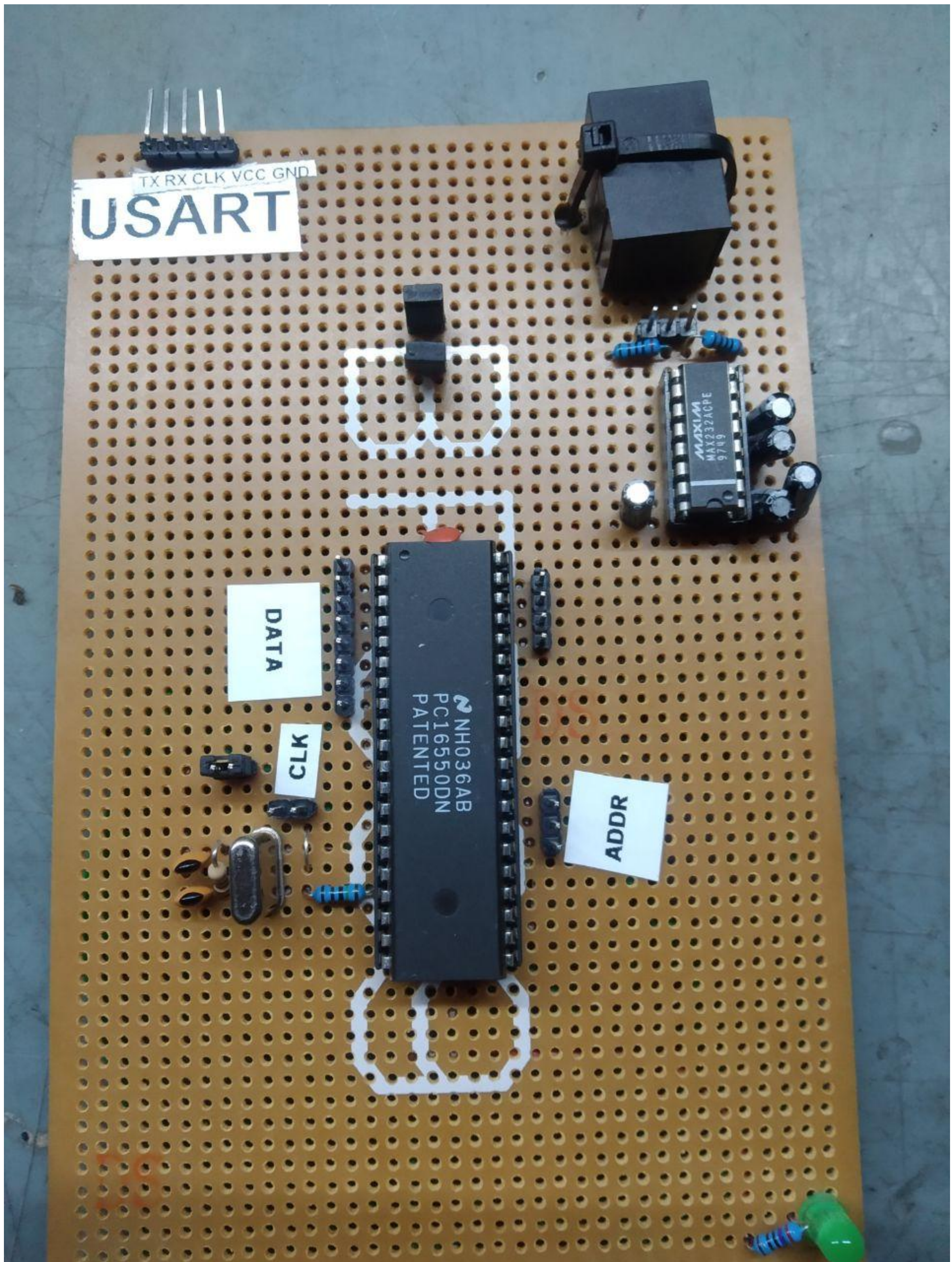

Figure xvii: The final uart module with the pc16550 uart in the center

2.9 Audio Digital-Analog-Converter

A digital to analog converter takes a digital number and converts it to a analog signal. The output of one such conversion is called a sample. With enough samples per second various different waveforms can be produced, which, when amplified and put onto a speaker, can be heared by the human ear as a tone. With various tones in series a melody can be produced, which is what the DAC in this implementation does.

2.9.1 TLC 7528 Dual R2R Ladder DAC

The TLC 7528 is a Dual output parallel input R2R Ladder DAC with a maximum sample rate of 10MHz [7], and which (should be) is monotonic over the entire D/A Conversion Range. The TLC-7528 was the only component chosen, where availability was not a factor, but rather it's design. It is the cheapest dual R2R Ladder dac which takes **PARALLEL** input, which is an important feature, because the backbone of the project is its parallel bus. Further the DAC was developed for audio aplications[7] which made its use obvious and the TLC-7528 was the only IC available as DIP [K], of which the pinout can be seen in figure xviii

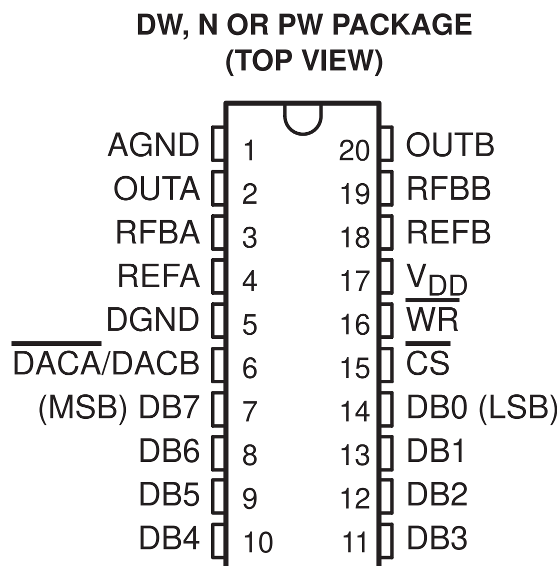

Figure xviii: TLC-7528 Pinout[7]

[K]DIP... Dual Inline Package

2.9.2 IDT7201 CMOS FIFO Buffer

The IDT7201 is an asychronous CMOS FIFO, which means that it can be read with a completely independant speed from which it is written and vice versa. It has 9 bit words, which can be seen in figure xix, and can store up to 256 words[8]. It is used as a buffer to store data describing the targeted waveform in order to free time on the parallel bus for interaction with the 16550 UART.

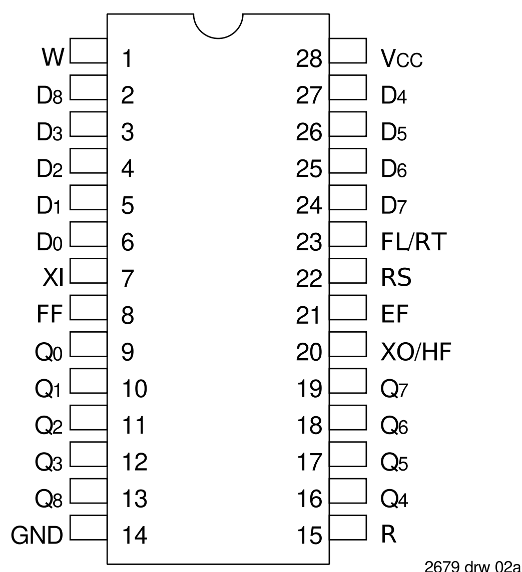

Figure xix: IDT-7201 Pinout[8]

2.9.3 Theory verfication

Before tests of the complete unit were conducted, the functionality of the device and the validity of the knowledge of operations were performed. For that the DAC was directly connected to the ATMega without the FIFO in front of it. A saw was generated on only the DACA channel, which was put into voltage mode as described in the datasheet[7] and seen in figure xx. After the result seen in figure xxi was measured, a lot of effort was put in to determine the source of the heavy noise, however no obvious conclusions can be made, execpt that it comes from the DAC itself and is consistant over whatever frequency used. A damaged IC could be the reason or a sloppy production progress. Filters can be used to reduce the noise, however this was not done in this thesis, as the generated audio does not seem to suffer from these non-linearities as badly as when measured standalone.

voltage-mode operation

It is possible to operate the current multiplying D/A converter of these devices in a voltage mode. In the voltage mode, a fixed voltage is placed on the current output terminal. The analog output voltage is then available at the reference voltage terminal. Figure 11 is an example of a current multiplying D/A that operates in the voltage mode.

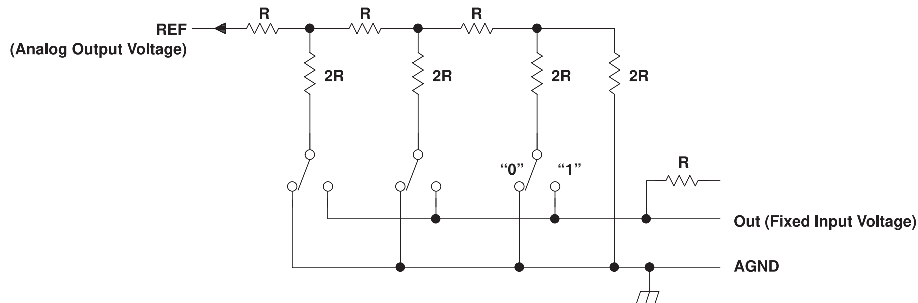

Figure 11. Voltage-Mode Operation

The following equation shows the relationship between the fixed input voltage and the analog output voltage:

$V_O = V_I\ (D/256)$

Figure xx: TLC-7528 in voltage modet[7]

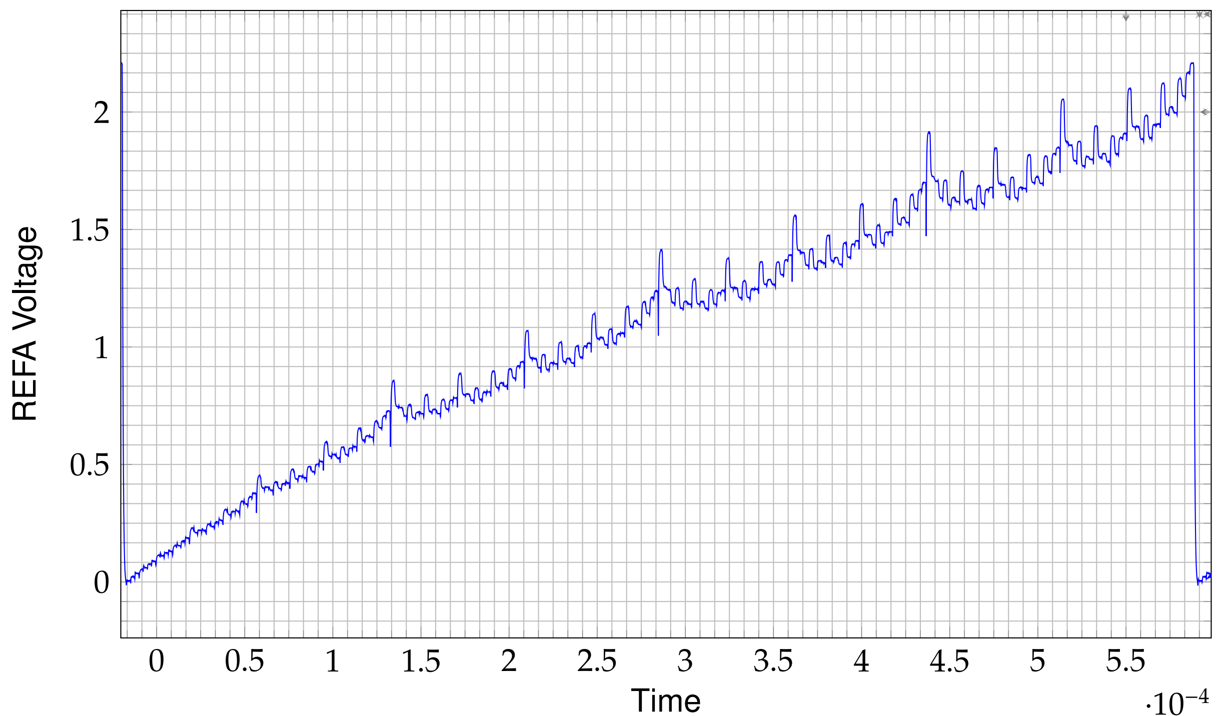

Figure xxi: Measurement of a generated SAW signal via the TLC7528

2.9.4 Schematics

Based on the descriptions in the datasheets the schematic in figure xxii was developed.

Figure xxii: The schematic of the DAC Module

Element Description Diodes D1 through D4 are used as OR-Gates in conjunction with R1 and R2 to generate the $\neg MODRD$ and $\neg MODWR$ signals for the D Flip-Flop [L] and FIFO respectively, by these formulas:

$\neg MODRD = \neg RD \vee \neg MS2$

$\neg MODWR = \neg WR \vee \neg MS2$

On a read access the output enable of the D-Latch becomes low, which writes the status bits of the FIFO onto the data bus. C1, C2 and C3 are for stability reasons and are good practice similar to the UART module. 74HC00 is a quad NAND-Gate[10] which is only used for inversion, chosen, like the 74HC374, for availability reasons. The A part of the NAND-Gate inverts the MR signal from the bus to a $\neg MR$ signal, as the FIFOs reset is low active. The B part of the NAND-Gate inverts the FIFO Empty flag. It's output is connected to the $\neg WR$ input of the DAC, which means that the DAC doesn't convert the input anymore, if the FIFO Empty flag is set to low.

The NE555 generates the audio clock signal, which should be the double of 44.1kHz[M] as 44.1kHz is the standard samling rate of CD-Audio[11]. Resistors R9 and R10 togehter with C7 form the Oscillator part of the NE55. C4 is for stability reasons and doesn't define the frequency of the oscillator.

The generated clock is used for the $\neg RD$ of the FIFO and inverted on the DAC, which makes the data available on the output before being stored into the DAC as it receives the signal to store the data after the FIFO makes it available on the bus.

The DAC is operated in voltage mode as described in xx, with it's voltage source beeing available at either $3.472V_{pp}$ for professional audio or $0.894V_{pp}$ for consumer audio, as defined per convention.[12] The voltage source can be controlled via Jumper JP1.

C5 and C6 together with the load resistance on the audio jack form a high pass with a cutoff frequency of

$f_C = \frac{1}{2\pi RC} = \frac{1}{2\times\pi\times10K\Omega\times100\mu F} = 0.159154943Hz$

which should cover the hearable spectrum. The high pass was needed to generate a positive and negative half of the wave form, as the DC-Offset with a frequency of 0Hz is orders of magnitudes lower than the f_C of the highpass gets filtered away.

R7 and R8 have been installed in order to unload the capacitors after device poweroff.

[L] 74HC374[9]

[M] Because we have 2 output channels

NE55 Clock Source Though used as a clock source, the NE555 is a bad clock source, if a stable frequency is needed, because it varies widely with temperature, preasure and ageing elements. A better solution would have been a quartz, which is divided down to the desired frequency, which was what CD-Drives used to do, but more commonly in modern CD Drives, an ASIC [N] with an internal PLL is used, thus the required quartz can no longer be sourced via conventional electronic resellers.

2.9.5 DAC Module Read

On a read the status bits of the FIFO, which have been latched into the 74HC374 D-Flip-Flop, are written onto the Data bus. Table 1

Table 1: The layout of the Data Bus on read

2.9.6 Demonstration Software

SAW Generator To prove that read and write access from the D Flip-Flop and the FIFO are working, the same saw signal has been generated as in figure xxi , however the signal was put into the FIFO and not the DAC directly. The resulting saw wave can be seen in figure xxiii together with the FIFO Empty flag. The FIFO Empty flag, as explained before, is inverted and starts/ends the complete D/A conversion, until further data is received.

[N] ASIC...Application-specific integrated circuit

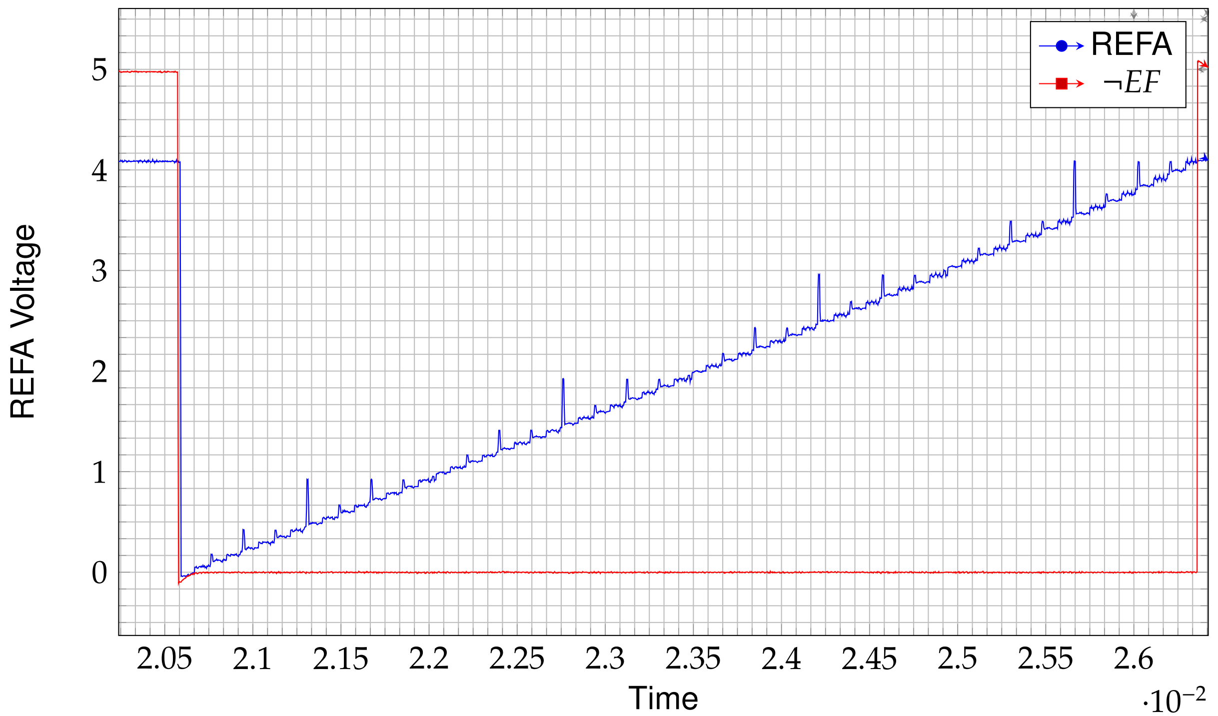

Figure xxiii: Measurement of a generated SAW signal with the FIFO Empty flag

The time difference betwen a store and complete write cycle can be seen in figure xxv, while figure xxiv shows the transmission between dac and fifo in more detail.

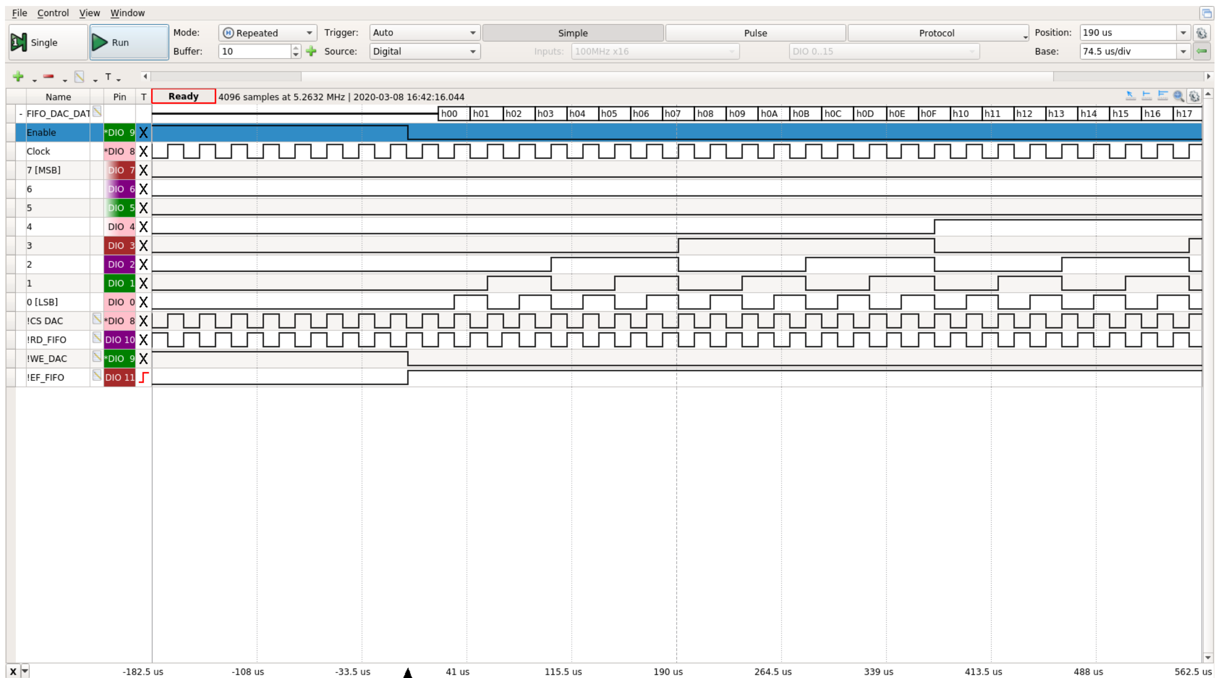

Figure xxiv: A transmission between the FIFO and the DAC

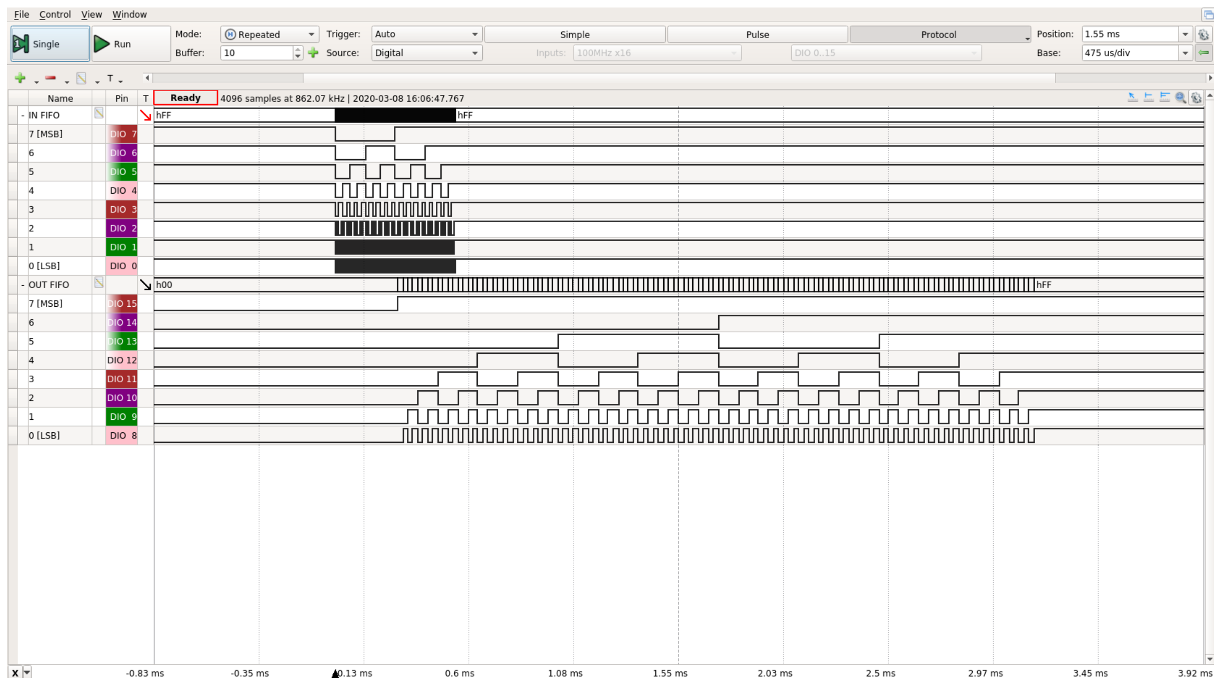

Figure xxv: A fifo store operation in contrast to the load operation

The initialisation routines and read/write operations for the DAC module are basically the same as for the UART module, and have thus been ommitted. They can be seen in listing II.

```
int routine(){

    for(uint8_t i = 0; i  < 0xFF; i++){
        write_to_dac(0x00, i);
    }

    write_to_dac(0x00, 0x00);

    _delay_ms(10);
        return 0;
}
```

Listing IV: SAW Generation for the DAC with FIFO

Sine Generator As a further example a sine was generated and played on the DAC. The ATMega itself is not powerful enough to generate the sine on the fly, therefore a lookup-table had to be generated, which can be seen in listing V. How the data is transmitted to the FIFO can be seen in listing VI and figure xxvi, and the resulting sine on both output channels can be seen in figure xxvii.

```
    /* Generate sine table */
```

```
    uint8_t sine_table[256];
    for(size_t i = 0; i < 256; i++){
        sine_table[i] = 0xFF&((int)((sin(i/((double)255)*(3.141592*2))*
            127.5+127.5)));
    }
```

Listing V: Sine LUT Generation

The look-up table has a size of 256, which is the maximum value an 8 bit integer can take. This size was chosen to make operation faster as it only takes one cycle to load an array value into a register and another one to store it into the GPIO register. The sine table in further examples was pre-genrated on the compiling host to reduce startup time. The mothod shown in listing V is not fast due to the lack of a floating point unit on the AVR. [2]

```
int routine(){

    for(uint8_t i = 0; i  < 0xFF; i++){
        write_to_dac(i%2, sine_table[i]);
    }

    write_to_dac(0x00, 0x00);
    write_to_dac(0x01, 0x00);

    _delay_ms(10);
        return 0;
}
```

Listing VI: DAC Sine Generation

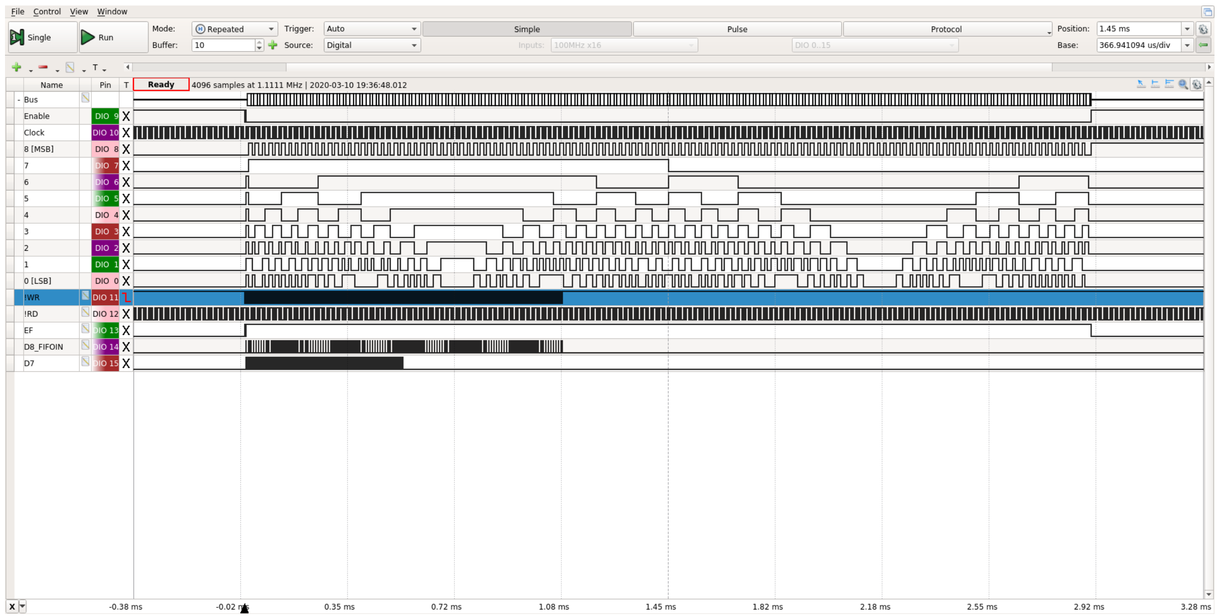

Figure xxvi: Storage and retrieval of a sine to and from the FIFO

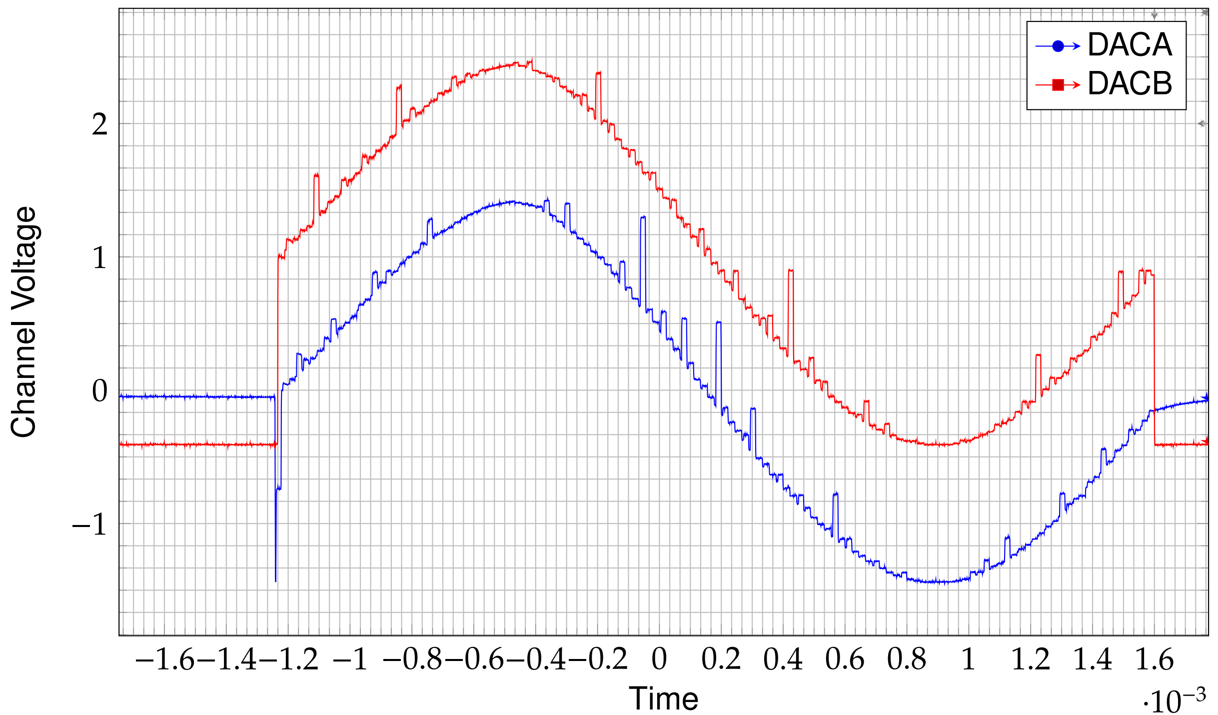

Figure xxvii: Measuremet of the generated sine from the sine LUT on DACA and DACB

3 Addressing DACA and DACB

The DAC used has 2 output channels which can be selected by the $\neg DACA/DACB$ pin as seen in figure xviii. This pin was mapped to bit 0 of the address bus in order to make use of it. Bit 8 on the fifo was used to store the bit. It was not implemented with half the

bus clock to make both channels independent of each other. This however uses more time on the backend because it means the fifo is used up at twice the speed. No current example makes use of this, but it may be used in future examples and implementations on this unit.

On the audio jack DACA is mapped to the right channel and DACB to the left channel.

3.0.1 Final Module

The final module can be seen in figure xxviii with, from bottom to top, the 74HC374 D-Flip-Flop, the IDT-7201 FIFO, the 74HC00 NAND-Gates, the TLC-7528 DAC and the NE555 oszillator. The jumper on the left is the voltage select and the jumper on the right the clock select. The two pin headers on the top have been installed for voltage measurement on the left and right audio channels while the audio jack is in use.

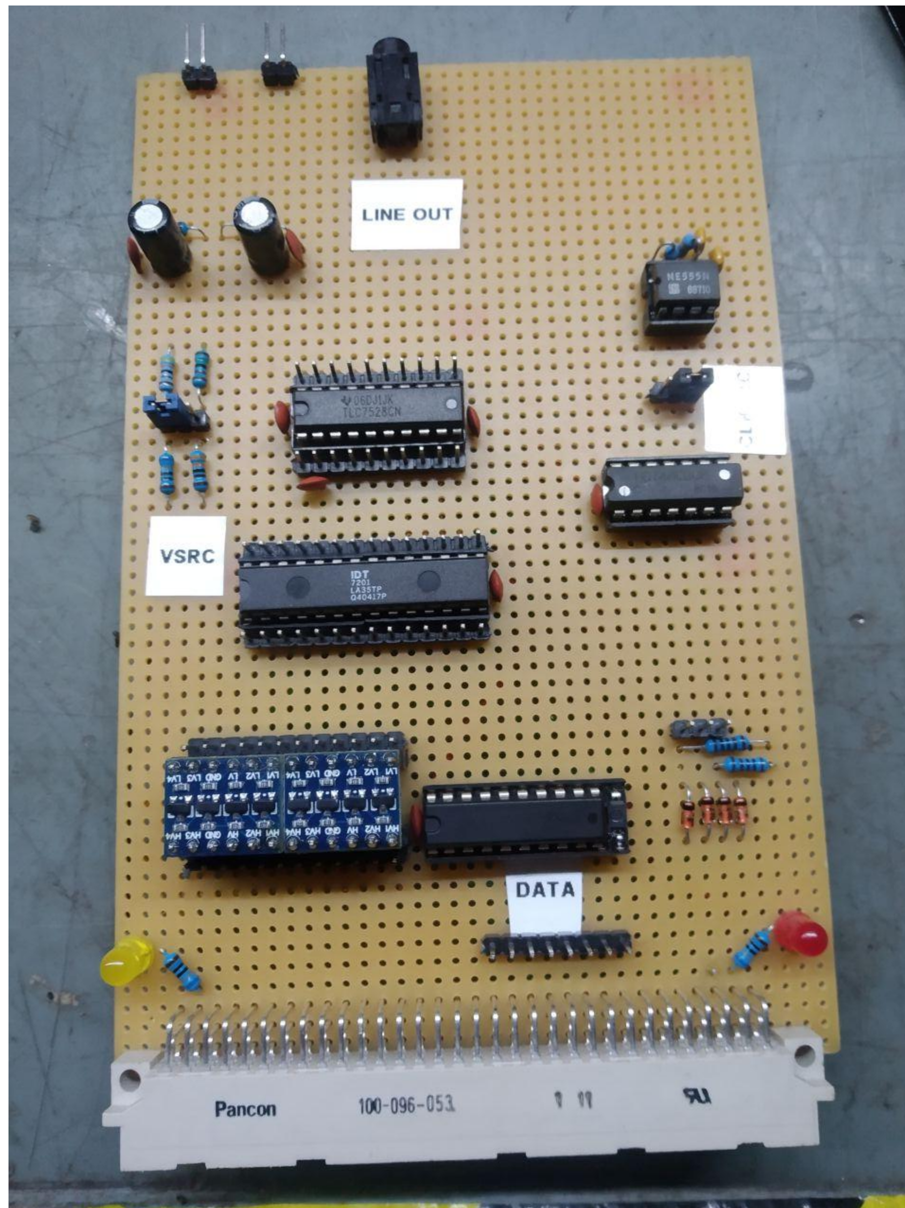

Figure xxviii: The final DAC module

3.1 FPGA to Hardware interface

To make the Hardware work with the FPGA's 3.3V I/O, level shifter have been installed and a FPGA module was built. This module maps the IO/Pins in a similar way to the ATMega 2560 used in examples before. The bidirectional 5V<->3.3V logic level converters have been obtained on amazon, and have not been well documented. Their functionality has been tested and verified in both directions, which can be seen in figures xxix and xxx. The schematic has also been determined through measurements with a multimeter and the schematic in figure xxxi shows similar resistor values in the same configuration [13].

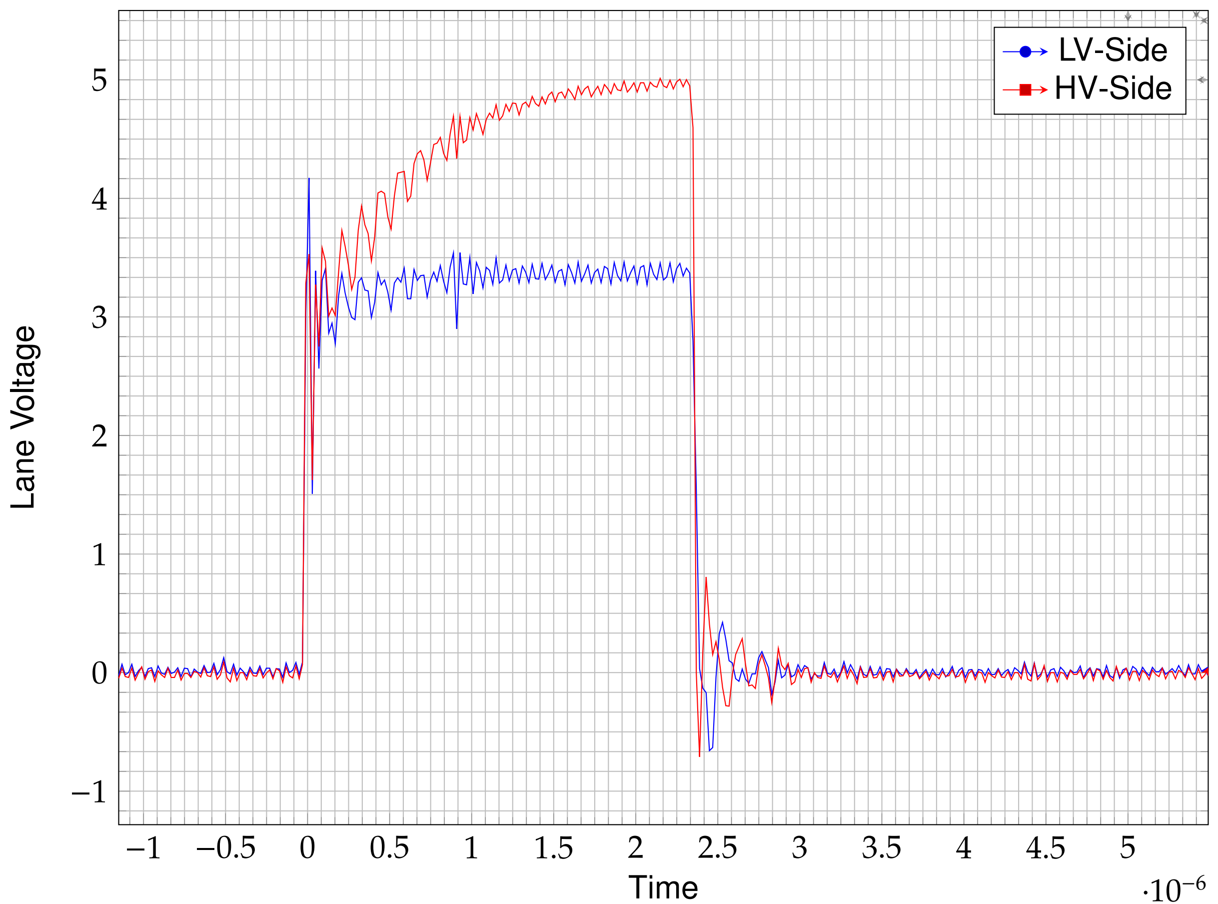

Figure xxix: 3.3V to 5V conversion using the level shifter

The in figure xxix shown output on the HV side, corresponds with the schematics in figure xxxi where it can be seen that the resistor R2 is loading the bus capacitance to a 5V high state.

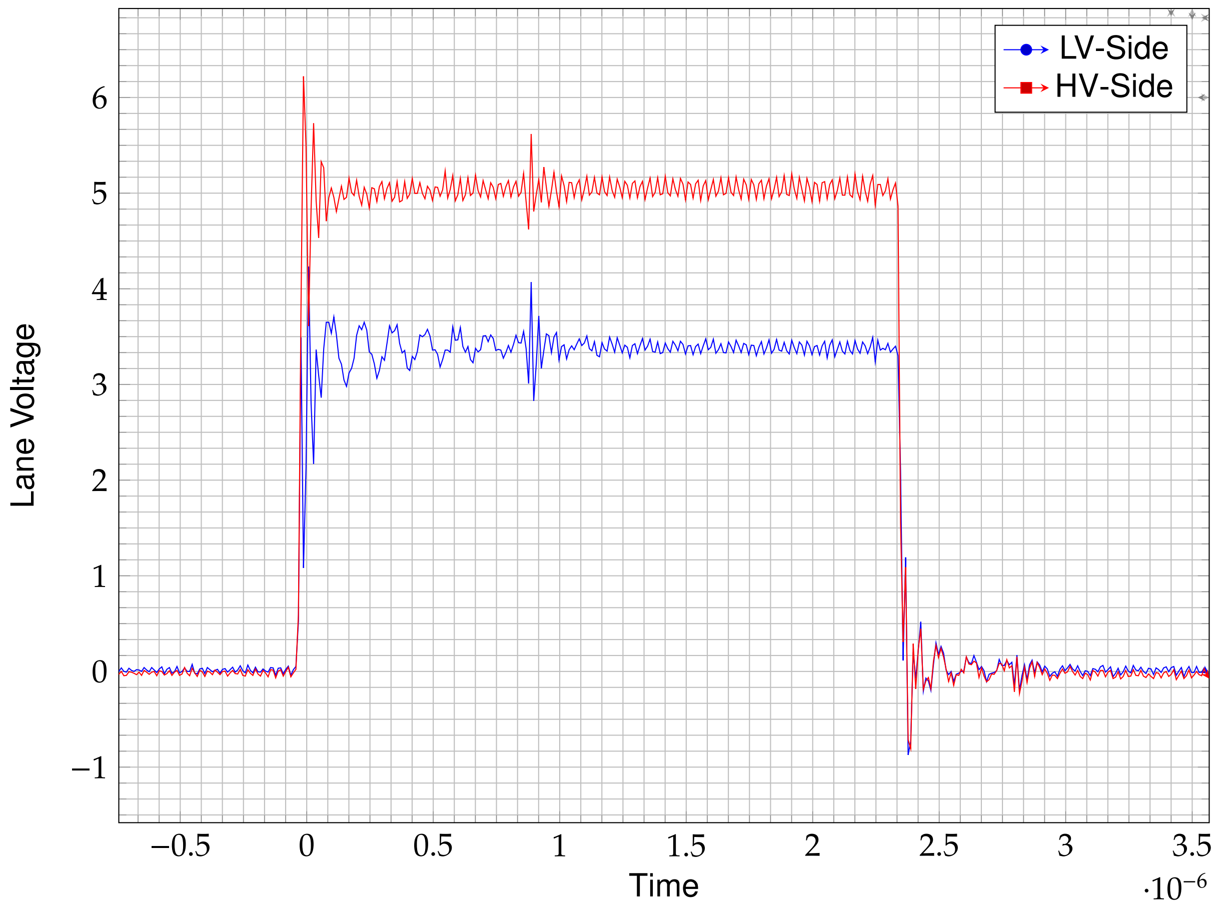

Figure xxx: 5V to 3.3V conversion using the level shifter

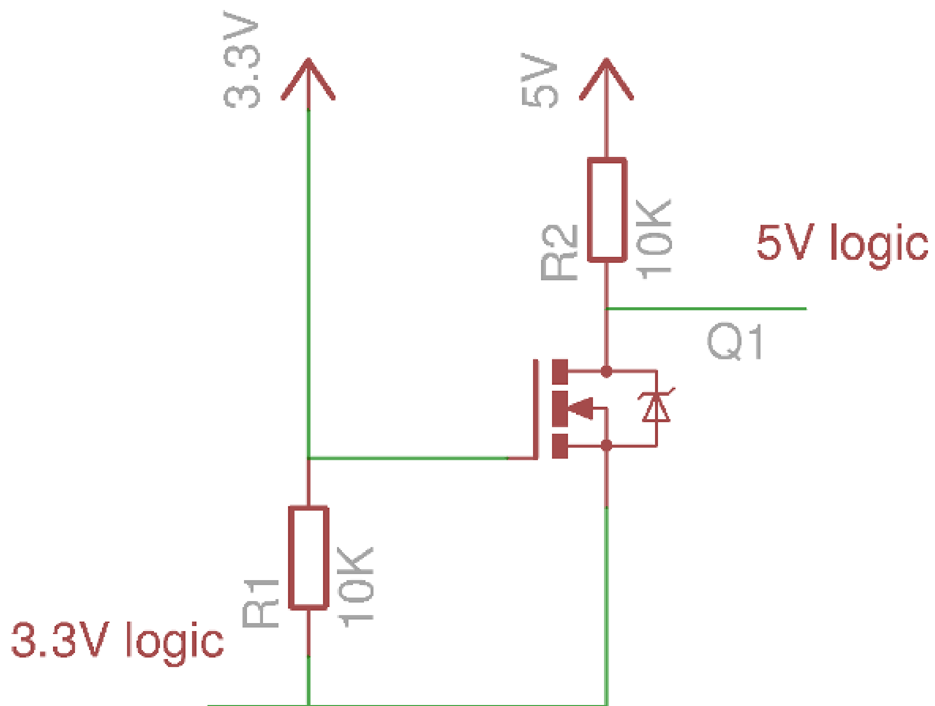

Figure xxxi: The internal schematics of the level shifter[13]

3.1.1 Measurement error

During an attempt to measure wether the level shifters in the final module were working, a measurement between the LV and the HV side showed only a difference of 0.7V. After some troubleshooting, it was found that the Analog Discovery has clamping diodes against the 3.3V rail shown in figure xxxii. These diodes produce the 0.7V offset and prevent the parallel bus from rising to 5V when a digial I/O pin of the Analog Discovery 2 is connected to the bus. [14].

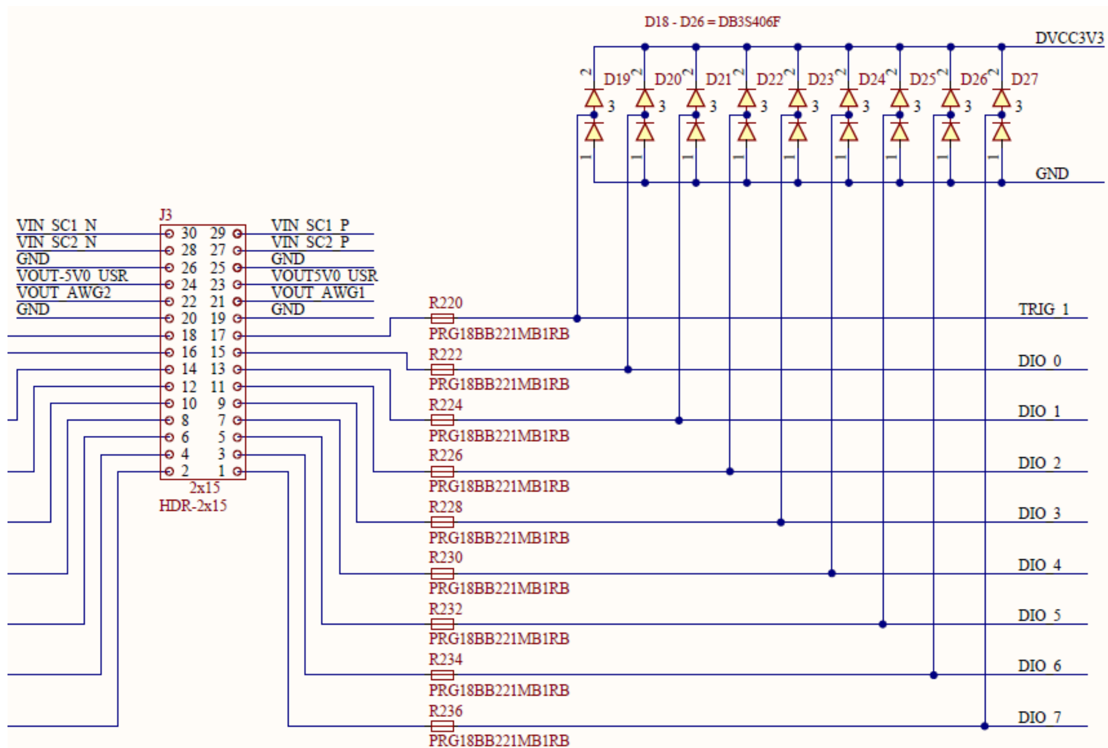

Figure xxxii: The internal clamping diodes of the Analog Discovery 2[3]

3.1.2 Final Module

The final module can be seen in figure xxxiii without the FPGA attached. The blue modules below are the level shifters.

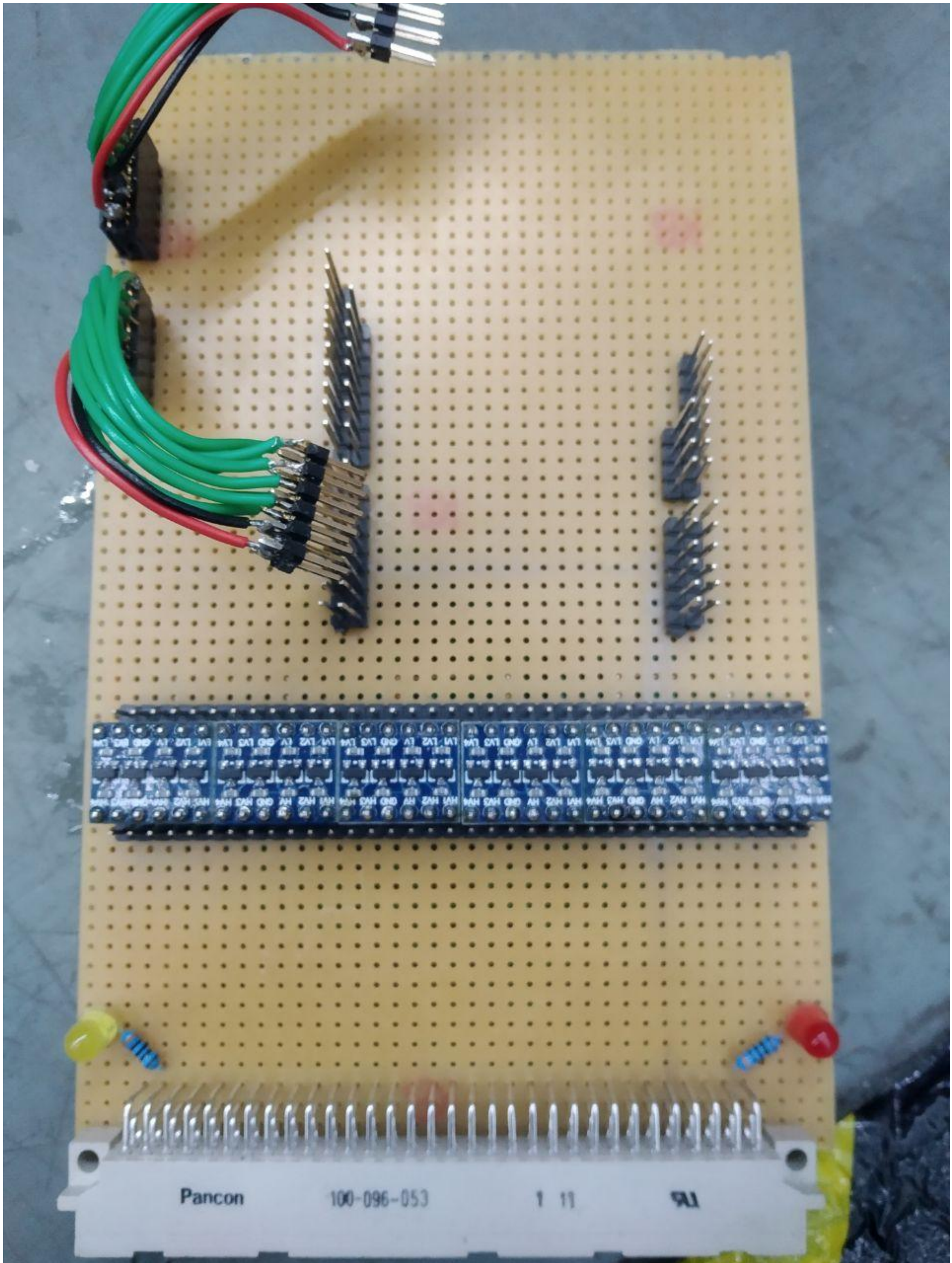

Figure xxxiii: The final FPGA interface module with the level shifters

4 TEXTADVENTURE

To illustrate how the components work together and can be used in various different applications, a small text-adventure with audio effects was written in C. The main goal was to show the capabilities of even small systems like the one developed.

4.1 General Implementation details

4.1.1 General definitions and pinout of the AVR

Like the before examples, the textadventure was implemented on an ATMega2560 and uses 3 different Registers for transmission: PORTF, PORTK and PORTL for address bus, data bus and control bus respectively, as can be seen in listing VII

```
/* Copyright (C) 2020 tyrolyean
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

#ifndef _AVR_H_TEXT
#define _AVR_H_TEXT



#define F_CPU 16000000UL
#include <avr/io.h>

/* Shift values for the peripherials on the control bus PORTL */

#define MR_SHIFT      0
#define WR_SHIFT      1
#define RD_SHIFT      2
#define CS_UART_SHIFT 3
#define CS_DAC_SHIFT  4

#define ADDR_REG      PORTK
#define DATA_REG      PORTF
#define CTRL_REG      PORTL

#define ADDR_DDR_REG  DDRK
#define DATA_DDR_REG  DDRF
#define CTRL_DDR_REG  DDRL

/* Included here to prevent accidental redefinition of F_CPU */
#include <util/delay.h>

/* Time it takes for the bus lanes to become stable for read and write
   access */
#define BUS_HOLD_US 1
```

```
void set_addr(uint8_t addr);

#endif
```

Listing VII: The avr.h header file

The in listing VII shown preprocessor macros MR_SHIFT, WR_SHIFT, RD_SHIFT, CS_UART_SHIFT and CS_DAC_SHIFT are used to indicate the position of the corresponding control lines inside the control bus register. All other shift values are the same bitordering in input as in output.

The BUS_HOLD_US is used to tell the avr how many microseconds it takes for the data bus to be latched into input register of the devices on write or how long it takes for the data bus to become stable on read. A delay of less than 1 microsecond is not possible due to limitations of the AVR and the bus capacity, which increases the BER[O] to a level which effects regular operation.

4.1.2 Read and Write routines

The set_addr function is the same as in the UART example code in listing I and has therefore been omitted, execept for its definiton in the avr.h file in listing VII. The read and write functions for the UART module and the DAC module are the same as in the example code for the modules and have been ommited therefore as well.

4.1.3 UART and DAC update polling

The AVR constantly polls the DAC and UART modules for updates as can be seen in listing VIII. The routine_MODULE functions poll their respective modules for updates as can be seen in listings IX and X. When a character is received, it is stored inside a bufer array and regular operation continues. If the $\neg EF$ status bit is set in a read from the dac, the feed_dac function is called which stores 256 bytes into the DAC and regular operation continues.

```

int routine(){
    routine_dac();
    routine_uart();
    routine_game();
```

[O]BER...Bit Error Ratio

```
        return 0;
```

Listing VIII: The routine function looped by the main

```
void routine_uart(){

    uint8_t received = read_from_uart(UART_REG_LSR);
    if(received & 0x01){
        received = read_from_uart(UART_REG_TXRX);
        ingest_user_char(received);
        if(received == '\r'){
            writechar_16550('\n');
        }
        writechar_16550(received); /* Echo back */
    }

    return;
}
```

Listing IX: The routine function for the UART

```
void routine_dac(){

    uint8_t received = read_from_dac(0x00);
    if(!(received & (0x01<<0))){
        feed_dac();
    }
    return;
}
```

Listing X: The routine function for the DAC

4.1.4 Program execution path

On microprocessors it is required to not leave a return path for programs, as a return path would lead to the microcontroller either resetting, or seicing to work until the next power cut. Therefore the program performs all it's tasks in an infinte loop. This loop can be seen in listing VIII and in figure xxxiv.

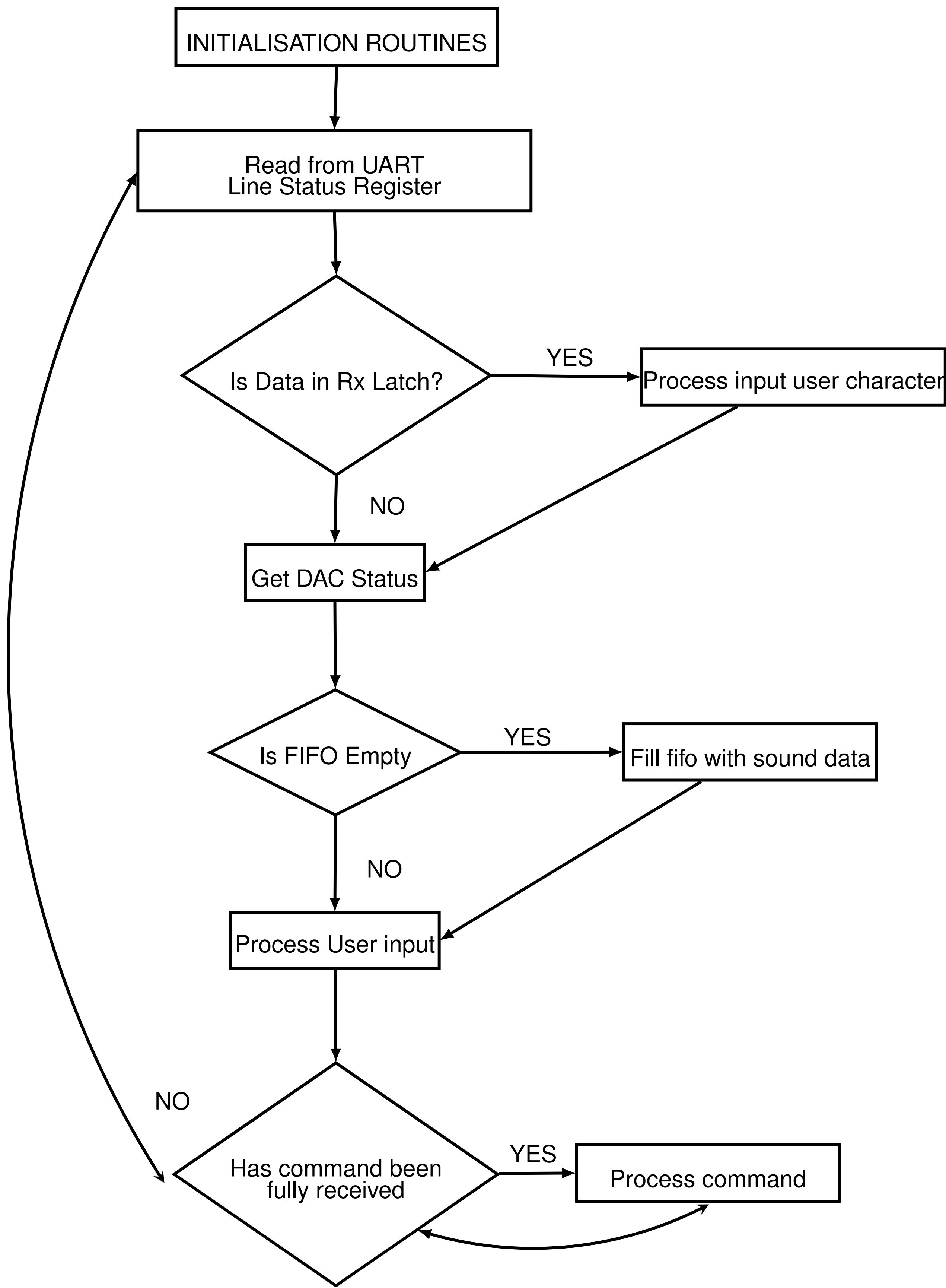

Figure xxxiv: A Flow-Chart of the program execution path

4.2 DAC sound generation

4.2.1 DAC modes

The DAC can produce any waveform described by 8 bit unsigned PCM code. Though possible to feed predefined waveforms into the DAC, the AVR doesn't have enough onboard memory to store more than a few seconds of these waveforms.

For example to store one second of 8 bit unsigned PCM Code at 2 times 44.1KHz sampling rate of the DAC, the AVR would have to store $s = 2 \times 44100 \frac{Bytes}{s} * 1s = 2 \times 44100 Bytes = 88.2KB$, but it has only a total of 256KB of onboard flash[2] which makes for a total track lengh of $t = \frac{256KB}{88.2\frac{KB}{s}} = 2.9s$ with only one track.

Therefore the AVR generates the audio on runtime. To do that it has 6 builtin modes in which it can run, as can be seen in listing XI:

1. silent mode: The DAC produces no output at all and is completely silent.
2. sine mode: The DAC produces a sine with a specific frequency and an amplitude of 255.
3. square mode: The DAC produces a square wave with a specific frequency and an amplitude of 255.
4. saw mode: The DAC produces a saw wave with a specific frequency and an amplitude of 255.
5. noise mode: The DAC produces a pseudo-random white-noise with a maximum amplitude of 255.
6. triangle mode: The DAC produces a triangle wave with a specific frequency and an amplitude of 255.

To perform these tasks the DAC takes two parameters, again seen in listing XI:

- A frequency deviation: Used to tell the dac how much the desired frequency deviates from the base frequency of each waveform.
- A mode: Used to tell it which waveform to generate

```c
/* The operation modes of the dac used for generation of different tones */
#define DAC_MODE_SILENT      0
#define DAC_MODE_SINE        1
#define DAC_MODE_SQUARE      2
#define DAC_MODE_SAW         3
#define DAC_MODE_NOISE       4
#define DAC_MODE_TRIANGLE    5

extern uint8_t dac_mode;
/* This variable is used to deviate the frequency from the baseline
   frequency
 * of around 1kHz. If this integer is positive it makes the produced
   waveform
 * longer, if it is negative the produced waveform becomes less sharp, but
   the
 * frequency goes up. 0 is the baseline */
extern int16_t dac_frequency_deviation;
```

Listing XI: The DAC operation modes

```c
void feed_dac(){
    /* Internal counter for positioning inside the currently playing
     * waveform */
    static uint8_t threash = 0x00;
    /* Used to generate the desired frequency offset if the waveform should
     * be made "longer" --> the frequency made lower from baseline
     */
    static int16_t freq_delay_cnt = 0x00;
    switch(dac_mode){

        default:
        case DAC_MODE_SILENT:
            for(uint8_t i = 0; i  < 0xFF; i++){
                write_to_dac(i%2, 0);
            }

        break;

        case DAC_MODE_SINE:
        /* Generates a sine from a predetermined sine table in program
         * space */
            for(uint8_t i = 0; i  < (0xFF/2); i++){
                write_to_dac(1,
                    pgm_read_byte(&sine_table[threash]));
                write_to_dac(0,
                    pgm_read_byte(&sine_table[threash]));

```

```
28                  if(dac_frequency_deviation >=0){
29                      freq_delay_cnt++;
30                      if(freq_delay_cnt >=
31                          dac_frequency_deviation){
32                          freq_delay_cnt = 0;
33                          threash++;
34
35                      }
36
37                  }else{
38                      threash -= dac_frequency_deviation;
39                  }
40
41              }
42                  break;
43          case DAC_MODE_SQUARE:
44          /* Generates a square wave tone */
45              for(uint8_t i = 0; i  < (0xFF/2); i++){
46                  if(threash > (0xFF/2)){
47                      write_to_dac(0, 0xFF);
48                      write_to_dac(1, 0xFF);
49                  }else{
50                      write_to_dac(0, 0);
51                      write_to_dac(1, 0);
52                  }
53                  if(dac_frequency_deviation >=0){
54                      freq_delay_cnt++;
55                      if(freq_delay_cnt >=
56                          dac_frequency_deviation){
57                          freq_delay_cnt = 0;
58                          threash++;
59
60                      }
61
62                  }else{
63                      threash -= dac_frequency_deviation;
64                  }
65              }
66                  break;
67          case DAC_MODE_SAW:
68              /* Generates a saw wave tone */
69              for(uint8_t i = 0; i  < (0xFF/2); i++){
70                  write_to_dac(0, threash);
71                  write_to_dac(1, threash);
72                  if(dac_frequency_deviation >=0){
73                      freq_delay_cnt++;
74                      if(freq_delay_cnt >=
```

```
75                          dac_frequency_deviation){
76                          freq_delay_cnt = 0;
77                          threash++;
78 
79                      }
80 
81                  }else{
82                      threash -= dac_frequency_deviation;
83                  }
84              }
85          break;
86          case DAC_MODE_NOISE:
87          /* Generates white noise from a predetermined LUT
88           */
89              for(uint8_t i = 0; i  < (0xFF/2); i++){
90                  static uint16_t noise_cnt = 0;
91                  write_to_dac(1,
92                      pgm_read_byte(&noise_table[noise_cnt]));
93                  write_to_dac(0,
94                      pgm_read_byte(&noise_table[noise_cnt]));
95 
96                  noise_cnt++; /* Doesn't have frequency diversion
97                              */
98                  if(noise_cnt >= 1024){
99                      noise_cnt = 0;
100                 }
101 
102             }
103                 break;
104         case DAC_MODE_TRIANGLE:
105             /* Generates a triangle wave tone */
106             for(uint8_t i = 0; i  < (0xFF/2); i++){
107                 static int8_t direction = 1;
108                 if((threash == 0xFF) | !threash){
109                     direction = -direction;
110                 }
111                 write_to_dac(0, threash);
112                 write_to_dac(1, threash);
113                 if(dac_frequency_deviation >=0){
114                     freq_delay_cnt++;
115                     if(freq_delay_cnt >=
116                         dac_frequency_deviation){
117                         freq_delay_cnt = 0;
118 
119                         threash += direction;
120 
121                     }
```

```
122
123                 }else{
124                     if((dac_frequency_deviation *
125                                                         direction) >
126                         (0xFF - threash)){
127                         threash = 0xFF;
128                         continue;
129                     }
130                     threash = (dac_frequency_deviation *
131                         direction);
132                 }
133             }
134         break;
135     }
136 
137     return;
138 }
```

Listing XII: The DAC waveform generation code

4.2.2 Tones and Tracks

A sound track inside the textadventure consists of independent tones. A tone is a waveform at a specific frequency played for a specific time. To perform the specific time functionality independant of DAC speed, an ISR [P] on the AVR was used to change to the next tone every millisecond. A track is an array of tones with an end marker tone at the end which is a tone with a length of 0ms. The end marker tone tells the ISR to reset to the initial tone. The ISR can be seen in listing XIII and the sound update function, which actually updates the current tone and is responsible for playing a track in listing XIV. The output of an example track can be seen in figures xxxv and xxxvi.

```
1 ISR(TIMER0_COMPA_vect)
2 {
3     update_sound();
4 }
```

Listing XIII: The ISR which fires every millisecond

```
1 /* Loops a track indefinitely and changes voices according to predefined
    tables.
2  * A new track resets the internal state. A voice with a length of 0ms is
    used
3  * to mark the end of a track and continue at the beginning
```

[P]ISR...Interrupt Service Routine

```
4  */
5 void update_sound(){
6
7     static uint16_t audio_time = 0;
8     static size_t tone_pointer = 0x00;
9     static struct tone_t current_tone = {DAC_MODE_SILENT, 0,0};
10    if(current_track == NULL){
11        /* ABORT */
12        audio_time = 0x00;
13        return;
14    }
15    audio_time++;
16    static const struct tone_t * old_track = NULL;
17
18    if(audio_time >= current_tone.length ||
19        current_track != old_track){
20
21        if(old_track != current_track){
22            tone_pointer = 0;
23            audio_time = 0x00;
24            old_track = current_track;
25        }
26        memcpy_P(&current_tone,&(current_track[tone_pointer]),
27            sizeof(current_tone));
28
29        if(current_tone.length == 0){
30            tone_pointer = 0;
31            memcpy_P(&current_tone,&(current_track[tone_pointer]),
32                sizeof(current_tone));
33
34        }
35
36        dac_mode = current_tone.waveform;
37        dac_frequency_deviation = current_tone.frequency_deviation +
38            global_frequency_offset;
39        audio_time = 0x00;
40        tone_pointer++;
41    }
42    return;
43 }
```

Listing XIV: The sound update function

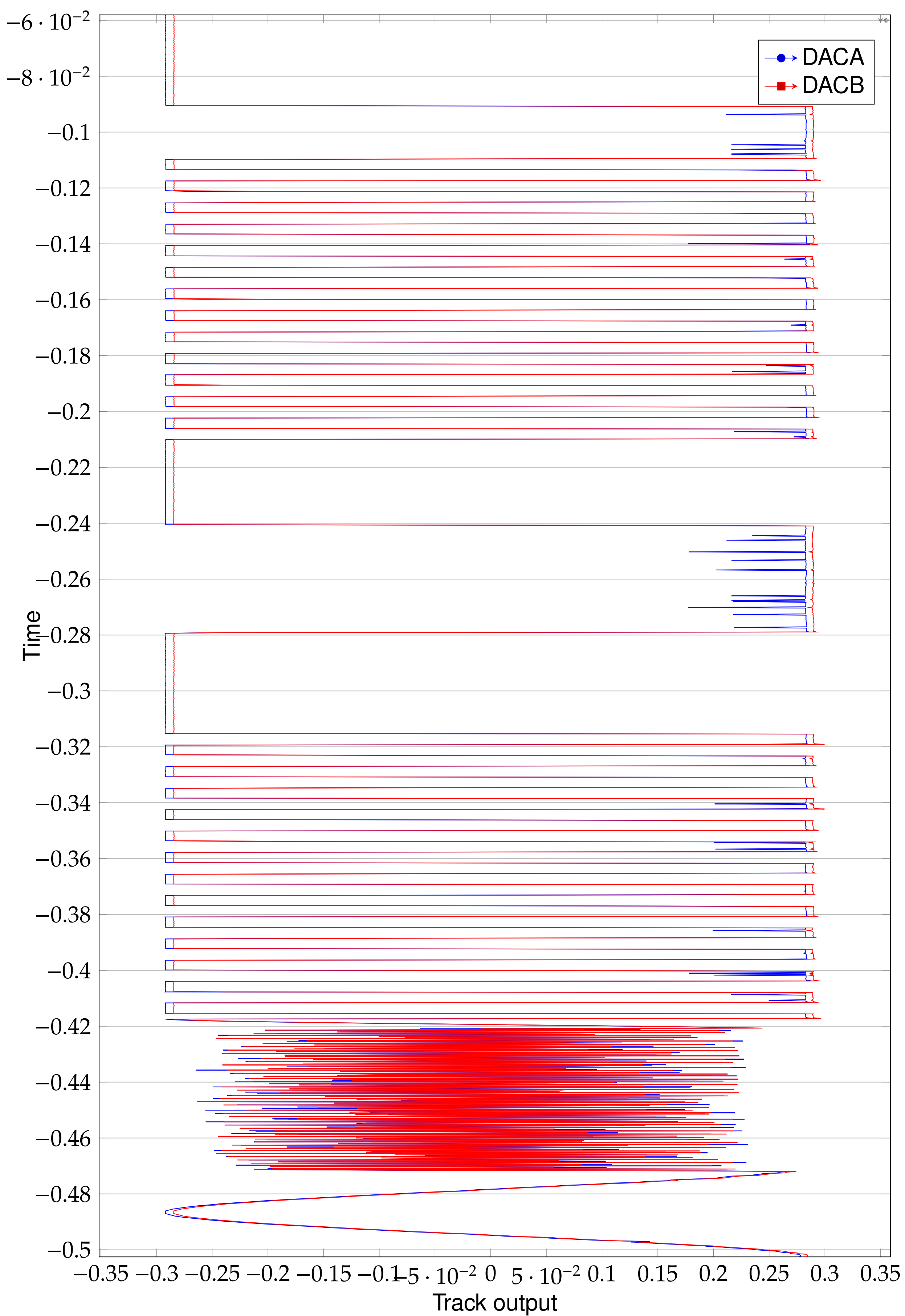

Figure xxxv: The output of an example track part 1

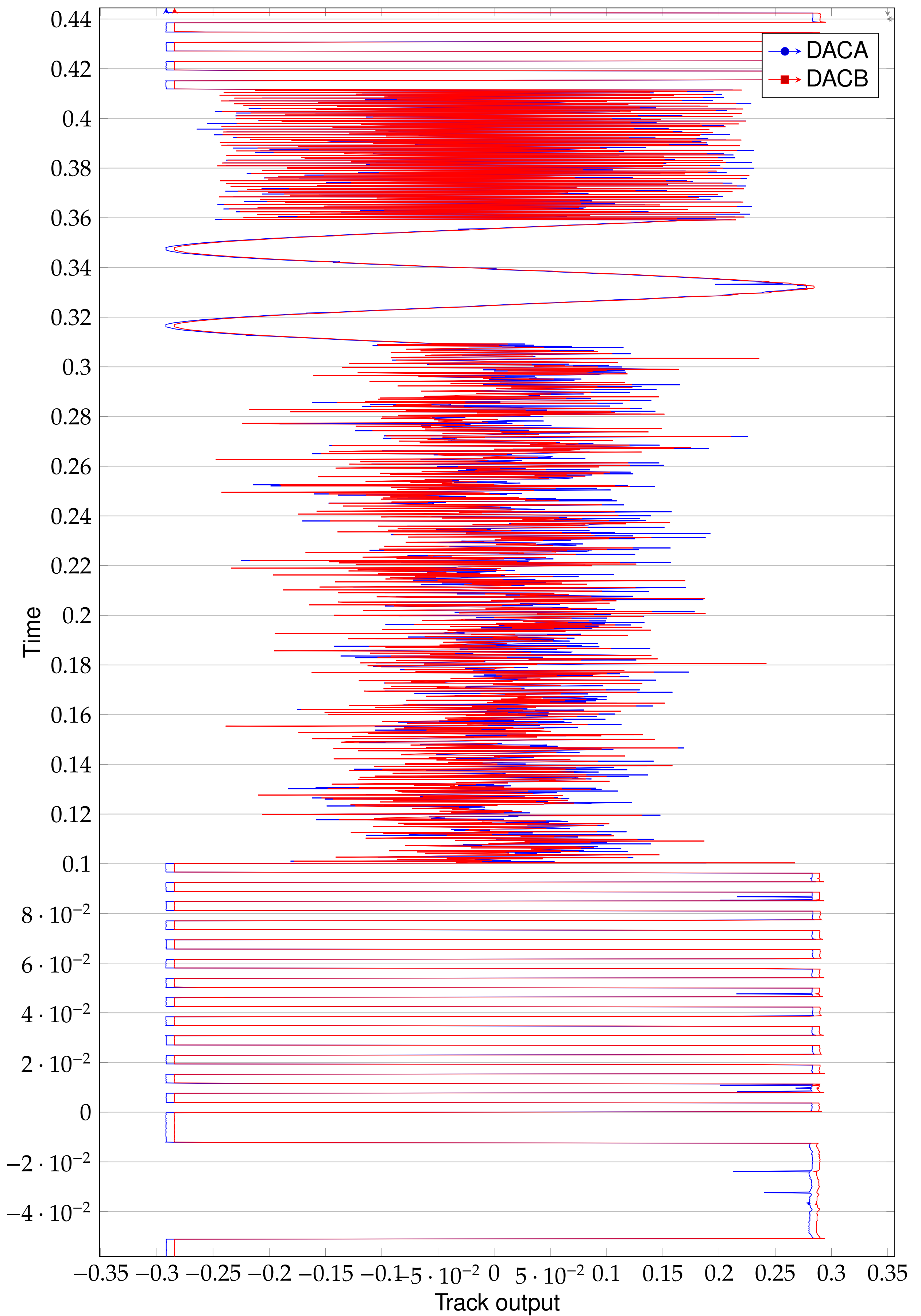

Figure xxxvi: The output of an example track part 2

4.2.3 Track switching

To switch tracks on different actions, there is a map of tracks associated with rooms. Every room has an associated track, where the association can change on actions performed, which allows for a game atmosphere change. Track changes are performed outside the ISR, which could theoretically result in a race condition where the ISR would load a faulty track for 1ms if the track change was not performed fast enough, but this is prevented by disabling global interrupts during a track change.

4.3 User command interpretation

4.3.1 Command structure and parsing

As in other text adventures [15] a command consits of one line of input terminated by a newline or line feed character \n. The carriage return character which is sometimes transmitted with a line feed character is not parsed in this text adventure. Incoming character parsing can be seen in listings IX and XV.

As one command is parsed each part is required to be separated by an empty space character which is ascii code 32 [16]. The first part of the given input is then compared to an array of actions a user can perform, for example use or search, as can be seen in listing XVI

In listing IX the comment echo back can be seen. The write_char function before it writes the last received character back to the terminal which sent it. This is done to write what the user typed out to the terminal as otherwise it would not be seen what has been typed on any VT100 compatiable terminal[17] or terminal emulator.

```
void ingest_user_char(char in){
    if(in == 0x7F /* DELETE CHAR */){
        command_buffer[command_buffer_pointer--] = 0x00;

    }else{
        command_buffer[command_buffer_pointer++] = in;
    }
    return;
}
```

Listing XV: The character ingest function

The in listing XV shown branch overrides the last received character with 0x00 which

is ascii NUL and decrements the buffer pointer by one if the received character was 0x7F. 0x7F is the ADCII DELETE character [16] which instructs the receiving end that the last received character was a mistake and should be purged. This is also what a vt100 compiant terminal emulator sends when the backspace or delete key is pressed [17].

```
void routine_game(){

    if(command_buffer_pointer >= sizeof(command_buffer)){

        command_buffer_pointer = 0x00;
        memset(command_buffer, 0, sizeof(command_buffer));

        println("\nToo much input!");
        return;
    }

    if(command_buffer[command_buffer_pointer-1] == '\n' ||
            command_buffer[command_buffer_pointer-1] == '\r'){
        /* A command from the user has been received, we are ready to
         * do something!*/

        int8_t action_id = -1;
        for(size_t i = 0; i < sizeof(action_table)/sizeof(const char*);
            i++){
            if(strncasecmp(action_table[i], command_buffer,
                strlen(action_table[i])) == 0){
                action_id = i;
                break;
            }

        }
        if(action_id < 0){
            println(info_table[1]);
        }else{
            perform_action(action_id);

        }

        command_buffer_pointer = 0x00;
        memset(command_buffer, 0, sizeof(command_buffer));
    }

    return;
}
```

Listing XVI: The command parsing function

4.3.2 Command parameters

Command paramters are interpreted as the string that follows the action and the space behind it. As can be seen in the case for ACTION_USE in listing XVII the use item function is passed the command buffer[Q] plus the length of the entered command plus one for the space. So the string starting at the passed address should match the start address of the parameter. If no parameter is supplied, the address should point to a character containing ASCII NUL, which marks the end of a string, bcause after comand parsing the string is overwritten with zeros as seen in listing XVI.

```
void perform_action(uint8_t action_id){
    putchar_16550('\n', NULL);
    switch(action_id){
        default:
        case ACTION_HELP:
            println("You can:");
            for(size_t i = 0; i < NUM_ACTIONS; i++){
                println("    %s",action_table[i]);
            }
            break;

        case ACTION_DESCRIBE:
            describe_room(current_room, false);
            break;

        case ACTION_NORTH:
        case ACTION_SOUTH:
        case ACTION_WEST:
        case ACTION_EAST:
            move_direction(action_id -1);
            break;
        case ACTION_INVENTORY:
            print_inventory();
            break;
        case ACTION_SEARCH:
            print_room_item();
            break;
        case ACTION_TAKE:
            consume_room_item(command_buffer+
                strlen(action_table[ACTION_TAKE])+1);
            break;
        case ACTION_USE:
            use_item(command_buffer+
                strlen(action_table[ACTION_USE])+1);
```

[Q] which is an address in memory

```
35                break;
36
37        };
38        println(info_table[3]);
39
40        return;
41    }
```

Listing XVII: The command execution routine

4.4 Gameplay

The game itself plays like a regular game with limtations set in direction. Playeras can search for items in each room and grab the found items as can be seen in figure xxxvii. The general gamplay is perfomred via altering the map data and the strings output to the user.

```
INIT
LONELY ROAD

You are on the dead end of a lonely road. You look right and left ofyou, but
you cannot remember why you are here... You are terrified.
▒INIT
LONELY ROAD

You are on the dead end of a lonely road. You look right and left ofyou, but
you cannot remember why you are here... You are terrified.
help

You can:
    help
    north
    south
    west
    east
    describe
    use
    inventory
    search
    take
What are you going to do?
dearch
Invalid command!
search

You found a PISTOL
What are you going to do?
take pistol

You took the PISTOL
What are you going to do?
north

Moving towards north
S/N DIRT ROAD

You travel a bit towards the moon, you think that's the way to go. You find a
bear in the middle of the road sleeping seemingly in peace.
What are you going to do?
use pistol

You can't use that!
What are you going to do?
use sausage

You can't use that!
What are you going to do?
search sauasage

You found a SAUSAGE
What are you going to do?
take sausage

You took the SAUSAGE
What are you going to do?
use sausage

it ran away...
What are you going to do?
```

Figure xxxvii: A regular beginning of the game

4.4.1 Memory constraints

The AVR has 8kB of internal SRAM which are used for stack and heap [2]. During the build of the program an ELF file can be obtained which contains infromation on the programs structure and memory usage on boot. Strings and variables are contained within the .data section of the elf file, and loaded into memory during boot[18]. This is done for integer variables, as well as for strings, which makes the use of strings limited not to the flash size but to the RAM size of the AVR. To save memory, sound tracks as well as the sine and noise table have been put into program space with the PROGMEM attribute as described by the avr-libc documentation[19]. In listing XII a read from program memory can be seen in the noise and sine modes. Strings have not been put into programmspace as this would require each string to be declared independantly and then be put into arrays[19] as is done now, which would make the code much less readable and increase overhead As well as make the usage of buffers nescessary in order for the override of the printf function to work.

4.4.2 Story

The basics of the storyline are that you wake up in the middle of a forest and don't remember anything. You have to get through the forest to an old house, while having to get rid of a bear which is blocking the way. Inside the house you have to get a computer to start. The game then proceeds to get recursive and your goal is to break out of the recursion.

4.4.3 Recursion

The game, when performing the recursion, resets your inventory and internal state machines, before putting you back to the starting point. However by altering the orientation of rooms, altering the list of items found inside rooms and by altering the texts output by the game, the atmosphere can be changed, and the outcome changed.

4.4.4 Computer State Machine

One example of a state machine inside the game is the computer inside the old-house. The computer needs three items: a keyboard to type on, something to boot from, for example a floppy disk, and a screwdriver to start it. The state machine implementation can be seen in listing XVIII and the state diagram in figure xxxviii.

```
bool perform_computer_action(uint8_t item_id){

    static bool fleshed = false;
    if(item_id == ITEM_KEYBOARD &&
        computer_state == COMPUTER_STATE_NOTHING){
        computer_state = COMPUTER_STATE_KEYBOARD;
        inventory[item_id] = false;
        println("You connected the keyboard");
        return true;
    }

    if(item_id == ITEM_FLOPPY &&
        computer_state == COMPUTER_STATE_KEYBOARD){
        computer_state = COMPUTER_STATE_FLOPPY;
        inventory[item_id] = false;
        println("You inseted the floppy disk");
        return true;
    }
    if(item_id == ITEM_FLESH &&
        computer_state == COMPUTER_STATE_KEYBOARD){
        computer_state = COMPUTER_STATE_FLOPPY;
        inventory[item_id] = false;
        println("You inserted the flesh into the floppy drive");
        fleshed = true;
        return true;
    }
    if(item_id == ITEM_SCREWDRIVER &&
        computer_state == COMPUTER_STATE_FLOPPY){
        computer_state = COMPUTER_STATE_FLOPPY;
        inventory[item_id] = false;
        /* Perform a reset of the game */
        println("You start the computer with the screwdriver, sit down"
            " and watch it boot into a textadventure:");

        reset_game(fleshed);
        return true;
    }

    return false;
}
```

Listing XVIII: The computer FSM

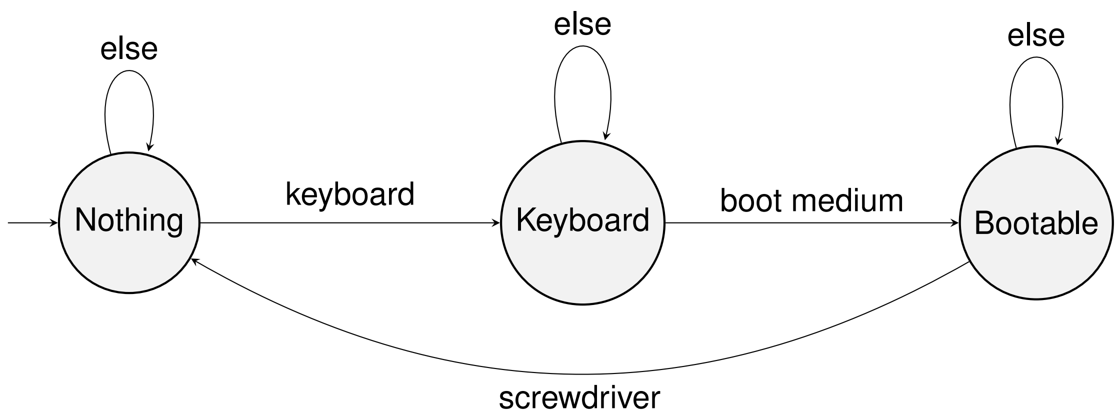

Figure xxxviii: A state diagram of the computer state machine

Part I

A short introduction to VHDL

Designing a processor is a big task, and it's easiest to start very small. With software projects, this is usually in the form of a "Hello World" program - we will be designing a hardware equivalent of this.

5 Prerequisites

Other than a text editor, the following Free Software packages have to be installed:

`ghdl` **[20]** to analyze, compile, and simulate the design

`gtkwave` **[21]** to view the simulation waveform files

`yosys` **[22]** to synthesize the design

`ghdlsynth-beta` **[23]** to synthesize the design

`nextpnr-xilinx` **[24]** to place and route the design

`Project X-Ray` **[25]** for FPGA layout data and bitstream tools

`openFPGALoader` **[26]** to load the bitstream onto the FPGA

6 Creating a design

A simple starting design is an up/down counter. The following VHDL code describes the device:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter is
    port (
        clk       : in std_logic;
        reset     : in std_logic;
        enable    : in std_logic;
        direction : in std_logic;

        count_out : out std_logic_vector(7 downto 0)
```

```vhdl
13     );
14 end counter;
15
16 architecture behaviour of counter is
17     signal count : unsigned(7 downto 0) := (others => '0');
18 begin
19     proc: process(clk)
20     begin
21         if reset then
22             count <= (others => '0');
23         elsif rising_edge(clk) and enable = '1' then
24             if direction = '1' then
25                 count <= count + 1;
26             else
27                 count <= count - 1;
28             end if;
29         end if;
30     end process;
31
32     count_out <= std_logic_vector(count);
33 end behaviour;
```

counter.vhd

In order to test this design, a test bench has to be created:

```vhdl
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity counter_tb is
6  end counter_tb;
7
8  architecture test of counter_tb is
9      signal clk, reset, enable, direction : std_logic;
10     signal s_count_out : std_logic_vector(7 downto 0);
11
12     signal count_out : unsigned(7 downto 0);
13 begin
14     uut: entity work.counter
15         port map (
16             clk       => clk,
17             reset     => reset,
18             enable    => enable,
19             direction => direction,
20
21             count_out => s_count_out
22         );
23
24     count_out <= unsigned(s_count_out);
25
26     simulate: process
27         begin
28             clk <= '0';
29             reset <= '1';
30             enable <= '0';
31
32             wait for 30 ns;
33             assert count_out = 0;
34
```

```
35              reset <= '0';
36
37              clk <= '0';
38              wait for 10 ns;
39              clk <= '1';
40              wait for 10 ns;
41
42              assert count_out = 0;
43
44              enable <= '1';
45              direction <= '0';
46
47              clk <= '0';
48              wait for 10 ns;
49              clk <= '1';
50              wait for 10 ns;
51
52              assert count_out = 255;
53
54              direction <= '1';
55
56              clk <= '0';
57              wait for 10 ns;
58              clk <= '1';
59              wait for 10 ns;
60
61              clk <= '0';
62              wait for 10 ns;
63              clk <= '1';
64              wait for 10 ns;
65
66              assert count_out = 1;
67
68              wait for 30 ns;
69              wait;
70          end process;
71  end test;
```

counter_tb.vhd

7 Simulating a design

```
# analyze the design files
ghdl -a --std=08 *.vhd
# elaborate the test bench entity
ghdl -e --std=08 counter_tb
# run the test bench, saving the signal trace to a GHW file
ghdl -r --std=08 counter_tb --wave=counter_tb.ghw
# open the trace with gtkwave (using the view configuration in
   counter_tb.gtkw)
```

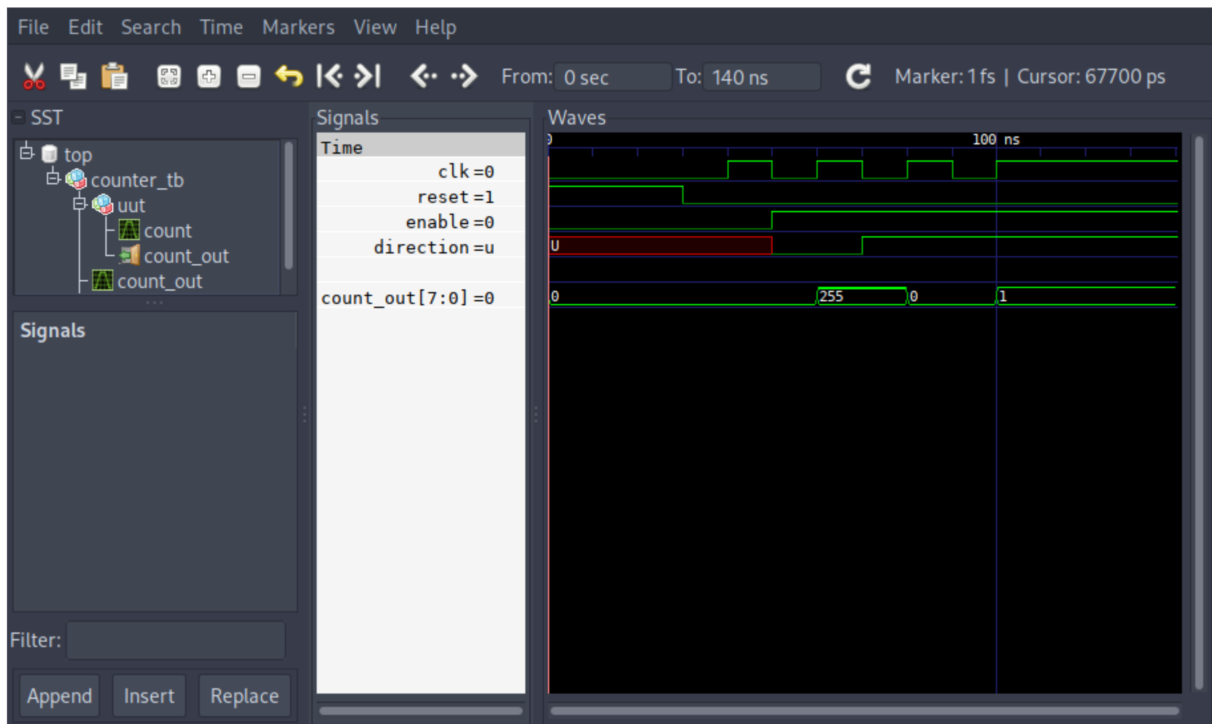

Figure xxxix: Screenshot of the resulting waveform in GTKWave

```
gtkwave counter_tb.ghw counter_tb.gtkw
```

Listing XXI: Commands required to simulate the counter design

8 Synthesizing a design

An additional Xilinx Design Constraints (XDC) file is required to assign the signals to pins on the FPGA:

```
1  set_property LOC D9  [get_ports clk]
2  set_property LOC C9  [get_ports reset]
3  set_property LOC A8  [get_ports enable]
4  set_property LOC C11 [get_ports direction]
5
6  set_property LOC F6  [get_ports count_out[0]]
7  set_property LOC J4  [get_ports count_out[1]]
8  set_property LOC J2  [get_ports count_out[2]]
9  set_property LOC H6  [get_ports count_out[3]]
10 set_property LOC H5  [get_ports count_out[4]]
11 set_property LOC J5  [get_ports count_out[5]]
12 set_property LOC T9  [get_ports count_out[6]]
13 set_property LOC T10 [get_ports count_out[7]]
14
15 set_property IOSTANDARD LVCMOS33 [get_ports clk]
16 set_property IOSTANDARD LVCMOS33 [get_ports reset]
17 set_property IOSTANDARD LVCMOS33 [get_ports enable]
18 set_property IOSTANDARD LVCMOS33 [get_ports direction]
19 set_property IOSTANDARD LVCMOS33 [get_ports count_out[0]]
20 set_property IOSTANDARD LVCMOS33 [get_ports count_out[1]]
21 set_property IOSTANDARD LVCMOS33 [get_ports count_out[2]]
22 set_property IOSTANDARD LVCMOS33 [get_ports count_out[3]]
```

```
23 set_property IOSTANDARD LVCMOS33 [get_ports count_out[4]]
24 set_property IOSTANDARD LVCMOS33 [get_ports count_out[5]]
25 set_property IOSTANDARD LVCMOS33 [get_ports count_out[6]]
26 set_property IOSTANDARD LVCMOS33 [get_ports count_out[7]]
```

counter.xdc

```
# synthesize with yosys
yosys -m ghdl.so -p '
    ghdl --std=08 counter.vhd -e counter;
    synth_xilinx -flatten;
    write_json counter.json'
# place and route the design with nextpnr
nextpnr-xilinx --chipdb xc7a35tcsg324-1.bin --xdc counter.xdc
   --json counter.json --fasm counter.fasm
# convert the FPGA assembly to frames
fasm2frames.py --part xc7a35tcsg324-1 counter.fasm counter.
   frames
# convert the frames to a bitstream
xc7frames2bit --part-name xc7a35tcsg324-1 --frm-file counter.
   frames --output-file counter.bit
# upload the bitstream to the FPGA
openFPGALoader -b arty counter.bit
```

Listing XXIII: Commands required to synthesize the counter design

The current value of the counter is displayed in binary on the eight LEDs on the board. When switch 0 (enable) is in the high position, the counter can be advanced using button 0, with the direction set by switch 1. Button 1 resets the counter to zero.

Part II

Meta

9 History

The project started out with the desire to build a CPU from scratch. Examples such as The NAND Game[27] and Ben Eater's Breadboard Computer series[28] served as

inspirations and guidance during development.

At first, a design similar to Ben Eater's consisting solely of discrete integrated circuits was considered, but soon discarded in favor of an FPGA-based design. Designing the logic alone was a difficult task, implementing it in discrete hardware would have pushed the project far over the allotted maximum development time.

RISC-V was chosen as the instruction set architecture for the processor. Its modular design with a very small base instruction set make it easy to implement a basic processor that is still fully compatible with existing software and toolchains.

As a starting point, a Terasic DE0 development board[R] containing an Altera Cyclone III[S] FPGA was borrowed from the school's inventory. It was used to implement a first version of the core.

The only method of synthesis for Altera devices is to use the proprietary Quartus IDE. However, the last version of Quartus to support the Cyclone III series of FPGAs (version 13.1) had already been out of date for several years at the start of the project. Because of this and the increasing resource demand of the developing core, an Arty A7-35T development board[T] with a Xilinx Artix-7[U] FPGA was ordered from Digilent.

The two FPGAs compare as follows:

	Altera EP3C16	Xilinx XC7A35T
Logic Elements	15000	33280
Multipliers	56	90
Block RAM (kb)	504	1800
PLLs	4	5
Global clocks	20	32

The periphery on the development boards:

[R] https://www.terasic.com.tw/cgi-bin/page/archive.pl?No=364

[S] https://www.intel.com/content/www/us/en/products/programmable/fpga/cyclone-iii.html

[T] https://store.digilentinc.com/arty-a7-artix-7-fpga-development-board-for-makers-and-hobbyists/

[U] https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html

	Terasic DE0	Digilent Arty A7-35T
Switches	10	4
Buttons	3	4
LEDs	10 + 4x 7-segment	4 + 3 RGB
GPIOs	2x 36	4x PMOD + chipKIT
Memory	8MB SDRAM	256MB DDR3L
Others	SD card, VGA	Ethernet

While the Digilent board offers fewer IO options, the DDR3 memory can be interfaced using Free memory cores and allows for much larger programs to be loaded, possibly even a full operating system. The missing VGA port has been substituted by a HDMI-compatible DVI interface that is accessible through one of the high-speed PMOD connectors.

10 Tooling

FPGA design is done using a Hardware Description Language (HDL). The two most well-known HDLs are Verilog and VHDL (VHSIC (Very high speed integrated circuit) HDL). As part of our studies at HTL, we exclusively worked with VHDL. For this reason, and because VHDL offers a better type system, it was chosen as the language of choice for the project.

10.1 Vendor Tools

The conventional way to work with FPGA designs is to use the FPGA vendor's development solution for simulation, synthesis and place-and-route. All of these tools are proprietary software specialized to a certain FPGA manufacturer, so a change of hardware also requires changing to a completely different software solution.

Vendor tools are usually free-of-charge for basic usage, but this also means there is no guaranteed support. During the development of this project, several bugs and missing features were found in vendor tools that required workarounds.

10.2 Free Software Tools

A somewhat recent development is the creation of Free Software[V] FPGA toolchains. A breakthrough was achieved by Claire (formerly Clifford) Wolf in 2013 with yosys[22],

[29], a feature-complete Verilog synthesis suite for Lattice's `iCE40` FPGA series. Since then, both yosys and place-and-route tools like nextpnr[30] have matured, however Lattice's iCE40 and ECP5 remained the only supported FPGA architectures for place-and-route.

Thus, two obstacles remained for Free toolchains to be viable for this project: synthesizing from VHDL code and synthesizing to Artix-7 FPGAs. During the development of the project, both of these were solved: Tristan Gingold released ghdlsynth-beta[23], a bridge between GHDL[20] and yosys allowing VHDL to be synthesized just the same as Verilog, and Dave Shah added Xilinx support to nextpnr[24]. The latter was preceded by many months of volunteer work reverse-engineering the Xilinx bitstream format as part of *Project X-Ray*[25].

With these two pieces in place, the project was switched over to a completely Free toolchain, removing any depencies on vendor tools:

- yosys, with ghdl as a frontend for processing VHDL, is used to synthesize the design
- nextpnr-xilinx, together with the Project X-Ray database, is used for place-and-route
- tools from Project X-Ray are used to convert the routed design to a bitstream
- openFPGALoader is used to transfer the bitstream to the FPGA via JTAG

11 Peripherals

11.1 UART

11.2 DVI graphics

The graphics submodule consists of a VGA timing generator, a text renderer with a font ROM, and a DVI encoder frontend:

[V] "Free Software" refers to software that grants its user the freedom to share, study and modify it - see `https://www.fsf.org/about/what-is-free-software`.

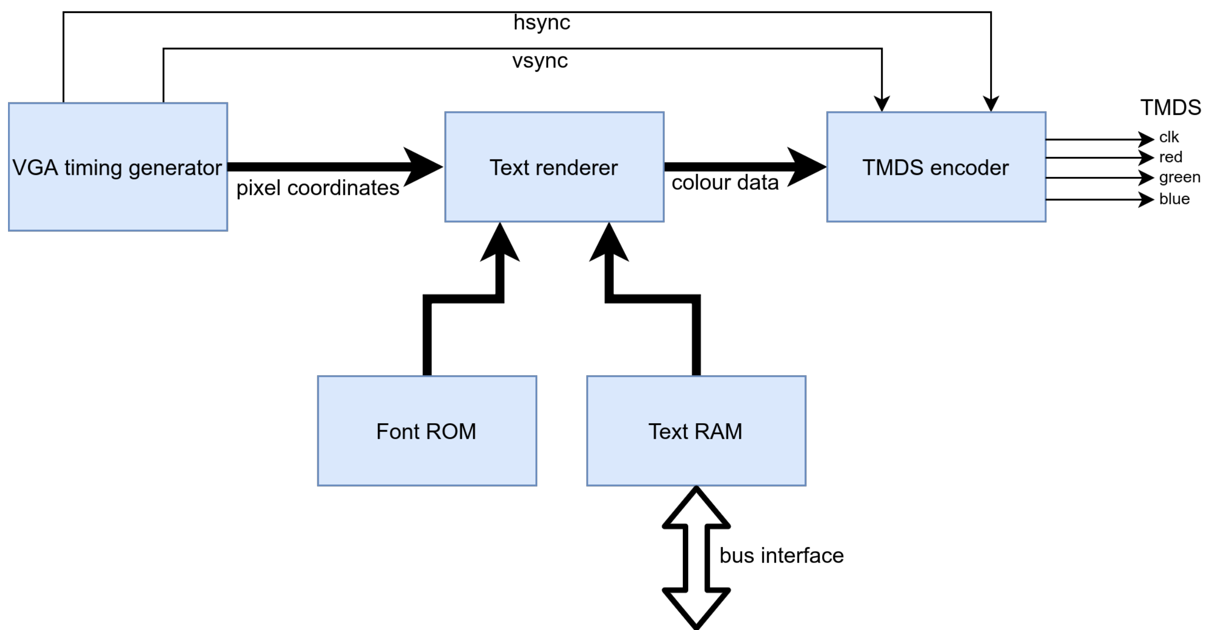

Figure xl: Block diagram of the video core

11.2.1 VGA timing

The timing of VGA signals dates back to analog monitors. Even though this original purpose is only very rarely used nowadays, the timing remained the same for analog and digital DVI all the way to modern HDMI.

In analog screens, the electron beams (one for each primary color red, green and blue) scan across the screen a single horizontal line at a time while being modulated by the color values, resulting in a continuous mixture of all three components. When a beam reaches the end of a scanline, it continues outside the visible area for a small distance (the "Front Porch"), is then sent to the beginning of the next line by a pulse of the hsync (Horizontal Sync) signal, and draws the next line after another short off-screen period (the "Back Porch").

The same applies to vertical timings: after the beam reaches the end of the last line, a few off-screen Front Porch lines follow, then a pulse of the vsync (Vertical Sync) signal sends the beam to the top of the screen, where the first line of the next frame is drawn after several invisible Back Porch lines.

The VGA timing module generates these hsync and vsync signals, along with a blanking signal (active during any front porch, sync and back porch) and, while in the visible area (i.e. not blanking), the row and column of the current pixel relative to the visible area.

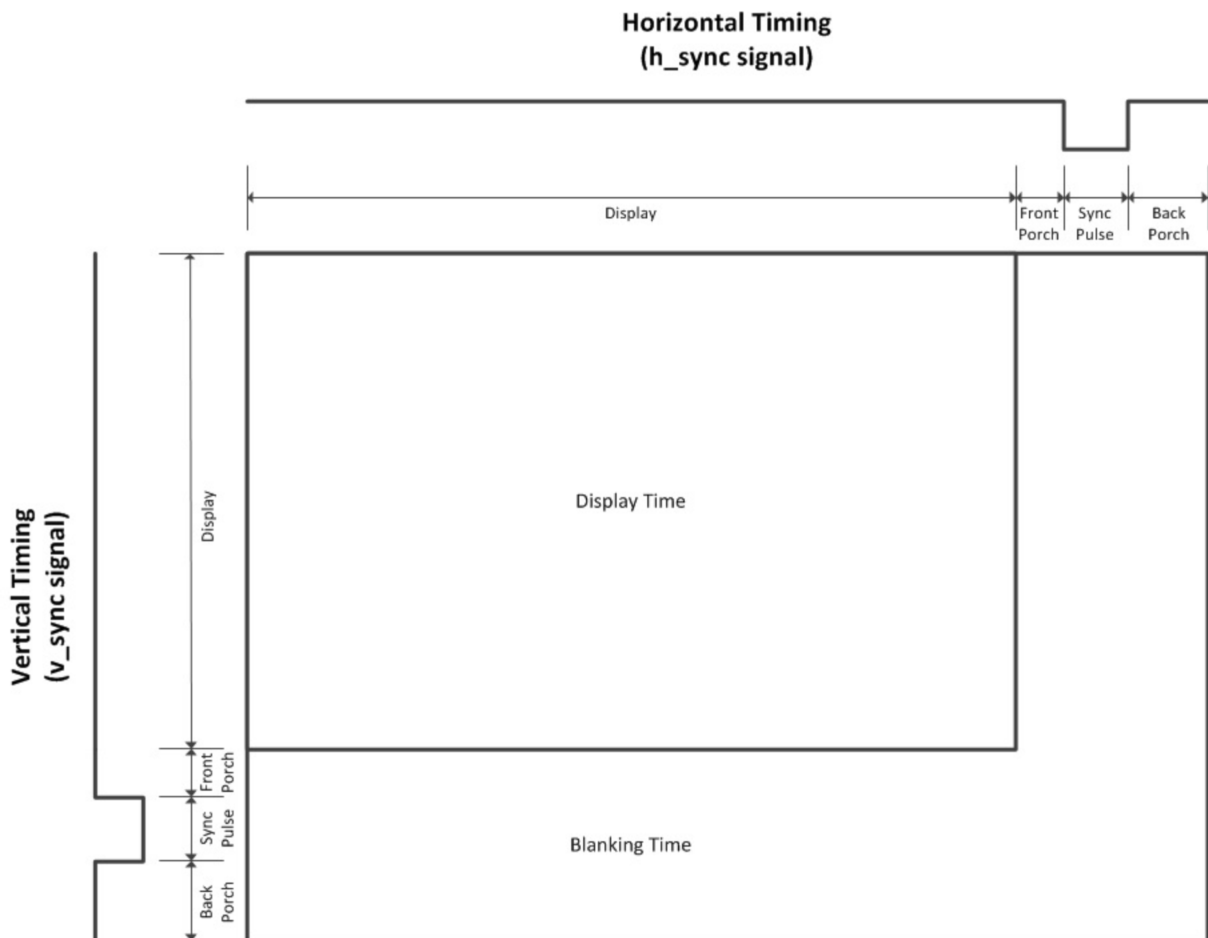

Figure xli: Diagram of VGA timing intervals

11.2.2 Text renderer

The text renderer converts a logical representation of a character, such as its ASCII code (henceforth referred to as its *codepoint*) to a visual representation (a *glyph*). This conversion is achieved using a *font*, a mapping of codepoints to glyphs.

First, the current pixel coordinate (created by the VGA timing generator) is split up into two parts: the character index, which specifies the on-screen character the pixel belongs to, and the offset of the pixel in this character. The character index is passed to the text RAM, which contains the codepoint for each on-screen character. This codepoint, along with the pixel offset, is looked up in the font ROM to determine the color of the pixel.

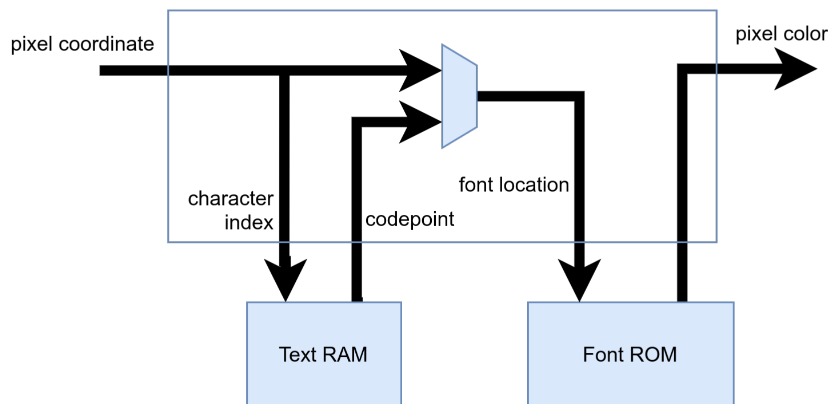

Figure xlii: Block diagram of the text renderer

11.2.3 TMDS encoder

DVI and HDMI are serial digital transmission standards. Three data lines (corresponding to red, green, and blue channels) along with a clock line transmit all color information as well as synchronization signals. The encoding used for these signals is Transition-minimized differential signaling (TMDS). It is a kind of 8b/10b encoding (transforming every 8-bit chunk of data into a 10-bit chunk) that is designed to minimize the number of changes of the output signal.

11.3 Ethernet

The Arty development board contains an RJ-45 Ethernet jack connected to an Ethernet PHY, which exposes a standardized media-independent interface (MII) to the FPGA. The LiteEth core[31], which is released under a Free Software license, is used to integrate the Ethernet interface into the SoC.

11.4 WS2812 driver

A hardware driver for WS2812 serially-addressable RGB LEDs is also included in the SoC. It was developed independently as part of the curriculum at HTL and later incorporated into the SoC.

The driver is designed to be attached to external circuitry that provides color data for any given LED index (address). This can either be discrete logic that generates the color value from the address directly, or a memory that stores a separate color value for each address.

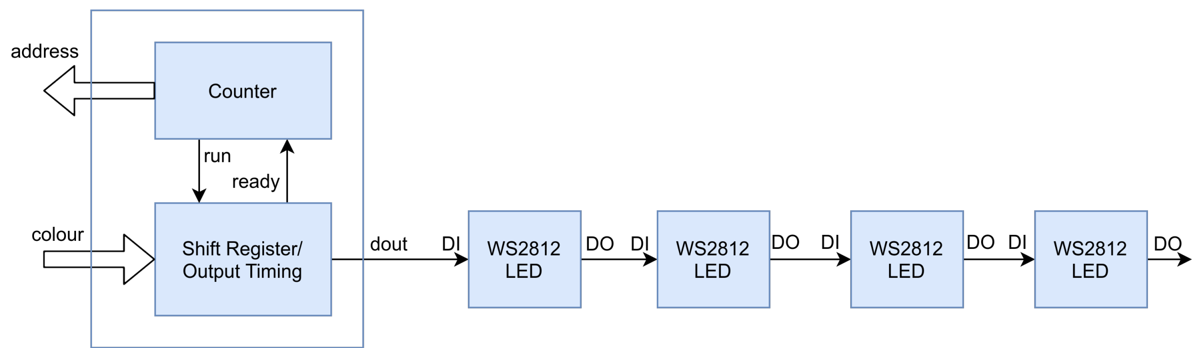

Figure xliii: Block diagram of the WS2812 driver

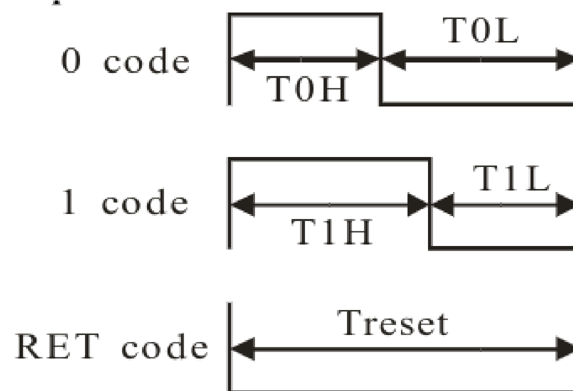

Figure xliv: Timing diagram for the WS2812 serial protocol

The LEDs are controlled using a simple one-wire serial protocol. After a reset (long period of logic 0), the data for all LEDs is transmitted serially in one single blob. Each LED consumes and stores the first 24 bits of the stream and applies them as its color value (8 bits each for red, green, blue), all following bits are passed through unmodified. The second LED thus uses the first 24 bits of the stream it receives, but since the first LED already dropped its data, these are actually the second set of 24 bits of the source data.

Every bit is encoded as a period of logic 1, followed by a period of logic 0. The timing of these sections determines the value, see xliv.

The exact timing differs between models, so all periods can be customized using generics in the VHDL entity.

11.5 DRAM

11.6 External Bus

Bridging the internal SoC bus with the external peripheral bus requires a few steps. For one, the external data bus is bidirectional, so tri-state outputs must be used on the FPGA. In addition, the internal bus arbitrates components using addresses alone, while the external bus uses chip enable signals and overlapping address spaces.

Due to a mistake in the adapter board layout, the nibbles of the address and data buses are reversed (MSB to LSB are pins 7 to 0 on the FPGA, but 3 to 0 followed by 7 to 4 on the board). Thanks to the completely arbitrary mapping of FPGA pins, this can be mitigated without using any additional resources.

12 Testing

12.1 RISC-V Compliance Tests

The RISC-V Compliance Test Suite[32] can be used to empirically confirm the correct functionality of a RISC-V processor. It consists of a series of programs that perform some operations related to a specific feature, then write some result data to a memory region. This memory region is then compared to a "golden signature", which was produced by a processor implementation that is known to be correct.

The initial implementation of the compliance tests uncovered several bugs in the processor core:

- The bitshift instructions (SLL, SRL, SRA, etc.) must, according to the RISC-V standard, only use the lower 5 bits of the second operand as a shift offset. The implementation used all 31 bits instead, causing a test failure.
- Reading a signed value of a size less than 32 bits from memory would not perform proper sign extension. For example, reading a byte value of 0xFF (-1) would result in an expanded machine word of 0x0000_00FF (255) instead of 0xFFFF_FFFF.
- The `SLTIU` (Set less than immediate; unsigned) instruction compares a given register with a constant provided as part of the instruction (the immediate). While

the comparison is unsigned, the 12-bit immediate must be sign-extended as if it were a signed integer. The implementation wrongly assumed that the sign-extension should be unsigned as well.

- The Instruction Set Manual specifies exceptions that must be raised when a misaligned memory access occurs. These exceptions were not yet implemented, but since the compliance tests check for them, the functionality was added to make the tests pass.

Since these tests are easily automated, they were added to the GitLab Continuous Integration (CI) configuration. Whenever a new git commit is pushed to GitLab, the tests are run automatically, and any failures are reported to the responsible committer via email.

Part III

The Core

The core implements the rv32i architecture as specified by the RISC-V standard.

It is constructed according to the traditional RISC pipeline:

Fetch fetches the next instruction from memory.

Decode decodes the instruction into its constituent parts. At the same time, operand values are loaded from any required registers.

Execute performs the action required by the instruction, such as math performed by the Arithmetic Logic Unit (ALU) or writing to Control and Status Registers (CSRs).

Memory loads values from or stores values to the system's main memory or interacts with memory-mapped hardware devices.

Writeback stores a potential result value from Execute or Memory stages to the destination register.

Figure xlv: Block diagram of the CPU core

13 Overview

14 Control

```
entity control is
    generic (
        RESET_VECTOR : yarm_word
    );
    port (
        clk   : in std_logic;
        reset : in std_logic;

        fetch_enable    : out std_logic;
        fetch_ready     : in std_logic;
        fetch_instr_out : in yarm_word;

        decoder_enable         : out std_logic;
        decoder_instr_info_out : in instruction_info_t;

        registers_data_a : in yarm_word;
        registers_data_b : in yarm_word;

        alu_enable_math : out std_logic;
        alu_math_result : in yarm_word;
        alu_valid       : in std_logic;
        alu_enable_cmp  : out std_logic;
        alu_cmp_result  : in compare_result_t;

        csr_enable           : out std_logic;
        csr_ready            : in std_logic;
        csr_data_read        : in yarm_word;
        csr_increase_instret : out std_logic;

        datamem_enable         : out std_logic;
        datamem_ready          : in std_logic;

        alignment_raise_exc : out std_logic;
        alignment_exc_data  : out exception_data_t;

        registers_read_enable  : out std_logic;
        registers_write_enable : out std_logic;

        -- TRAP CONTROL

        may_interrupt     : out std_logic;
        -- the stage that will receive an interrupt exception
        interrupted_stage : out pipeline_stage_t;

        do_trap     : in std_logic;
        trap_vector : in yarm_word;

        trap_return_vec : in yarm_word;
        return_trap     : out std_logic;

        -- instruction info records used as input for the respective stages
```

```
        stage_inputs : out pipeline_frames_t
    );
end control;
```

control.vhd

The control unit is responsible for coordinating subcomponents and the data flow between them. Internally, it is based on `instruction_info_t` structures, which contain all the information required to pass an instruction along the different pipeline stages. Before the fetch stage, when an instruction is first scheduled, it contains only the instruction's address (because nothing else is known about it). Then, information is added incrementally by the different stages.

15 Decoder

```
entity decoder is
    port (
        clk    : in std_logic;
        enable : in std_logic;

        async_addr_rs1 : out register_addr_t;
        async_addr_rs2 : out register_addr_t;

        alu_muxsel_a    : out mux_selector_t;
        alu_muxsel_b    : out mux_selector_t;
        alu_muxsel_cmp2 : out mux_selector_t;

        csr_muxsel_in : out mux_selector_t;

        instr_info_in  : in instruction_info_t;
        instr_info_out : out instruction_info_t;

        raise_exc : out std_logic;
        exc_data  : out exception_data_t
    );
end decoder;
```

decoder.vhd

The decoder receives an instruction and interprets it. Among others, it determines

- The source and destination register addresses
- The pipeline stages that need to be run for the instruction
- The ALU operation, if any
- Whether the instruction should branch, and if so, under what condition

16 Registers

```
entity registers is
    port (
        clk    : in std_logic;

        read_enable  : in std_logic;
        write_enable : in std_logic;

        addr_a : in register_addr_t;
        addr_b : in register_addr_t;
        addr_d : in register_addr_t;

        data_a : out yarm_word;
        data_b : out yarm_word;
        data_d : in yarm_word
    );
end registers;
```

registers.vhd

The registers store the 32 general-purpose values required by rv32i (each 32-bit wide). They are accessible through two read ports and one write port. As specified by the RISC-V standard, the first register (`x0`) is hard-wired to 0, and any writes to it are ignored.

17 Arithmetic and Logic Unit (ALU)

```
entity alu is
    port (
        clk : in std_logic;

        enable_math : in std_logic;
        valid       : out std_logic;
        operation   : in alu_operation_t;
        a, b        : in yarm_word;
        math_result : out yarm_word;

        -- compare inputs
        -- do signed comparisons
        enable_cmp : in std_logic;
        cmp_signed : in std_logic;
        cmp1, cmp2 : in yarm_word;
        cmp_result : out compare_result_t
    );
end alu;
```

alu.vhd

The ALU contains a math/logic unit as well as a comparator. It is used both explicitly

by instructions such as `add` or `shiftl`, as well as to add offsets to base addresses for memory instructions and to decide whether an instructions should branch.

18 Control and Status Registers (CSR)

```
entity csr is
    generic (
        HART_ID : integer
    );
    port (
        clk    : in std_logic;
        reset  : in std_logic;
        enable : in std_logic;
        ready  : out std_logic;

        instr_info_in : in instruction_info_t;
        data_write    : in yarm_word;
        data_read     : out yarm_word;

        increase_instret : in std_logic;

        external_int : in std_logic;
        timer_int    : in std_logic;
        software_int : in std_logic;

        interrupts_pending : out yarm_word;
        interrupts_enabled : out yarm_word;
        global_int_enabled : out std_logic;
        mtvec_out : out yarm_word;
        mepc_out  : out yarm_word;

        do_trap       : in std_logic;
        return_m_trap : in std_logic;
        mepc_in       : in yarm_word;
        mcause_in     : in yarm_trap_cause;
        mtval_in      : in yarm_word;

        raise_exc : out std_logic;
        exc_data  : out exception_data_t
    );
end csr;
```

csr.vhd

The control and status registers contain configurations relevant to the core itself. For example, they can be used to control interrupts.

19 Memory Arbiter

```
entity memory_arbiter is
    port (
```

```
3         clk   : in std_logic;
4         reset : in std_logic;
5
6         fetch_enable    : in std_logic;
7         fetch_ready     : out std_logic;
8         fetch_address   : in yarm_word;
9         fetch_instr_out : out yarm_word;
10
11        fetch_raise_exc : out std_logic;
12        fetch_exc_data  : out exception_data_t;
13
14        datamem_enable        : in std_logic;
15        datamem_ready         : out std_logic;
16        datamem_instr_info_in : in instruction_info_t;
17        datamem_read_data     : out yarm_word;
18
19        datamem_raise_exc : out std_logic;
20        datamem_exc_data  : out exception_data_t;
21
22        -- little-endian memory interface, 4 byte address alignment
23        MEM_addr        : out yarm_word;
24        MEM_read        : out std_logic;
25        MEM_write       : out std_logic;
26        MEM_ready       : in std_logic;
27        MEM_byte_enable : out std_logic_vector(3 downto 0);
28        MEM_data_read   : in yarm_word;
29        MEM_data_write  : out yarm_word
30    );
31 end memory_arbiter;
```

memory_arbiter.vhd

Since both fetch and memory stages need to access the same system memory, access to this common resource has to be controlled. The memory arbiter acts as a proxy for both fetch and data memory requests and stalls either until the other one completes.

20 Exception Control

```
1  entity exception_control is
2      port (
3          clk   : in std_logic;
4
5          fetch_raise_exc : in std_logic;
6          fetch_exc_data  : in exception_data_t;
7
8          -- synchronous exceptions
9          decoder_raise_exc : in std_logic;
10         decoder_exc_data  : in exception_data_t;
11
12         csr_raise_exc : in std_logic;
13         csr_exc_data  : in exception_data_t;
14
15         alignment_raise_exc : in std_logic;
16         alignment_exc_data  : in exception_data_t;
17
```

```vhdl
18          datamem_raise_exc : in std_logic;
19          datamem_exc_data  : in exception_data_t;
20
21          -- interrupts
22          global_int_enabled : in std_logic;
23          interrupts_enabled : in yarm_word;
24          interrupts_pending : in yarm_word;
25
26          -- stage inputs for return address + trap value (instruction)
27          stage_inputs      : in pipeline_frames_t;
28          interrupted_stage : in pipeline_stage_t;
29
30          may_interrupt : in std_logic;
31          do_trap       : out std_logic;
32          trap_cause    : out yarm_trap_cause;
33          trap_address  : out yarm_word;
34          trap_value    : out yarm_word
35      );
36  end exception_control;
```

exception_control.vhd

Several components in the core may raise a synchronous exception when an unexpected error (such as a malformed instruction or an unaligned memory access) occurs. Additionally, asynchronous interrupts (like from a timer or a UART) can be triggered externally. When an exception or an enabled interrupt is registered, program flow is diverted to the trap handler, defined using the machine trap vector (`mtvec`) CSR.

21 Erklärung der Eigenständigkeit der Arbeit

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe. Meine Arbeit darf öffentlich zugänglich gemacht werden, wenn kein Sperrvermerk vorliegt.

Ort, Datum	Armin Brauns
Ort, Datum	Daniel Plank

I List of Figures

i An overview of the hardware peripherials . . . iv
ii Atari PBI Pinout;Source: `https://www.atarimagazines.com` . . . 2
iii System bus structural diagram; Source: `https://en.wikipedia.org/` 3
iv Harvard(left) vs Von-Neumann architecture(right); Source: `https://en.wikipedia.org/` . . . 6
v Digilent Analog Discovery 2;Source: `https://www.sparkfun.com/` 7
vi The ATMega 2560 module for the backplane . . . 8
vii Layout of the DIN41612 Connectors on the Backplane . . . 9
viii Measurement at around 1MHz bus clock on MS1 . . . 10
ix The case with installed backplane . . . 11
x PC-16550D Pinout[5] . . . 12
xi The schematic of the UART Module . . . 14
xii Measurement of the 1.8432 MHz Output on J1 . . . 15
xiii Measurement of a character transmission before and after MAX-232 . . 16
xiv Pinout of the RJ-45 Plug; Src: `https://www.wti.com/` . . . 16
xv Measurement of a character echo . . . 17
xvi Transmission of character A via the 16550 UART . . . 20
xvii The final uart module with the pc16550 uart in the center . . . 22
xviii TLC-7528 Pinout[7] . . . 23
xix IDT-7201 Pinout[8] . . . 24
xx TLC-7528 in voltage modet[7] . . . 25
xxi Measurement of a generated SAW signal via the TLC7528 . . . 25
xxii The schematic of the DAC Module . . . 26
xxiii Measurement of a generated SAW signal with the FIFO Empty flag . . . 29
xxiv A transmission between the FIFO and the DAC . . . 29
xxv A fifo store operation in contrast to the load operation . . . 30
xxvi Storage and retrieval of a sine to and from the FIFO . . . 32
xxvii Measuremet of the generated sine from the sine LUT on DACA and DACB 32
xxviii The final DAC module . . . 34
xxix 3.3V to 5V conversion using the level shifter . . . 35
xxx 5V to 3.3V conversion using the level shifter . . . 36
xxxi The internal schematics of the level shifter[13] . . . 36
xxxii The internal clamping diodes of the Analog Discovery 2[3] . . . 37
xxxiii The final FPGA interface module with the level shifters . . . 38
xxxiv A Flow-Chart of the program execution path . . . 43
xxxv The output of an example track part 1 . . . 50

xxxvi The output of an example track part 2 51
xxxvii A regular beginning of the game 56
xxxviii A state diagram of the computer state machine 59
xxxix Screenshot of the resulting waveform in GTKWave 63
xl Block diagram of the video core 68
xli Diagram of VGA timing intervals 69
xlii Block diagram of the text renderer 70
xliii Block diagram of the WS2812 driver 71
xliv Timing diagram for the WS2812 serial protocol 71
xlv Block diagram of the CPU core 74

II List of Tables

1 The layout of the Data Bus on read 28

III Listings

I Read and write routines for the 16550 UART 17
II 16550 INIT routines and single char transmission 19
III 16550 character echo 21
IV SAW Generation for the DAC with FIFO 30
V Sine LUT Generation 30
VI DAC Sine Generation 31
VII The avr.h header file 40
VIII The routine function looped by the main 41
IX The routine function for the UART 42
X The routine function for the DAC 42
XI The DAC operation modes 45
XII The DAC waveform generation code 45
XIII The ISR which fires every millisecond 48
XIV The sound update function 48
XV The character ingest function 52
XVI The command parsing function 53
XVII The command execution routine 54
XVIII The computer FSM 58
XIX Counter entity 60

XX Counter test bench entity 61
XXI Commands required to simulate the counter design 62
XXII Counter design constraints file 63
XXIII Commands required to synthesize the counter design 64
XXIV Header for control entity 74
XXV Header for decoder entity 75
XXVI Header for registers entity 76
XXVII Header for alu entity 76
XXVIII Header for csr entity 77
XXIX Header for memory_arbiter entity 77
XXX Header for exception_control entity 78

LITERATURVERZEICHNIS

[1] First Draft of a Report on the EDVAC. Report. United States Army Ordnance Department and the University of Pennsylvania, June 1945. URL: http://abelgo.cn/cs101/papers/Neumann.pdf.

[2] Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V. Atmel Corporation. Feb. 2014. URL: https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf.

[3] Analog Discovery 2 Reference Manual. Digilent, Inc. Sept. 2015. URL: https://reference.digilentinc.com/_media/reference/instrumentation/analog-discovery-2/ad2_rm.pdf.

[4] Interface Between Data Terminal Equipment and Data Circuit- Terminating Equipment Employing Serial Binary Data Interchange. Standard. Oct. 1997.

[5] PC16550D Universal Asynchronous Receiver/Transmitter With FIFOs. Texas Instruments Inc. 1995. URL: https://www.scs.stanford.edu/10wi-cs140/pintos/specs/pc16550d.pdf.

[6] MAX232x Dual EIA-232 Drivers/Receivers. Texas Instruments Inc. Feb. 1989. URL: https://www.ti.com/lit/ds/symlink/max232.pdf.

[7] DUAL 8-BIT MUTLIPLYING DIGITAL-TO-ANALOG CONVERTERS. Texas Instruments Inc. 1987. URL: https://www.ti.com/lit/ds/symlink/tlc7528.pdf.

[8] Integrated Device Technology, Inc.: CMOS ASYNCHRONOUS FIFO. RENESAS. 2002. URL: http://www.komponenten.es.aau.dk/fileadmin/komponenten/Data_Sheet/Memory/IDT7201.pdf.

[9] High-Speed CMOS Logic Octal D-Type Flip-Flop, 3-State Positive-Edge Triggered. Texas Instruments Inc. Feb. 1998. URL: https://www.ti.com/lit/ds/schs183c/schs183c.pdf.

[10] SNx4HC00 Quadruple 2-Input Positive-NAND Gates. Texas Instruments Inc. Dec. 1982. URL: https://www.ti.com/lit/ds/symlink/sn74hc00.pdf.

[11] Compact disc digital audio system. Standard. International Electrotechnical Commission, Sept. 1987.

[12] Ethan Winer: The Audio Expert: Everything You Need to Know About Audio. Focal Press, 2013. URL: https://books.google.com/books?id=TIfOAwAAQBAJ&pg=PA107#v=onepage&q=-%2010%20dbv&f=false.

[13] Jenny List: „Taking It To Another Level: Making 3.3V Speak With 5V“. In: (Dec. 2016). URL: https://hackaday.com/2016/12/05/taking-it-to-another-level-making-3-3v-and-5v-logic-communicate-with-level-shifters/.

[14] Schottky Barrier Diode DB3S406F0L Silicon epitaxial planar type. Panasonic. Mar. 2010. URL: https://industrial.panasonic.com/content/data/SC/ds/ds4/DB3S406F0L_E.pdf.

[15] Ron Schnell: Dunnet Source Code. Emacs. 1982. URL: https://github.com/jwiegley/emacs-release/blob/master/lisp/play/dunnet.el.

[16] ASCII Format for Network Interchange. Standard. Network Working Group, Oct. 1969. URL: https://tools.ietf.org/pdf/rfc20.pdf.

[17] VT100 SERIES TECHNICAL MANUAL. Digital Equipment Corporation. 1979. URL: https://vt100.net/docs/vt100-tm/ek-vt100-tm-002.pdf.

[18] Tool Interface Standard (TIS)Executable and Linking Format (ELF) Specification. Standard. TIS Committee, May 1995. URL: https://refspecs.linuxbase.org/elf/elf.pdf.

[19] Unknown Author: Data in Program Space. avr-libc 2.0.0 Standard C library for AVR-GCC. 2016. URL: https://www.nongnu.org/avr-libc/user-manual/pgmspace.html.

[20] Tristan Gingold: ghdl. URL: https://github.com/ghdl/ghdl.

[21] Tony Bybell: GTKWave. URL: http://gtkwave.sourceforge.net.

[22] Various Contributors: Yosys - Yosys Open SYnthesis Suite. URL: https://github.com/YosysHQ/yosys.

[23] Tristan Gingold: ghdlsynth-beta. URL: https://github.com/tgingold/ghdlsynth-beta.

[24] David Shah: nextpnr-xilinx. URL: https://github.com/daveshah1/nextpnr-xilinx.

[25] SymbiFlow: Project X-Ray. URL: https://github.com/SymbiFlow/prjxray.

[26] Gwenhael Goavec-Merou: openFPGALoader. URL: https://github.com/trabucayre/openFPGALoader.

[27] Olav Junker Kjær: The Nand Game. URL: http://nandgame.com.

[28] Ben Eater: Building an 8-bit breadboard computer! 2016. URL: https://www.youtube.com/playlist?list=PLowKtXNTBypGqImE405J2565dvjafglHU.

[29] Johann Glaser Clifford Wolf: „Yosys - A Free Verilog Synthesis Suite“. 2013. URL: http://www.clifford.at/yosys/files/yosys-austrochip2013.pdf.

[30] Various Contributors: nextpnr - a portable FPGA place and route tool. URL: https://github.com/YosysHQ/nextpnr.

[31] Florent Kermarrec: LiteEth. URL: https://github.com/enjoy-digital/liteeth.

[32] Lee Moore Jeremy Bennett: RISC-V Compliance Task Group. URL: https://github.com/riscv/riscv-compliance.

Anhang