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 31. 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, wodurch ebenfalls die Lesbarkeit erhöht wird. Die entstandene Dokumentation soll für Menschen mit einem grundlegenden Verständnis für Elektronik sowie der Hardware-Beschreibungssprache VHDL verständlich sein.

This diploma thesis demonstrates the operation of processors and their corresponding peripherals, both in modern and traditional forms. It attempts to illustrate the structure of a computer system in hard- and software. To reach this goal, a RISC-V processor was implemented in VHDL on a Xilinx FPGA and some parallel bus peripherals were designed using discrete hardware. These peripherals 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, and in a resulting effort to increase readability, this thesis is written in English. The written documentation should be comprehensible for anyone with a basic understanding of electronics as well as the hardware description language VHDL.

Result

The project is fully implemented with all functionality originally targeted. The system has been tested and verified. All example code has been documented and tested. Hardware implementations were created using Free software[1] programs, while the RISC-V processor can be compiled with a Free 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 peripherals can be seen in Figure 1.

[1] See subsection 1.1

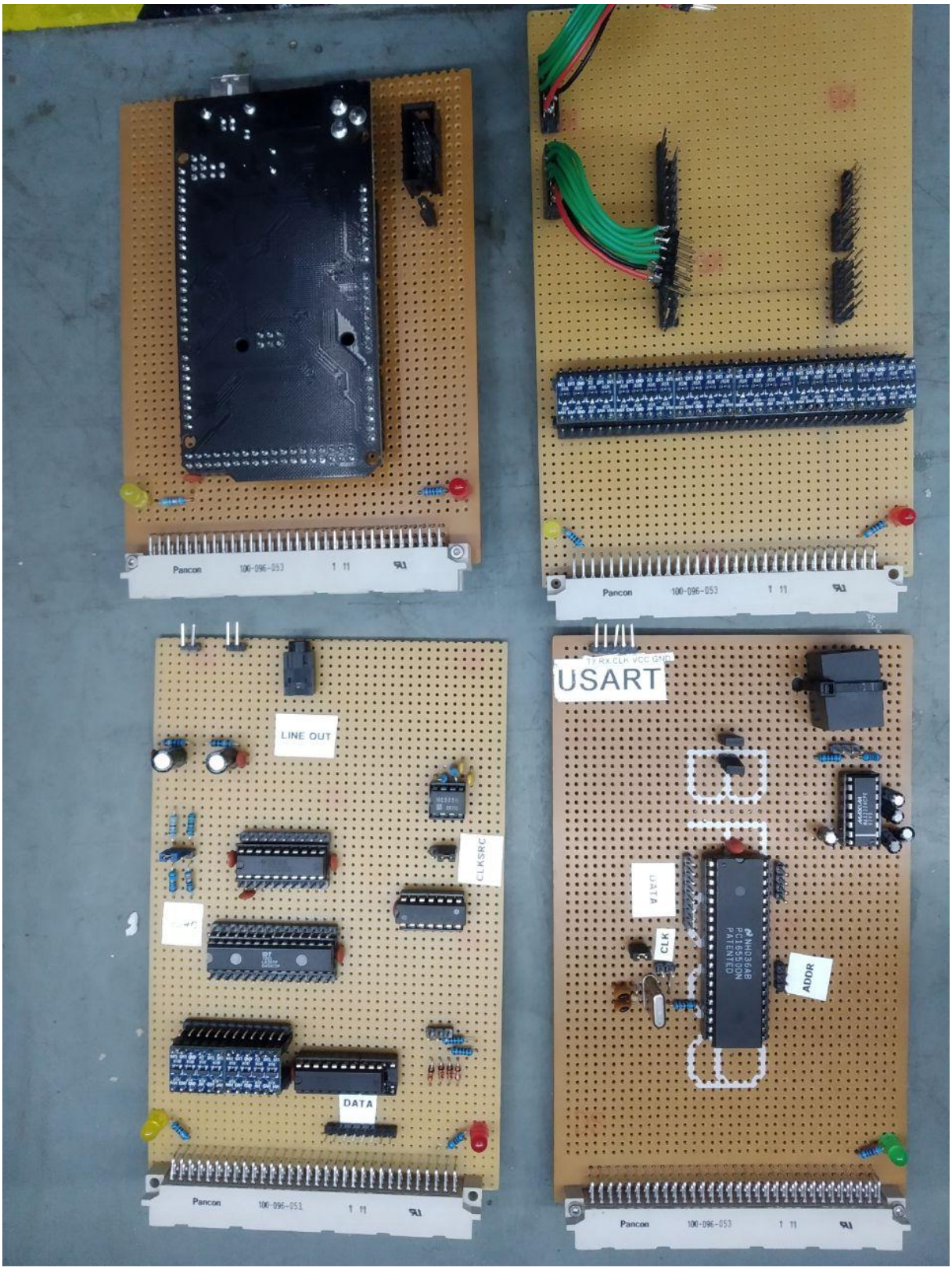

Figure 1: An overview of the hardware peripherals

Contents

1 Introduction

In early 2018, more than a year before the official start of the project, after searching for a subject for the diploma thesis, the idea of building a computer from scratch had come up. Multiple suggestions on how to implement it and the scope of the project were gathered. Originally, the goal was to design a computer consisting of seperate plug-in cards, one instruction would residing on each. This would open up the “black box“ of modern processor design, showing the basic components at a macroscopic scale.

The project’s aim was later redirected due to concerns about difficulty, and an FPGA-based design was opted for instead. After several months of implementation time, the project was split into two parts: the peripherals and the core processor. During the development process, and to get back to the original goal of making a processor understandable, the peripherals changed from being implemented in VHDL back to hardware. This increased the required effort, but would result in a far more understandable final product.

The decision to use a RISC-V based processor was made at the beginning of the project because the core architecture is well documented and modular, and because almost any feature not implemented inside the processor can be emulated using software instead.

1.1 Free software

For most of today’s processors, documentation only exists on the execution of programs (the runtime), not for their internals. In order the have the biggest possible educational potential, this project is entirely "Free as in speech": All involved software and hardware designs, as well as all the tools and utilities required to create them, comply with the Free Software Foundation’s definition for Free software [1]. They give the users the rights to share, study and modify them at their will. In this thesis, the capital-F “Free” is used to refer to this definition rather than the meaning of “free of charge” or “gratis”.

2 Task description

2.1 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[2] 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 constructed in Free 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.

3 Organization

3.1 Hardware peripherals

Planning of the peripherals was done based on the information provided on large parts by David Oberhollenzer. A lot of his advice contributed heavily to the direction the development went.

3.1.1 Peripherial selection

The selection of the hardware peripherals was done based on implementation difficulty, common use in computer systems, relevance in current times and wether they were fitting for demonstrative purposes.

[2] From now on reffered to simply as DAC

Serial communication interface Serial communication interfaces have been around for a long time. They have been used for many different applications from early mouse pointer devices [2] to user input terminals[3] which are far away from the real computer system. They are still very common in smaller embedded sytems and in the server space where they are used as a simple and less error prone way to interface with the operating system and programs running there. They are fairly easy to implement as there are a interface ICs which provide a more generic interface for serial communications [4]. Most SOCs [3] have some form of serial communication interface. The most common serial interface voltages are 3.3V, 5V and levels as per TIA-/EIA-232 specification[5].

Parallel Port interface Parallel ports are absent on most modern computer systems but historically have been the high speed interfaces on computers. Early computer systems used parallel-ports for expansions and the ISA-Bus [4] was for some time the main way of expansion for PCs [5]. Most younger people remeber parallel ports as the port for printers on their home PCs. A prallel port is easy to implement because it has simmilar use of control, data and address lines like a processor uses internally anyways[6]. Usage of the standard IEEE 1284 port limits the design to the signals on this port or makes the use of the signals on this port obligatory.

Digital to Analog Converter Digital to Analog Converters or more commonly DACs are used on all modern PCs for sound output. They have been around for longer and some external sound card interfaces have been standardisedlike AC '97[7]. Implementation of a standard audio interface requires higher speed connections or more precise timing for ac97 for example. Earlier computer systems did not have a sound card as it doesn't have import usage for computing and user input tasks and later on computer systems only had a PC speaker for diagnostics such as the IBM PC AT [8] which can only procude one specific frequency and does not have a DAC. A dac is not easy to implement as it requires a constant sampling rate and a buffer to be of any practical use.

Graphical output / GPU Graphical output on older computer systems such as the EDVAC [9] was not possible because it requires either a heavy load on the processor or dedicated hardware and due to the mostly scientific use it was easier to just print the caracters as letters via a printer. Drawing characters onto a screen is by itself not an

[3] SOC... System on a Chip
[4] ISA...Industry Standard Architecture
[5] PC in this thesis referrs to Computer Systems using the x86 Architecture

easy task as it requires, for example for VGA a Digital to Analog Converter with 25MHz sampling rate and a buffer to contain all needed data for one frame or at least parts of it, while the CPU renders the frame[10]. Screen output is one of the if not the most common form of output on a computer today.

Inter Integrated Circuit Inter Integrated Circuit or IIC for short is a standard for serial transmission between Integrated circuits[11]. This is done on a master-slave basis and transmission speed is fairly low in standard 100kBit/s mode. The bus is used on many different platforms for many different things including HDMI DDC [12]. Though there are some IIC ICs which can interface with a parallel bus such as the PCA9564 [13] but these are either limited in capability or not easy to use and implement. Most people don't have an understanding of IIC as it is only known in technical fields.

Utility analysis Among the above mentioned processor peripherals from the criteria mentioned before a utility analysis was performed. To do this different point have been credited for the criteria mentioned which can be seen in Table 1. The multipliers in Table 1 have been applied to the points and the sums in Table 3 resulted. Based on this result the DAC and Serial Communication interface were chosen as peripherals.

Criteria	serial port	parallel port	DAC	GPU	IIC
implementation	0	0	1	4	2
common use	2	1	3	3	1
relevance	2	1	3	3	1
demonstrative	2	1	3	2	1

Table 1: utility analysis base points for peripherals

Criteria	multiplier
implementation	-2
common use	1
relevance	2
demonstrative	3

Table 2: utility analysis multipliers for peripherals

Criteria	serial port	parallel port	DAC	GPU	IIC
implementation	0	0	-2	-8	-4
common use	2	1	3	3	1
relevance	4	2	6	6	2
demonstrative	6	3	9	6	3
SUM	12	6	16	7	2

Table 3: utility analysis results for peripherals

4 Hardware peripherals

4.1 Parallel bus

The core part of the hardware is the interface between the microprocessor and the hardware peripherals. 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 2 the layout of the Atari Parallel Bus Interface is shown as used on the Atari 800XL.

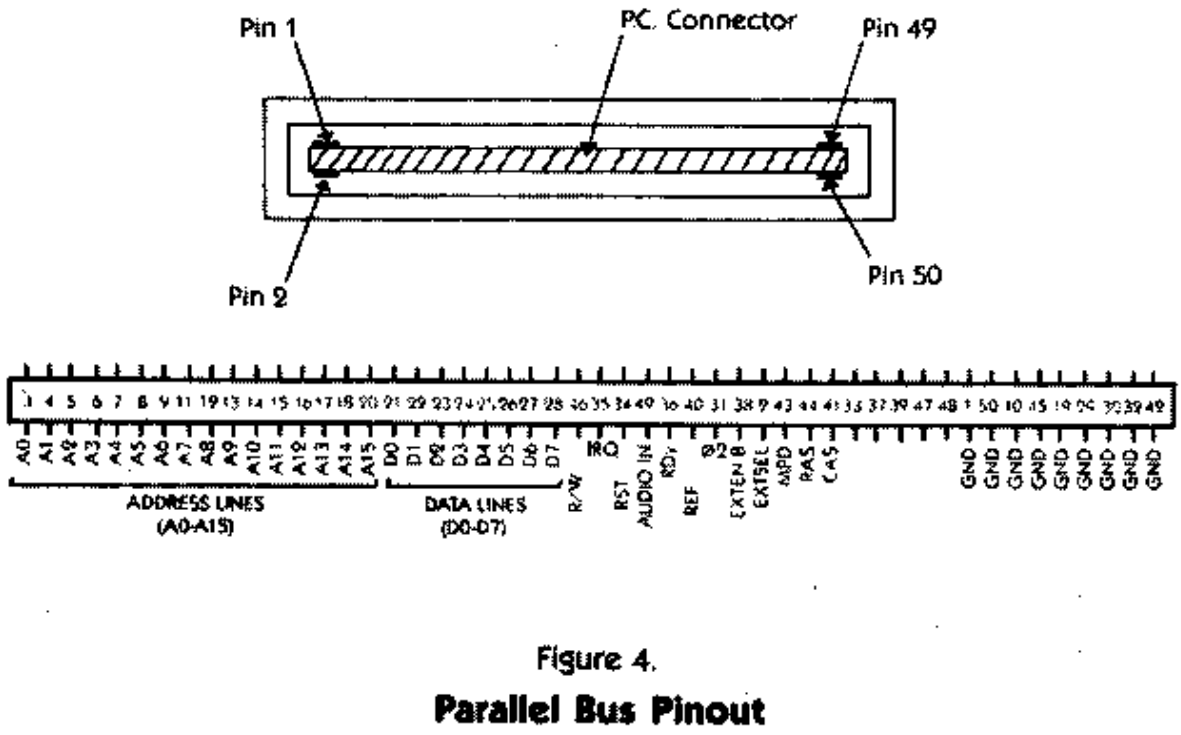

Figure 4.
Parallel Bus Pinout

Figure 2: 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 [6] and the picture seen in figure 3, 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.

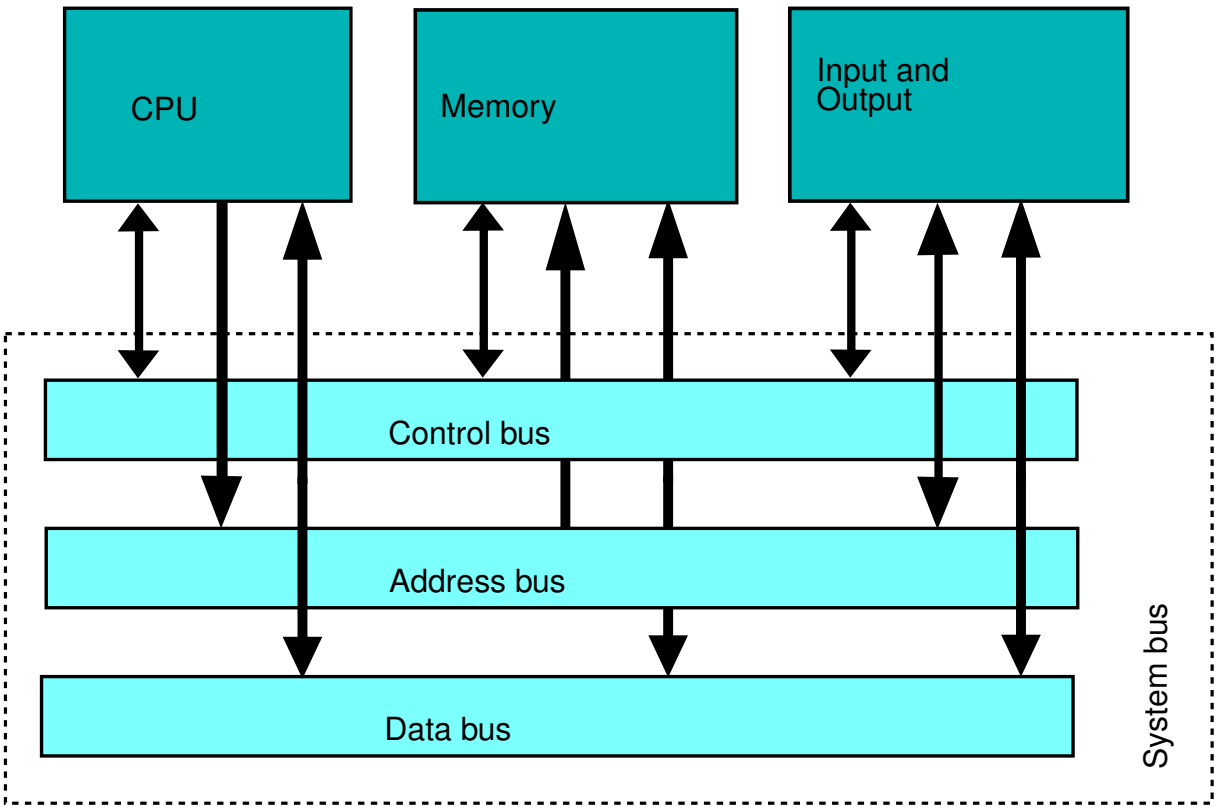

Figure 3: System bus structural diagram; Source: `https://en.wikipedia.org/`

4.1.1 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.

[6] `https://en.wikipedia.org/wiki/System_bus`

4.1.2 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 is shrunk down in our case to 8 bits. The data bus is bidirectional.

4.1.3 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:

Signal	Description
MR	Master Reset
$\neg WR$	Write Not
$\neg RD$	Read Not
$\neg MS1$	Module Select 1 Not
$\neg MS2$	Module Select2 Not

Table 4: Signals on the control bus

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

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.

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.

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.

Sepearation of $\neg RD/\neg WR$ **and**$\neg MS1/\neg MS2$ The Read Not and Write Not lines could be combined into one line $\neg RD/WR$. The same can be done for the Module Select lanes. In both cases this would have made wiring inside the finished modules more difficult and if both were combined the bus would not be able to not perform an action at any given point in time. Therefore these signals have not been combined.

4.2 Von Neumann Archtiecture

The term “von Neumann architecture“ refers 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[7] or the CA [8] as well as means to interface with its operator[9].

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.“[9, 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 peripherals 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“[14, 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 4

[7] ALU...arithmetic logic unit
[8] CA...Central Arithmetic Part

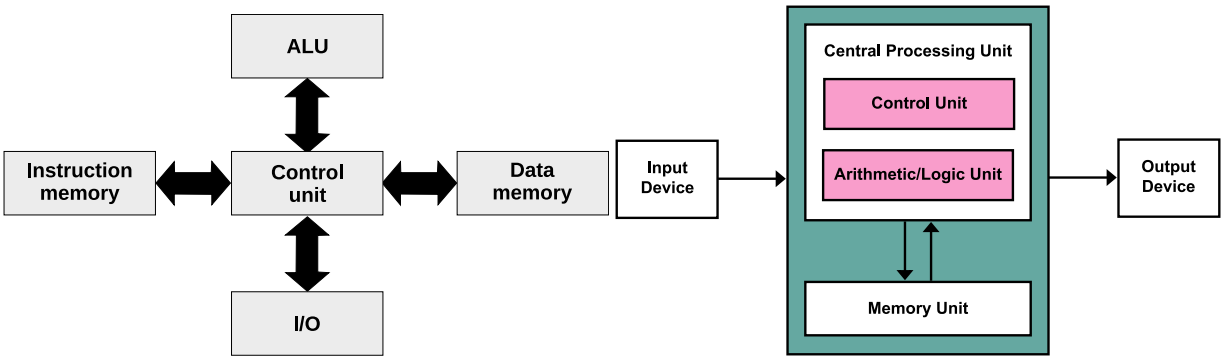

Figure 4: Harvard(left) vs Von-Neumann architecture(right);
Source: `https://en.wikipedia.org/`

4.3 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.

4.3.1 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[15]. These were enough for all nescessary measurements. Though due to the size and construction of the device, which can be seen in figure 5, errors were encountered while performing the measurements. These are noted on occurance.

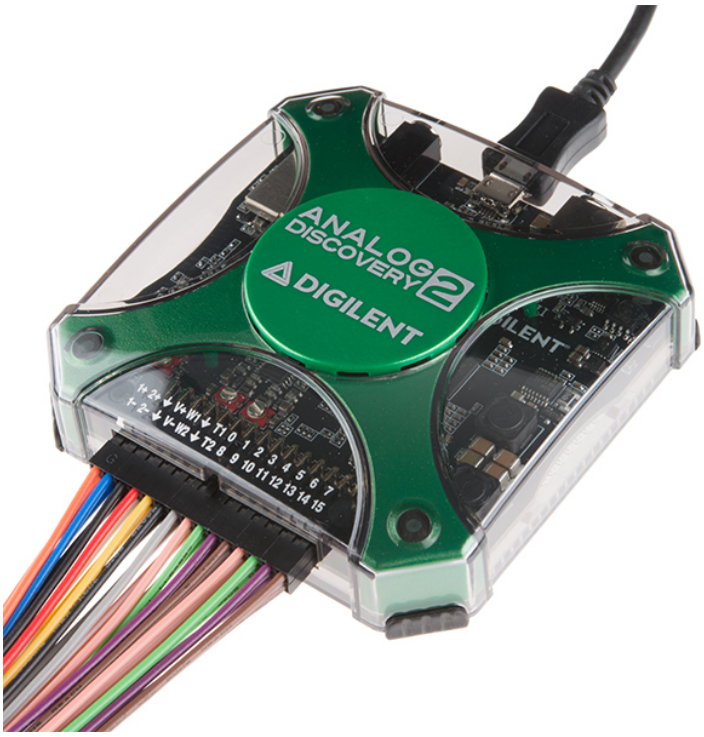

Figure 5: Digilent Analog Discovery 2;Source: `https://www.sparkfun.com/`

4.3.2 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 peripherals, way of addressing pins (8 at a time) and availability. [14] 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 6. The ATMega is beneath the the black PCB [9] 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 peripherals.

[9] Printed circuit board

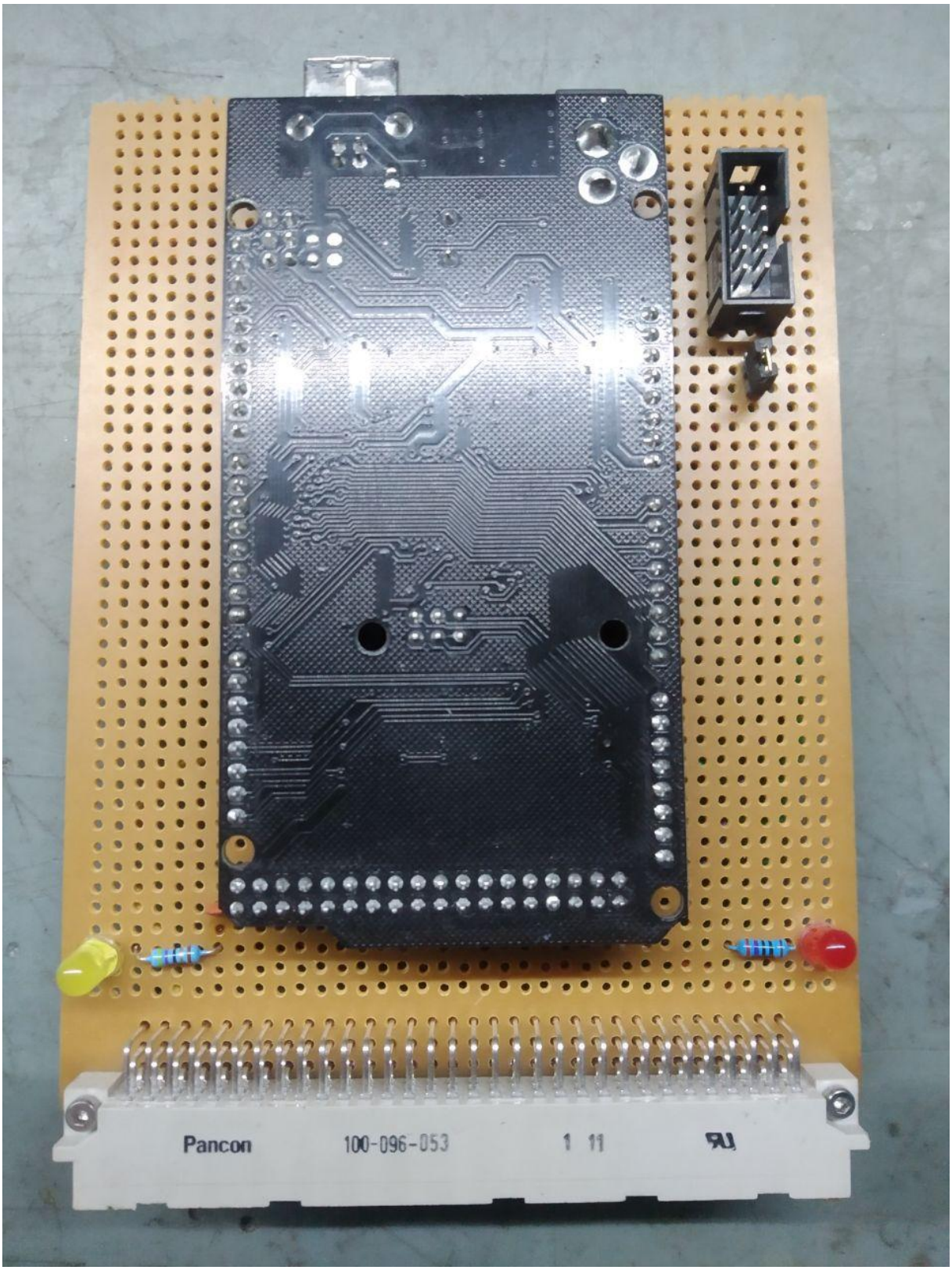

Figure 6: The ATMega 2560 module for the backplane

4.4 Backplane

To connect the modules to the microprocessor, many pins need to be connected straight through. For this purpose a backplane with DIN41612 connectors is beeingused . These connectors are used 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 7.

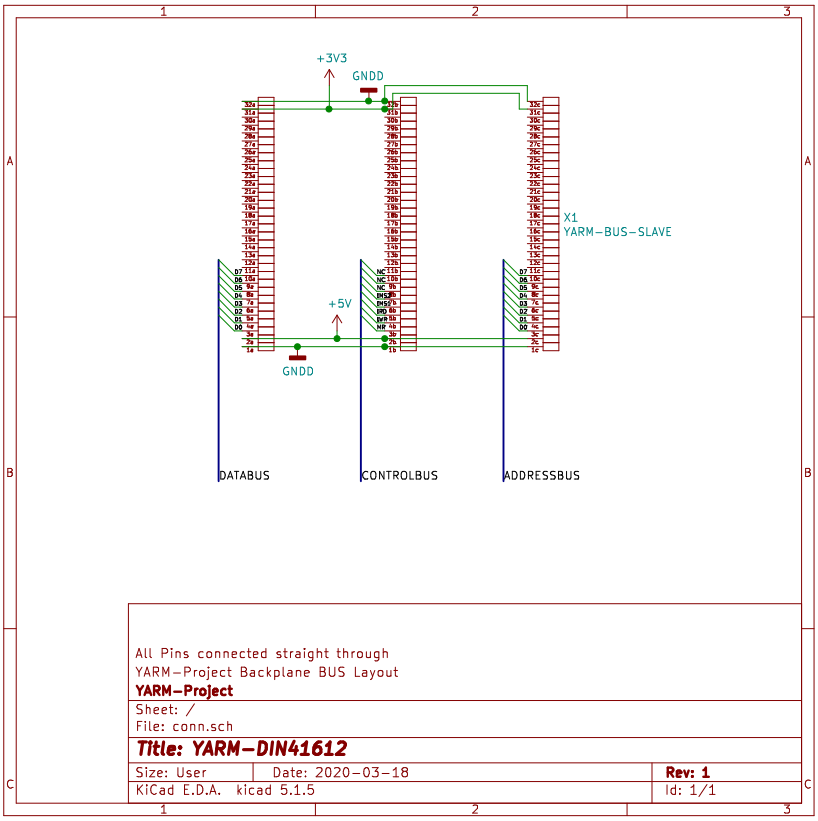

Figure 7: Layout of the DIN41612 Connectors on the Backplane

4.4.1 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 8

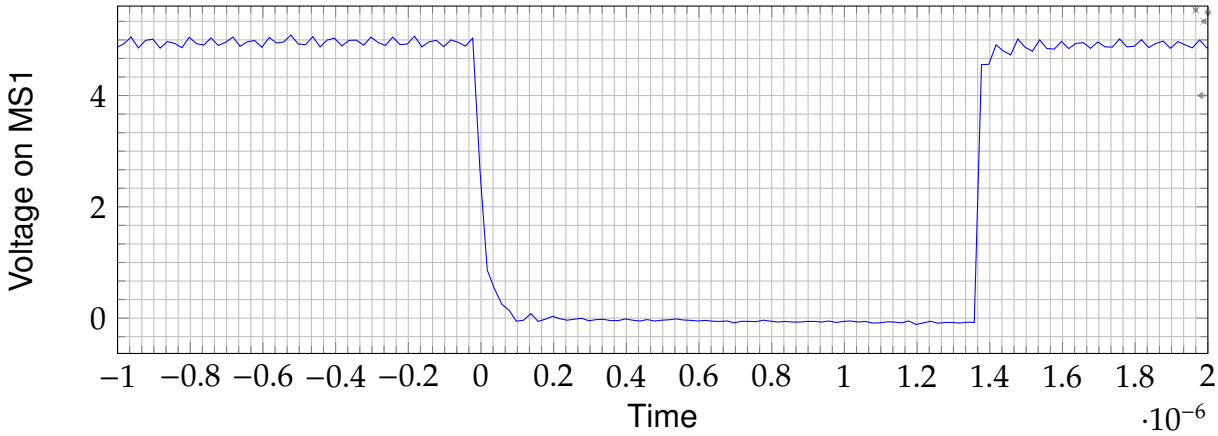

Figure 8: Measurement at around 1MHz bus clock on MS1

The ripple seen in Figure 8 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.

4.5 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 9: The case with installed backplane

4.6 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[10], or as it should be called now, TIA-[11]/EIA-[12]232.[5] Voltage-levels, as per TIA-/EIA-232 standard, are not practical to handle over short distances however, so other voltages are used on most interface chips and need to be converted.

4.6.1 16550 UART

The 16550 UART[13] 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 [14] buffer. It needs three address lines, and 8 data lines, which can be seen in Figure 10

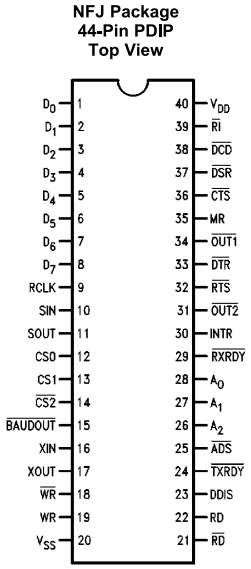

Figure 10: PC-16550D Pinout[4]

[10] RS... Recommended Standard
[11] TIA...Telecommunications Industry Association
[12] EIA.. Electronic Industries Alliance
[13] Uinversal Asynchronous Receiver and Transmitter
[14] First-In First-Out

In Figure 10 the most important lanes are the SIN and SOUT lanes, as they contain the serial data to and from the 16550 UART.

4.6.2 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[16].

4.6.3 Schematics

Based on the descriptions in the datasheets, the schematic in figure 11 was developed.

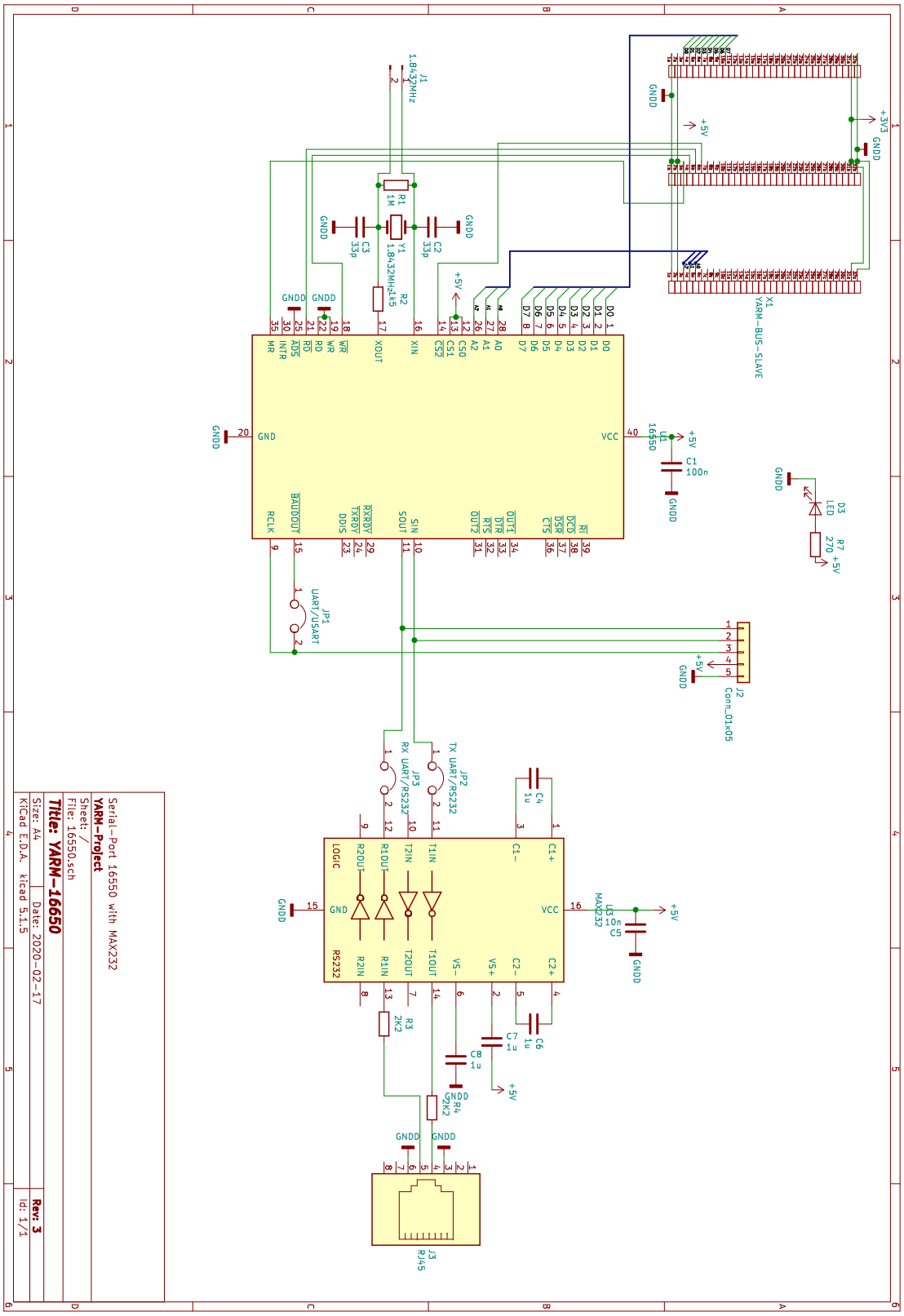

Figure 11: The schematic of the UART Module

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 [4]. 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 12. 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[4]. The final output of the 16550 UART can be used and measured via J2, as shown in Figure 13 . 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[16]. 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 14. C5 has the same functionality for the MAX-232 as the C1 has to the 16550-UART.

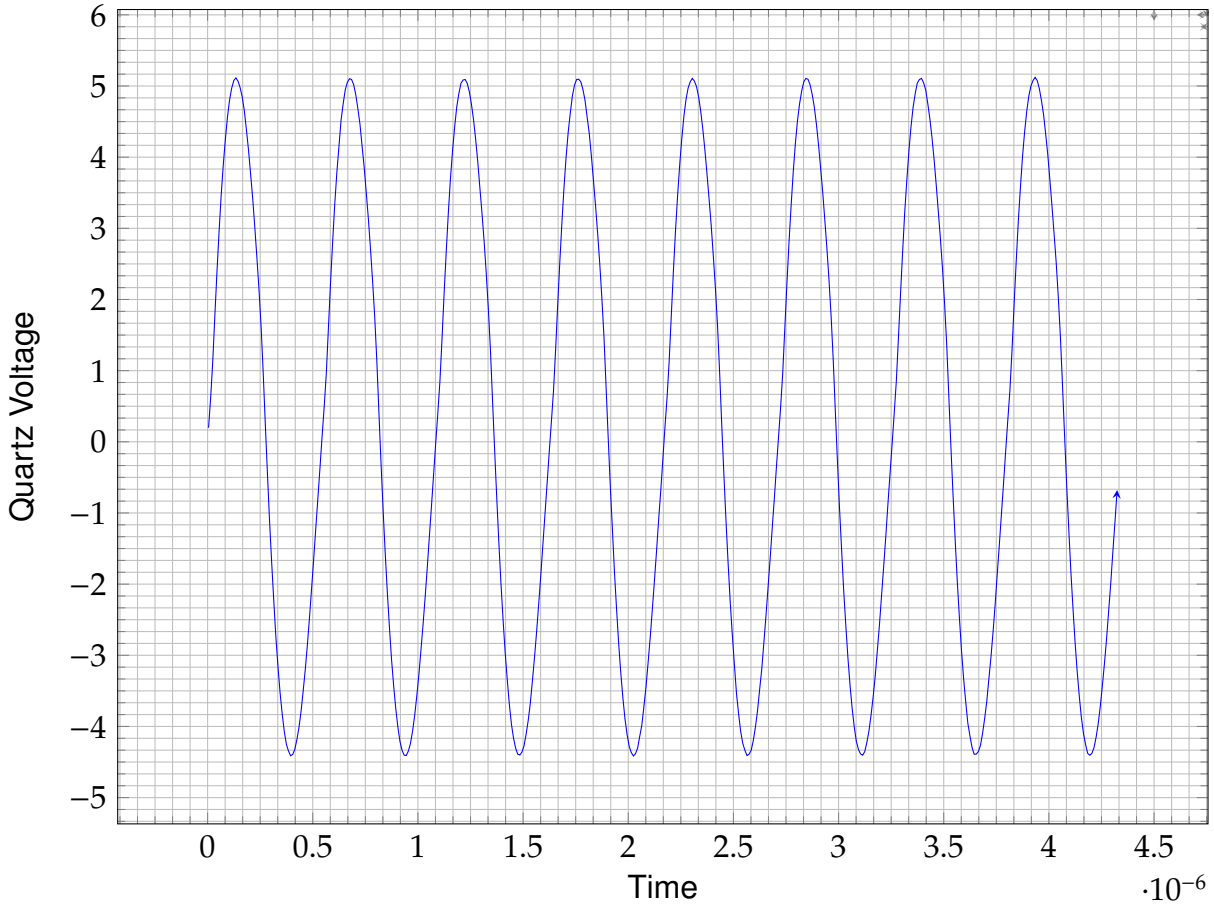

Figure 12: Measurement of the 1.8432 MHz Output on J1

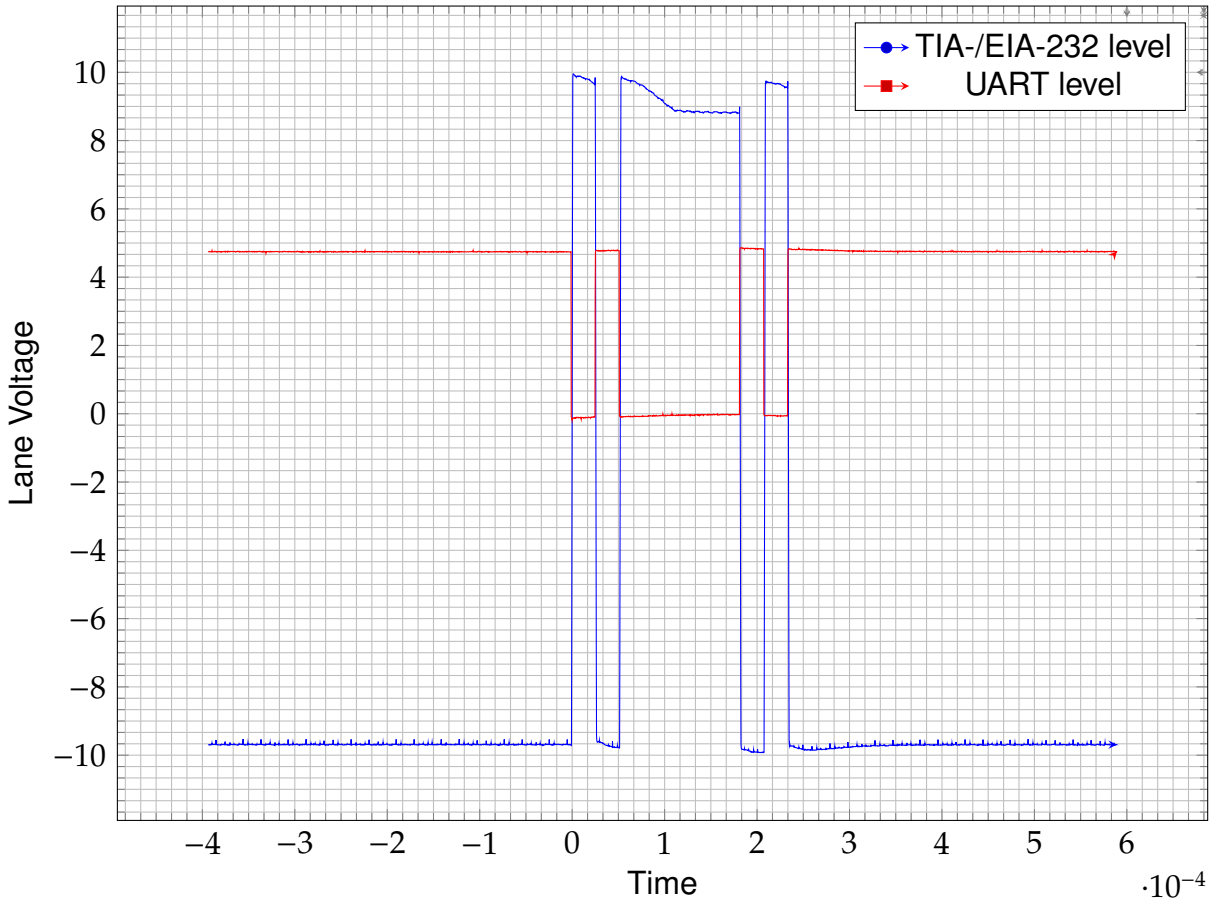

Figure 13: Measurement of a character transmission before and after MAX-232

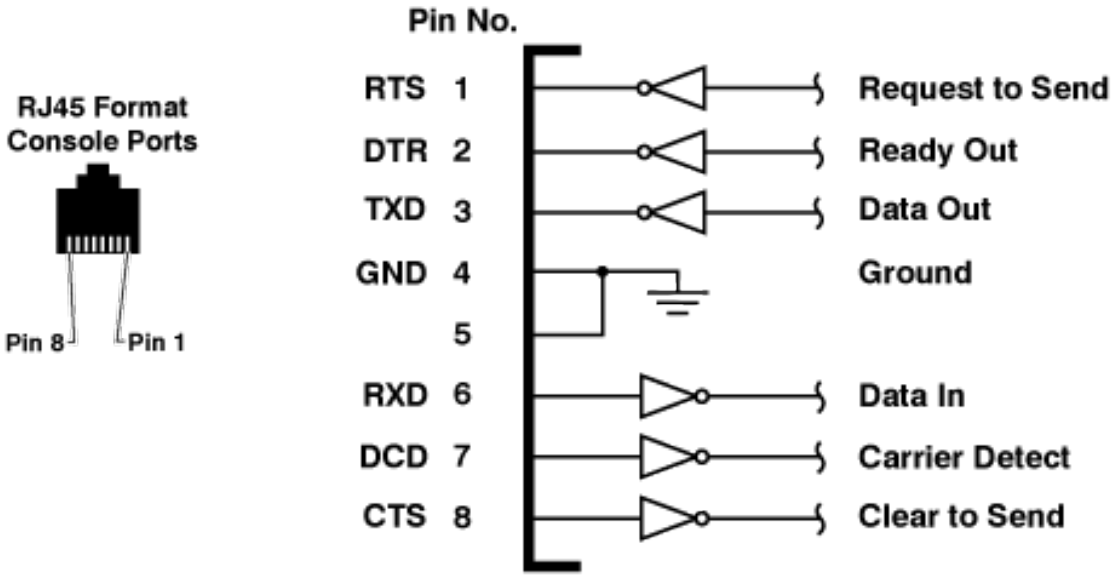

Figure 14: Pinout of the RJ-45 Plug; Src: `https://www.wti.com/`

4.6.4 Demonstration Software

To demonstrate the functionality and prove that the schematic has no underlying error, a program which regularly transmits a character as well as a simple echo program, which transmits all received characters are used. Both programs transmit 8 bit characters without parity at 38400 Baud. The output for program one can be seen in Figure 13 and the output for program two in Figure 15.

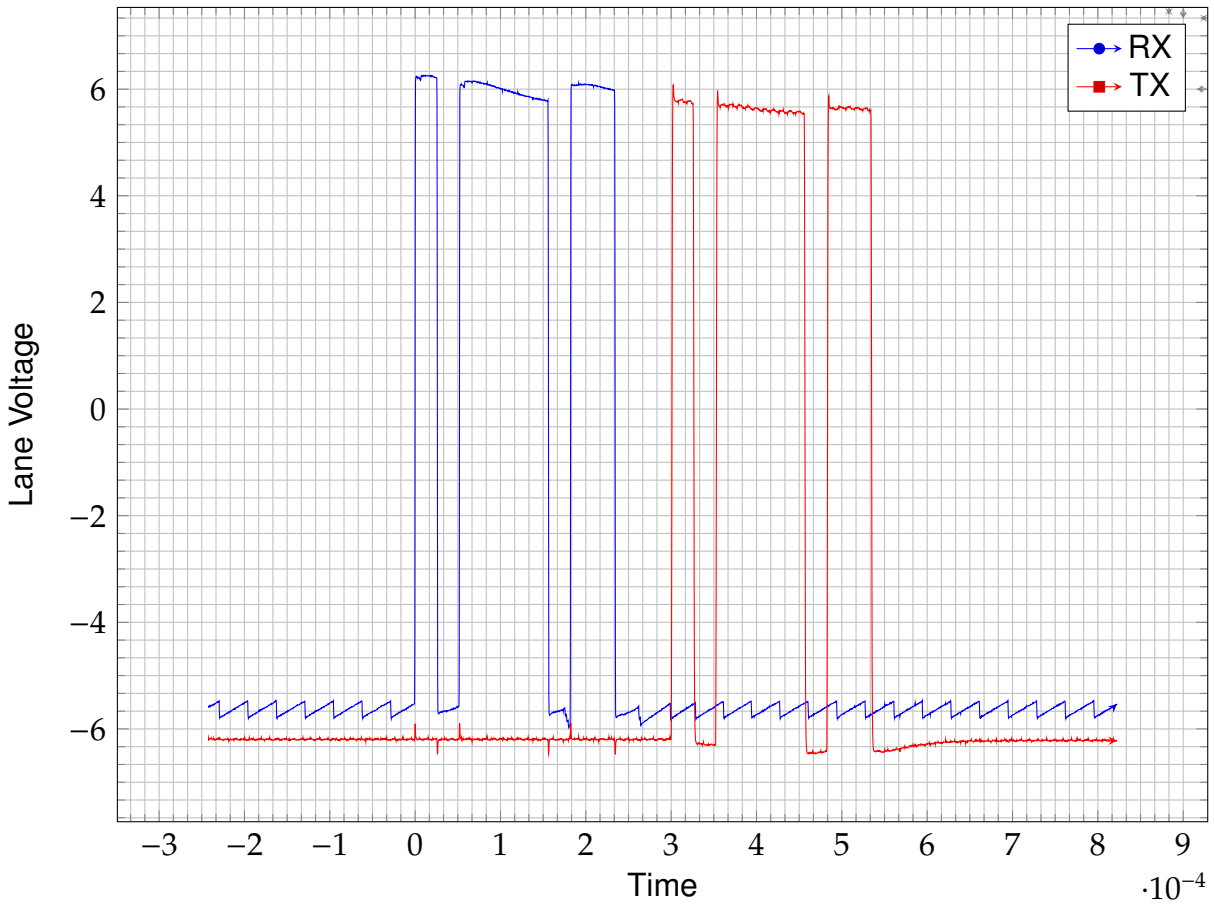

Figure 15: Measurement of a character echo

Transmit code The transmit code regularly transmits the letter capital A via the 16550 UART. Some initialisation is required beforehand. The functions shown in Listing 1 are the read and write routines for accessing the 16550 UART. These routines also apply to the echo code.

```
1 #define F_CPU 16000000UL
2
3 #include <stdint.h>
4 #include <util/delay.h>
5
6 #define BUS_HOLD_US 1
```

```
7
8  /* Shift values inside the PORTL Register */
9  #define WR_SHIFT 1
10 #define RD_SHIFT 2
11 #define MR_SHIFT 0
12 #define CS_SHIFT 3
13 #define CS_ADC_SHIFT 4
14
15 /* Registers in the 16550 UART */
16
17 #define UART_REG_DLLS   0
18 #define UART_REG_DLMS   1
19 #define UART_REG_TXRX   0
20 #define UART_REG_IER    1
21 #define UART_REG_IIR    2
22 #define UART_REG_LCR    3
23 #define UART_REG_MCR    4
24 #define UART_REG_LSR    5
25 #define UART_REG_MSR    6
26 #define UART_REG_SCR    7
27
28 void set_addr(uint8_t addr){
29
30     PORTK = addr;
31     return;
32 }
33
34 void write_to_16550(uint8_t addr, uint8_t data){
35
36
37     set_addr(addr);
38     DDRF = 0xFF;
39     PORTL &= ~(1<<WR_SHIFT);
40     PORTF = data;
41     PORTL &= ~(1<<CS_SHIFT);
42
43     _delay_us(BUS_HOLD_US);
44
45     PORTL |= 1<<CS_SHIFT;
46     set_addr(0x00);
47     PORTL |= 1<<WR_SHIFT;
48     PORTF = 0x00;
49     return;
50 }
51
52 uint8_t read_from_16550(uint8_t addr){
53
```

```
54      uint8_t data = 0x00;
55      set_addr(addr);
56      DDRF = 0x00;
57      PORTF = 0x00;
58      PORTL &= ~(1<<RD_SHIFT);
59      PORTL &= ~(1<<CS_SHIFT);
60      _delay_us(BUS_HOLD_US);
61      data = PINF;
62      PORTL |= 1<<CS_SHIFT;
63      set_addr(0x00);
64      PORTL |= 1<<RD_SHIFT;
65      DDRF = 0xFF;
66      PORTF = 0x00;
67      _delay_us(BUS_HOLD_US); /*Wait for the data and signal lanes to become
        stable*/
68      return data;
69  }
```

Listing 1: 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[4]. 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 2

```
1   int main(){
2
3       /* Disable interrupts during initialisation phase */
4       cli();
5
6       /* Setup Data Direction Registers and populate with sane default
7          values */
8       DDRF = 0xFF; /* Data Bus */
9       DDRK = 0xFF; /* Address Bus */
10      DDRL = 0xFF; /* Control Bus */
11      PORTF = 0x00;
12      PORTK = 0x00;
13      PORTL = 0x00;
14
15      /* Cleanly reset the 16550 uart */
16      PORTL |= (1<<WR_SHIFT);
17      PORTL |= (1<<RD_SHIFT);
18      PORTL |= (1<<CS_SHIFT);
```

```
    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 2: 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 16

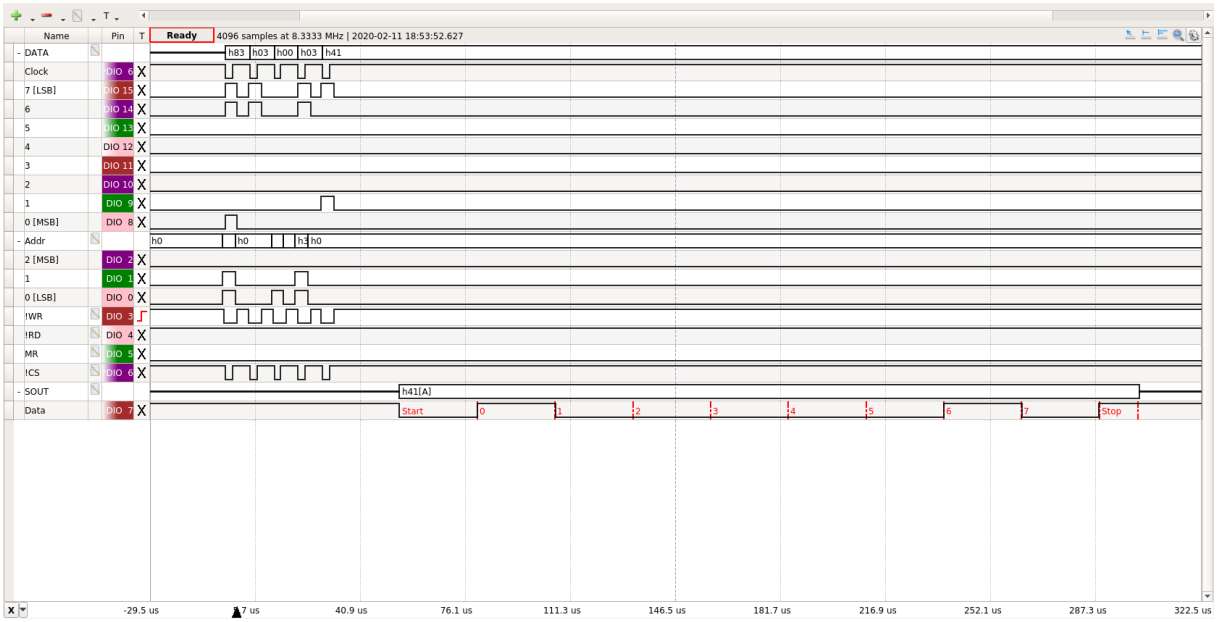

Figure 16: Transmission of character A via the 16550 UART

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 15.

The initialisation is practically the same for the echo code as well as the read and write routines in Listing 1.

```
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 3: 16550 character echo

4.6.5 Final Module

The final module can be seen in Figure 17 with the pc16550 UART in the center and the MAX-232 above.

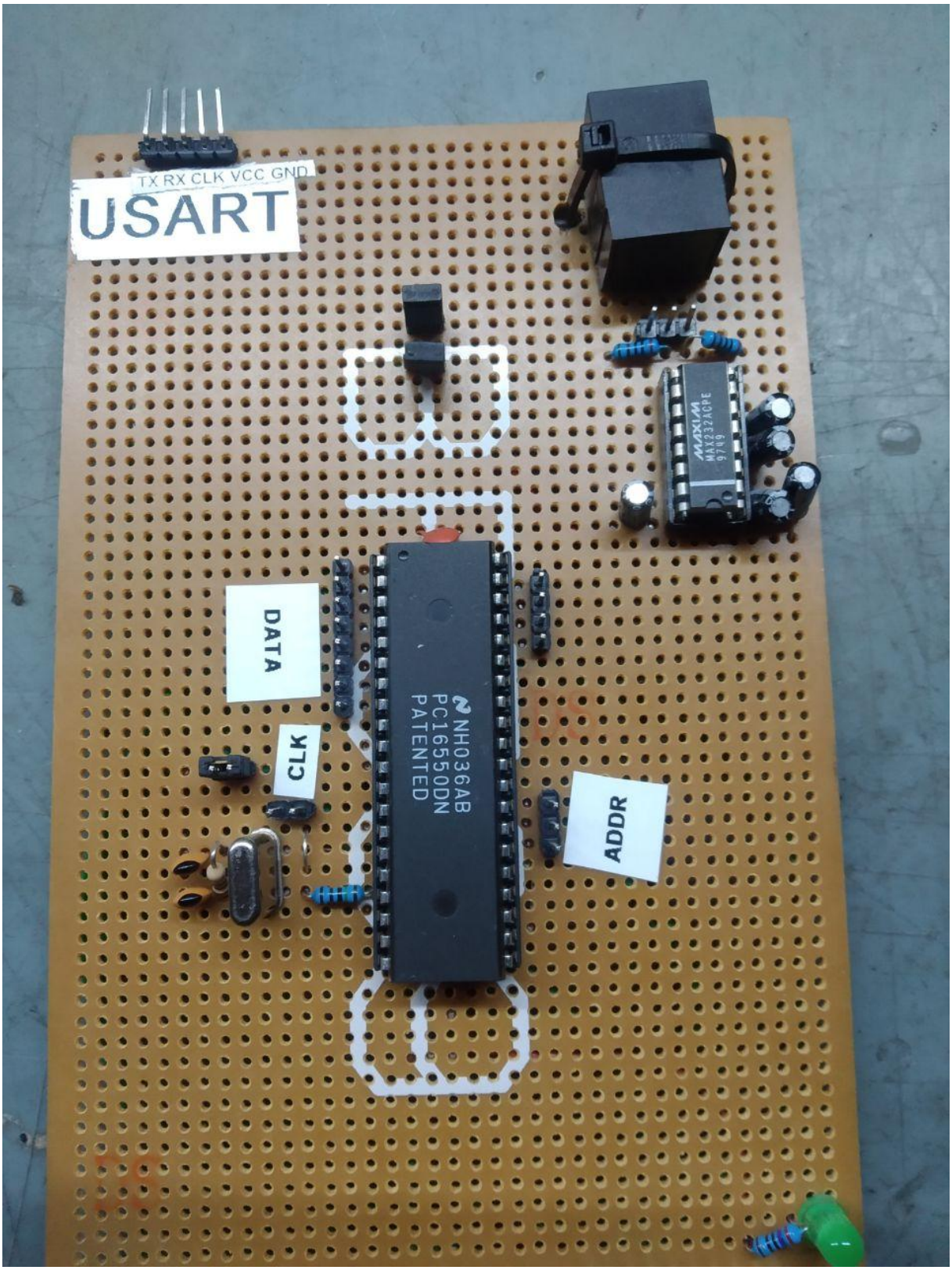

Figure 17: The final uart module with the pc16550 uart in the center

4.7 Audio Digital-Analog-Converter

A digital to analog converter takes a digital number and converts it to a analog signal. The output of such a 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.

4.7.1 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 [17], and which (should be [15]) is monotonic over the entire D/A Conversion Range. The TLC-7528 is the only component chosen, where availability is 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[17], which made its use obvious. The TLC-7528 was the only IC available as DIP [16], of which the pinout can be seen in Figure 18.

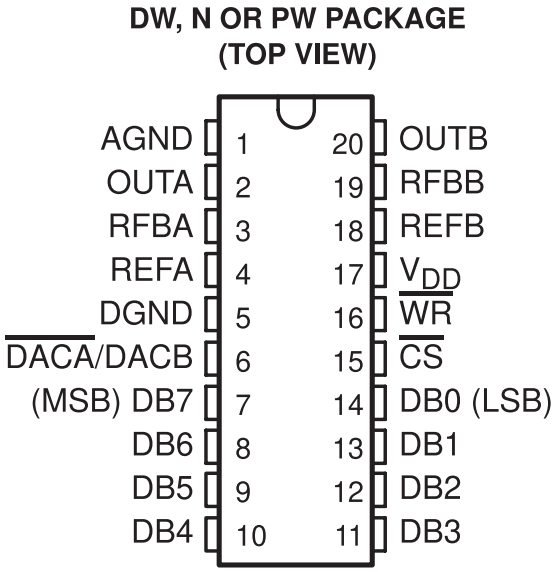

Figure 18: TLC-7528 Pinout[17]

[15] See Figure 21
[16] DIP... Dual Inline Package

4.7.2 IDT7201 CMOS FIFO Buffer

The IDT7201 is an asychronous CMOS FIFO. That 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 19, and can store up to 256 words[18]. 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.

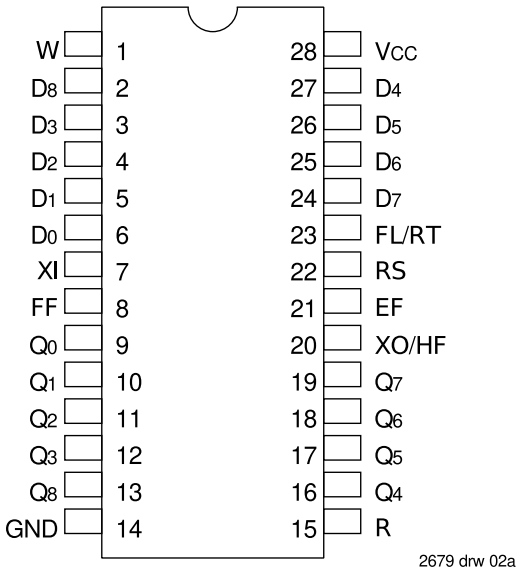

Figure 19: IDT-7201 Pinout[18]

4.7.3 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[17] and seen in Figure 20. After the result seen in Figure 21 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.

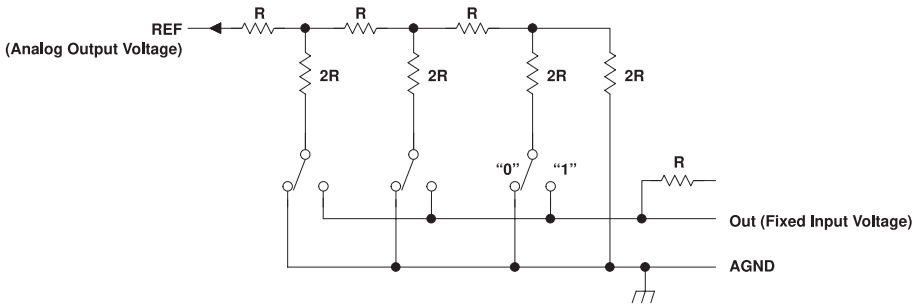

Figure 11. Voltage-Mode Operation

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

$V_O = V_I (D/256)$

Figure 20: TLC-7528 in voltage modet[17]

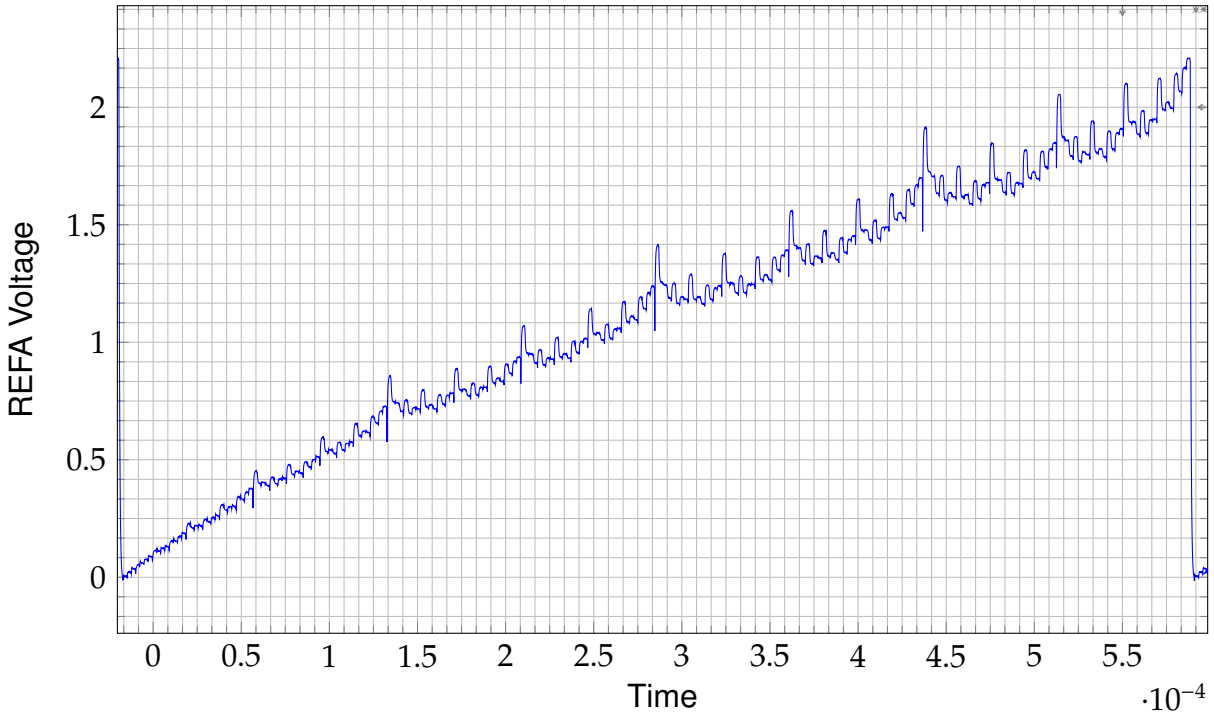

Figure 21: Measurement of a generated SAW signal via the TLC7528

4.7.4 Schematics

Based on the descriptions in the datasheets the schematic in figure 22 was developed.

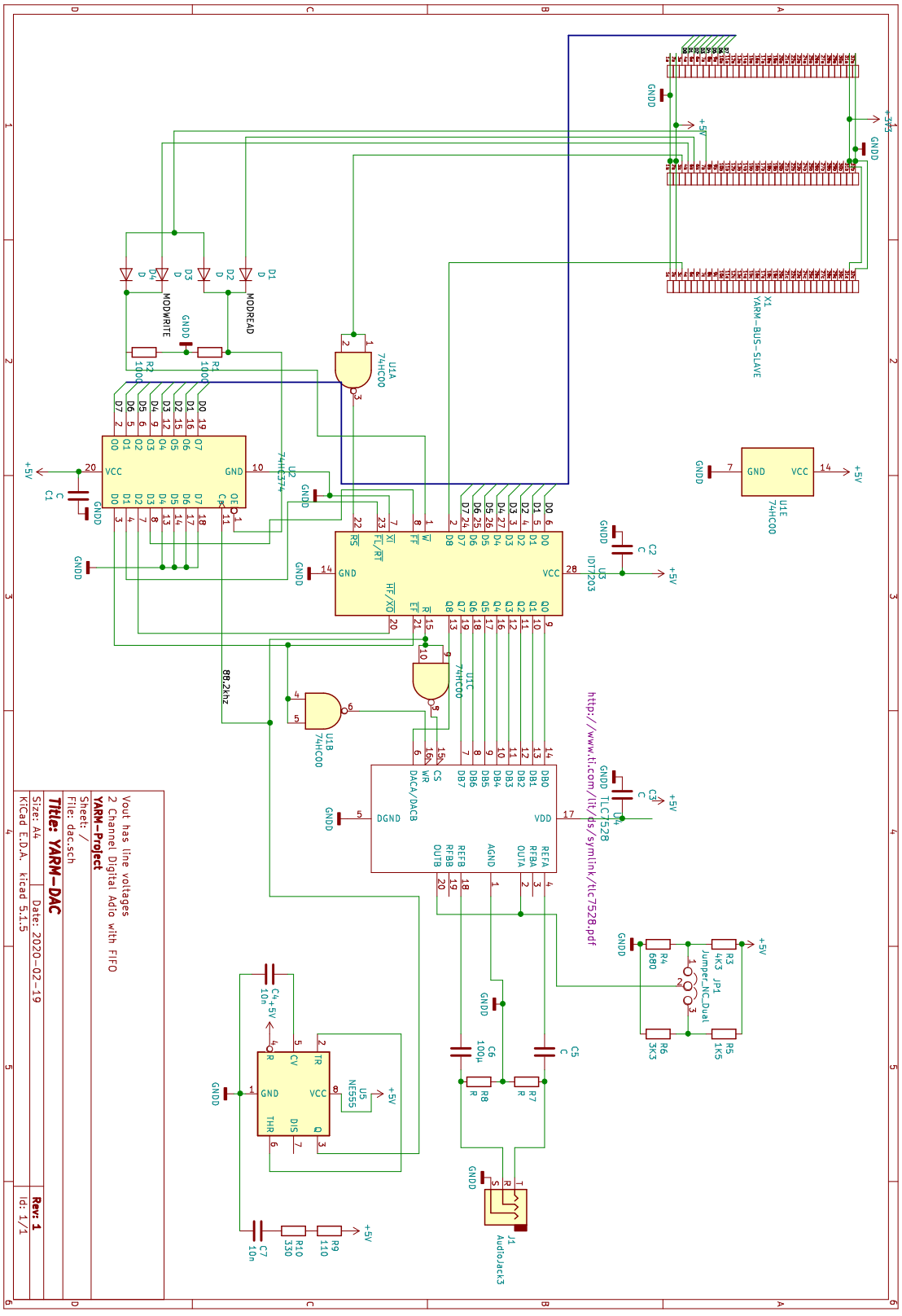

Figure 22: The schematic of the DAC Module

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 [17] 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[20], 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[18], as 44.1kHz is the standard samling rate of CD-Audio[21] and 2 channels need to be sampled. 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 Figure 20, 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.[22] 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

$f_C = \frac{1}{2\pi RC} = \frac{1}{2\times\pi\times 10K\Omega\times 100\mu F} = 0.159154943Hz$

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.

[17] 74HC374[19]
[18] Because we have 2 output channels

Functional Description On a read of the module the $\neg MS2$ line goes low as well as the $\neg RD$ line, which combined by the as OR gate used diodes D1/D2 and resistor R1 forms the MODREAD signal. The modread signal is passed on to the $\neg OE$ pin of the D-Flip-Flop which writes the FIFO status bits onto the data bus.

On write the same or gate is formed with diodes D3/D4 and resistor R2 which combines signals $\neg MS2$ and $\neg WR$ into MODWRITE. MODWRITE is then fed into the $\neg W$ pin of the FIFO which stores the data on the data bus into it's internal buffer.

The FIFO is read with the clock generated by the NE555 (see the NE555 paragraph below) which puts the data onto the bus between FIFO and DAC. The DAC reads the data into its internal buffer after the FIFO has put it onto the DATA lanes due to the inversion by the B part of the 74HC00 and the output beeing mapped to the $\neg CS$ pin of the DAC. When the FIFO is empty it produces nonsense as output, to mittigate errors resulting from this the $\neg EF$ output of the FIFO is inverted by the C part of the 74HC00 and put onto the $\neg WR$ pin of the DAC.

The maximum amplitude can be selected by jumper JP1. Generated waveforms by the DAC are filtered against a DC offset via the highpasses built by C5/R7 and C6/R8 respectively. The resulting waveform can be measured on audio jack J1.

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 [19] with an internal PLL is used, thus the required quartz can no longer be sourced via conventional electronic resellers.

4.7.5 DAC Module Read

On a read the status bits of the FIFO, which has been latched into the 74HC374 D-Flip-Flop, are written onto the Data bus. Table 5 defines the layout of these status bits on the data bus.

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

Bit position	Usage
0	FIFO empty flag
1	Not used (originally FIFO low)
2	FIFO half full
3	FIFO full
4	Not used
5	Not used
6	Not used
7	Not used

Table 5: The layout of the Data Bus on DAC read

4.7.6 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 21 , however the signal was put into the FIFO and not the DAC directly. The resulting saw wave can be seen in figure 23 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.

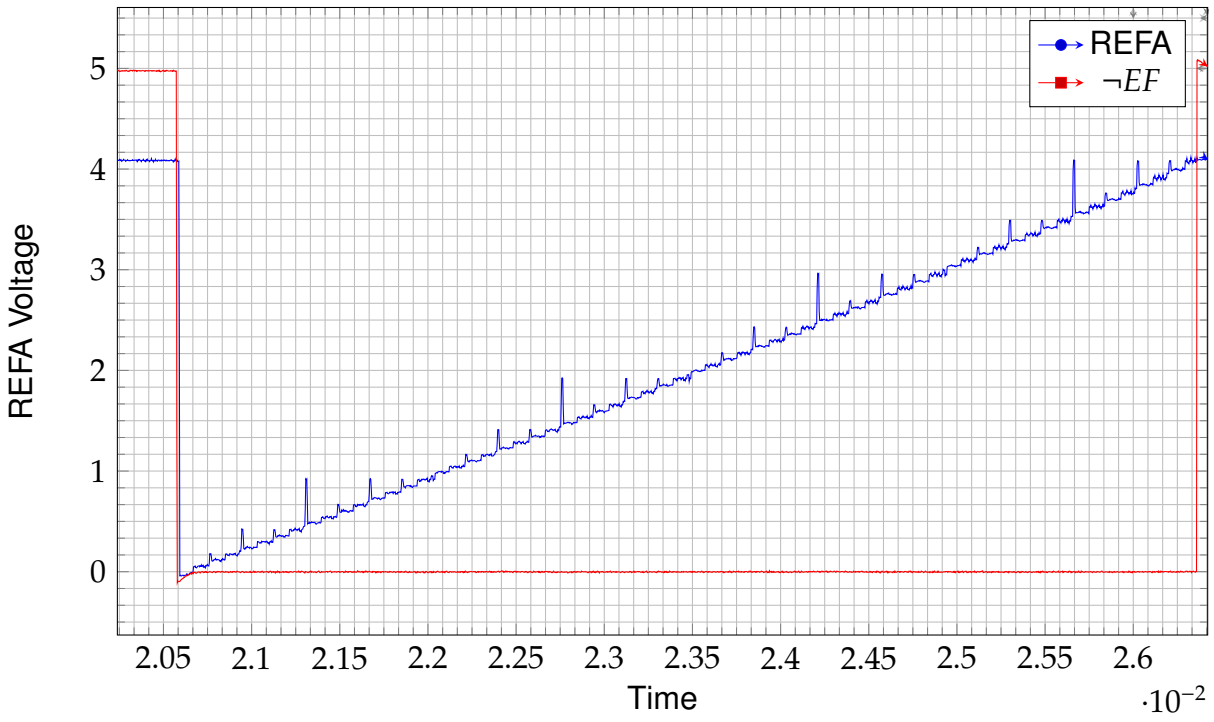

Figure 23: Measurement of a generated SAW signal with the FIFO Empty flag

The time difference betwen a store and complete write cycle can be seen in figure 25, while figure 24 shows the transmission between dac and fifo in more detail.

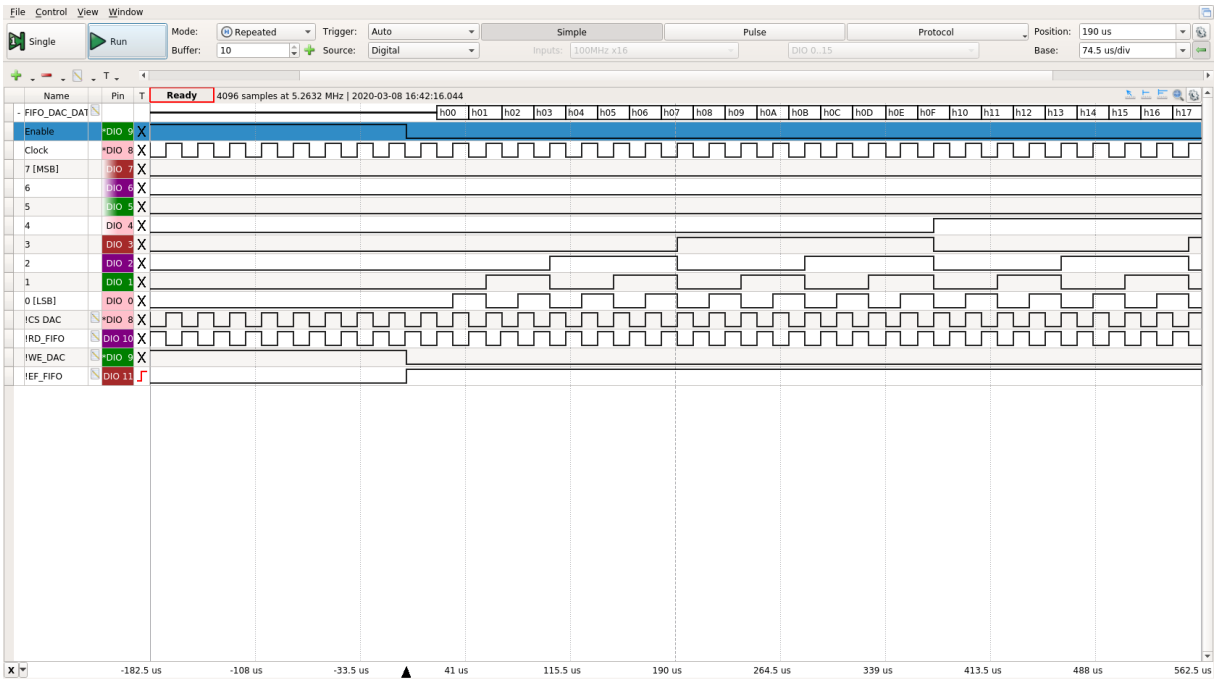

Figure 24: A transmission between the FIFO and the DAC

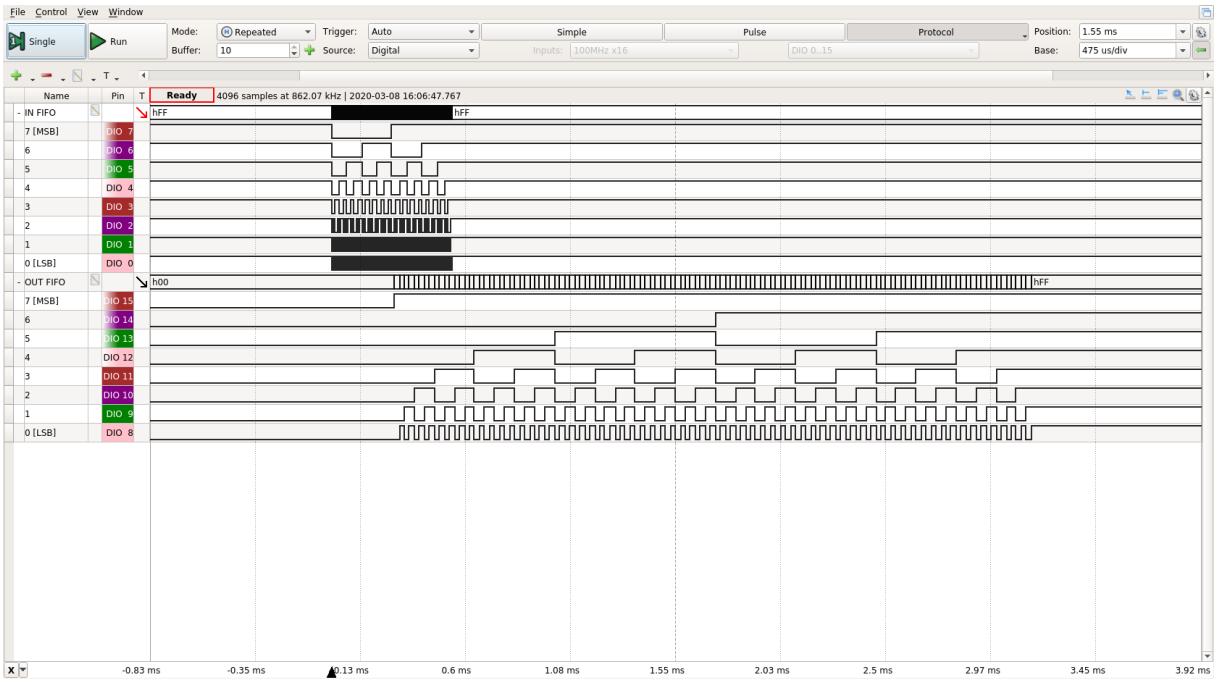

Figure 25: A fifo store operation in contrast to the load operation

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 2.

```
1 int routine(){
2
```

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

    write_to_dac(0x00, 0x00);

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

Listing 4: 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 5. How the data is transmitted to the FIFO can be seen in listing 6 and figure 26, and the resulting sine on both output channels can be seen in figure 27.

```
    /* 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 5: 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 method shown in listing 5 is not fast due to the lack of a floating point unit on the AVR. [14]

```
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;
```

```
12 }
```

Listing 6: DAC Sine Generation

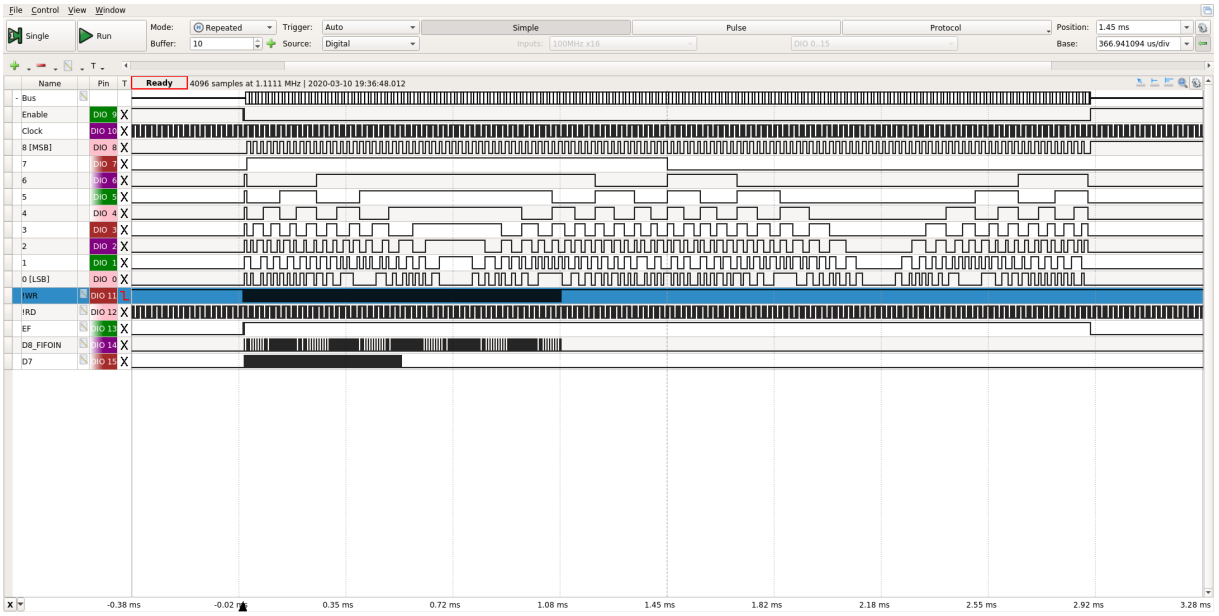

Figure 26: Storage and retrieval of a sine to and from the FIFO

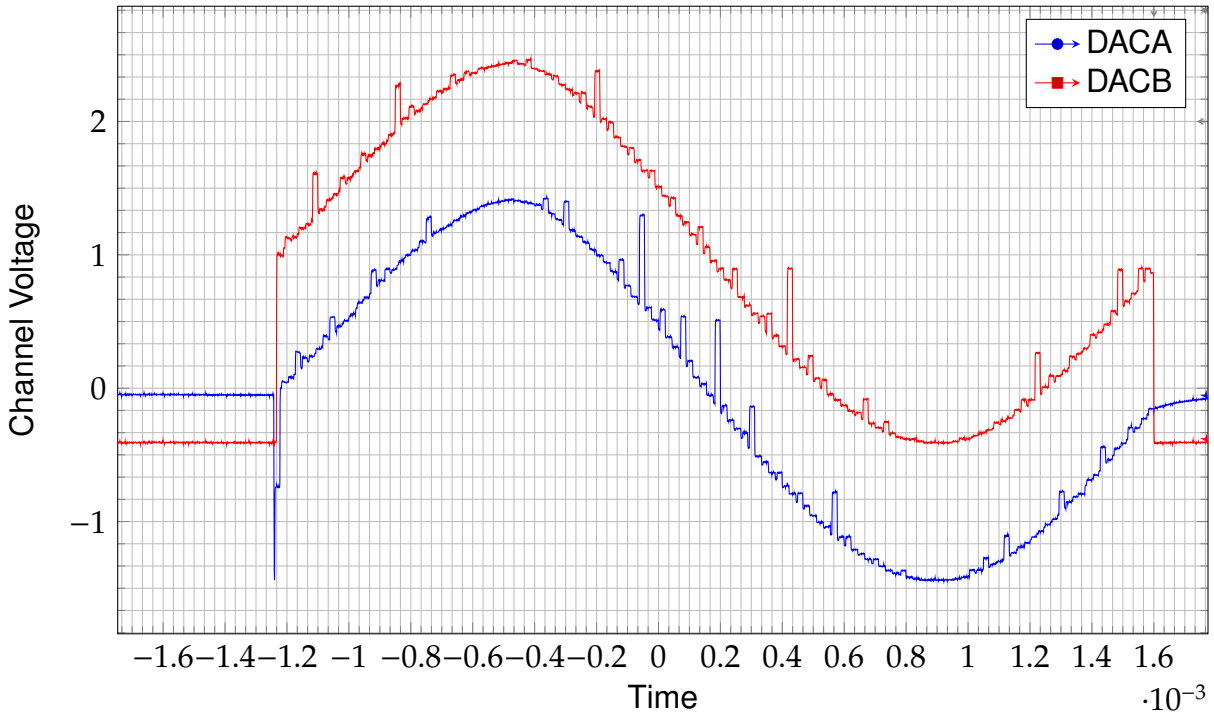

Figure 27: Measuremet of the generated sine from the sine LUT on DACA and DACB

4.7.7 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 18. 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 is 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.

4.7.8 Final Module

The final module can be seen in figure 28 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.

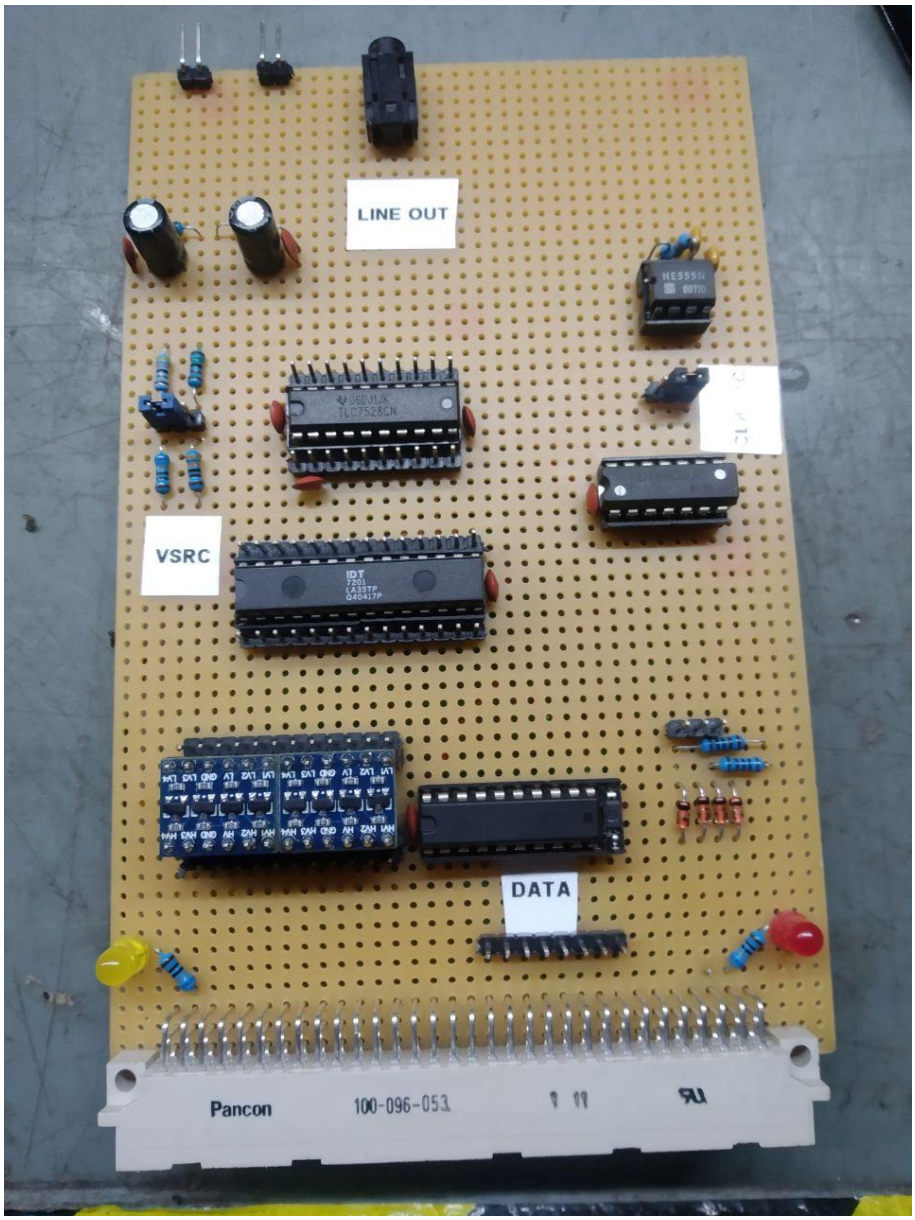

Figure 28: The final DAC module

4.8 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 I/O 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 are not well documented. Their functionality is tested and verified in both directions, which can be seen in figures 29 and 30. The schematic was determined through measurements with a multimeter, and the schematic in Figure 31 shows similar resistor values in the same configuration [23].

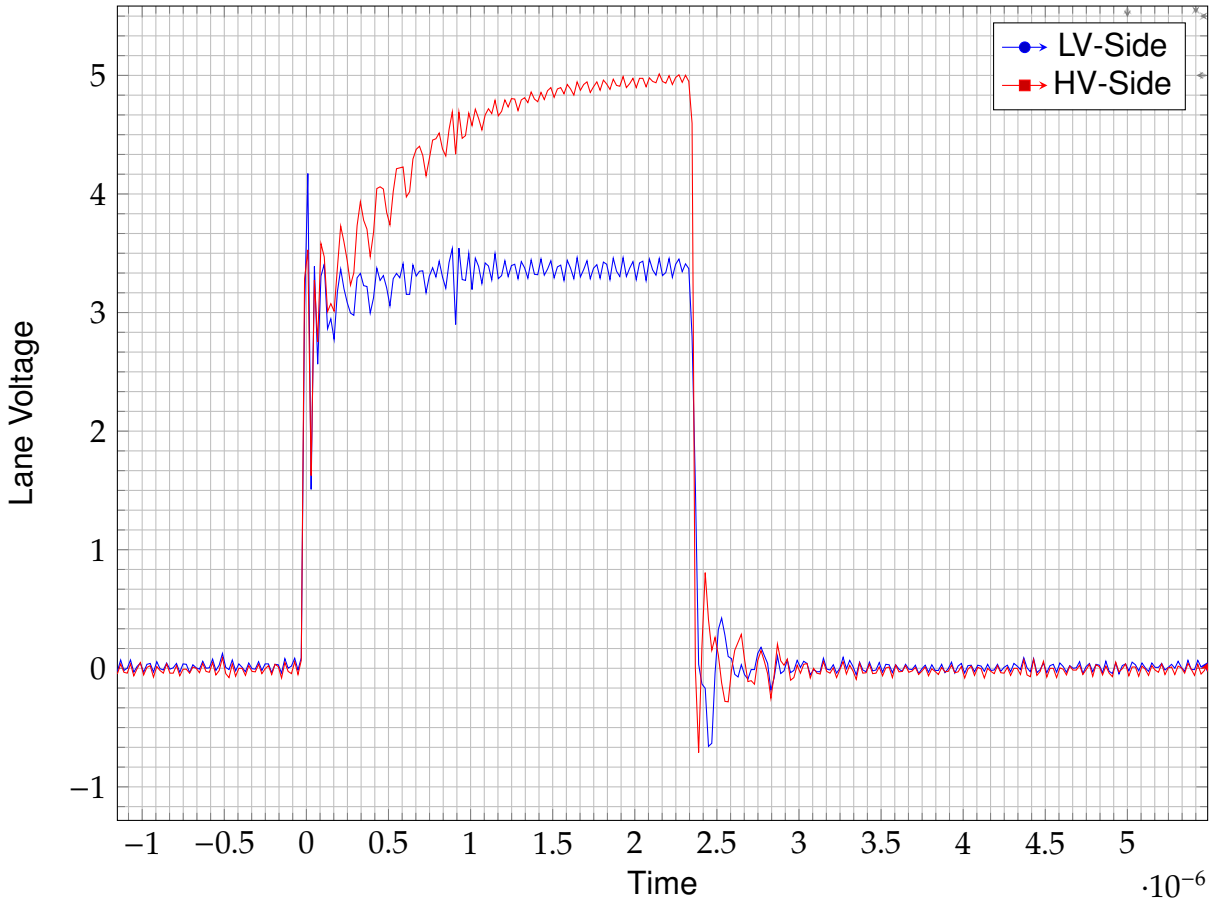

Figure 29: 3.3V to 5V conversion using the level shifter

The in Figure 29 shown output on the HV side corresponds with the schematics in Figure 31, where one can see, that the resistor R2 is loading the bus capacitance to a 5V high state.

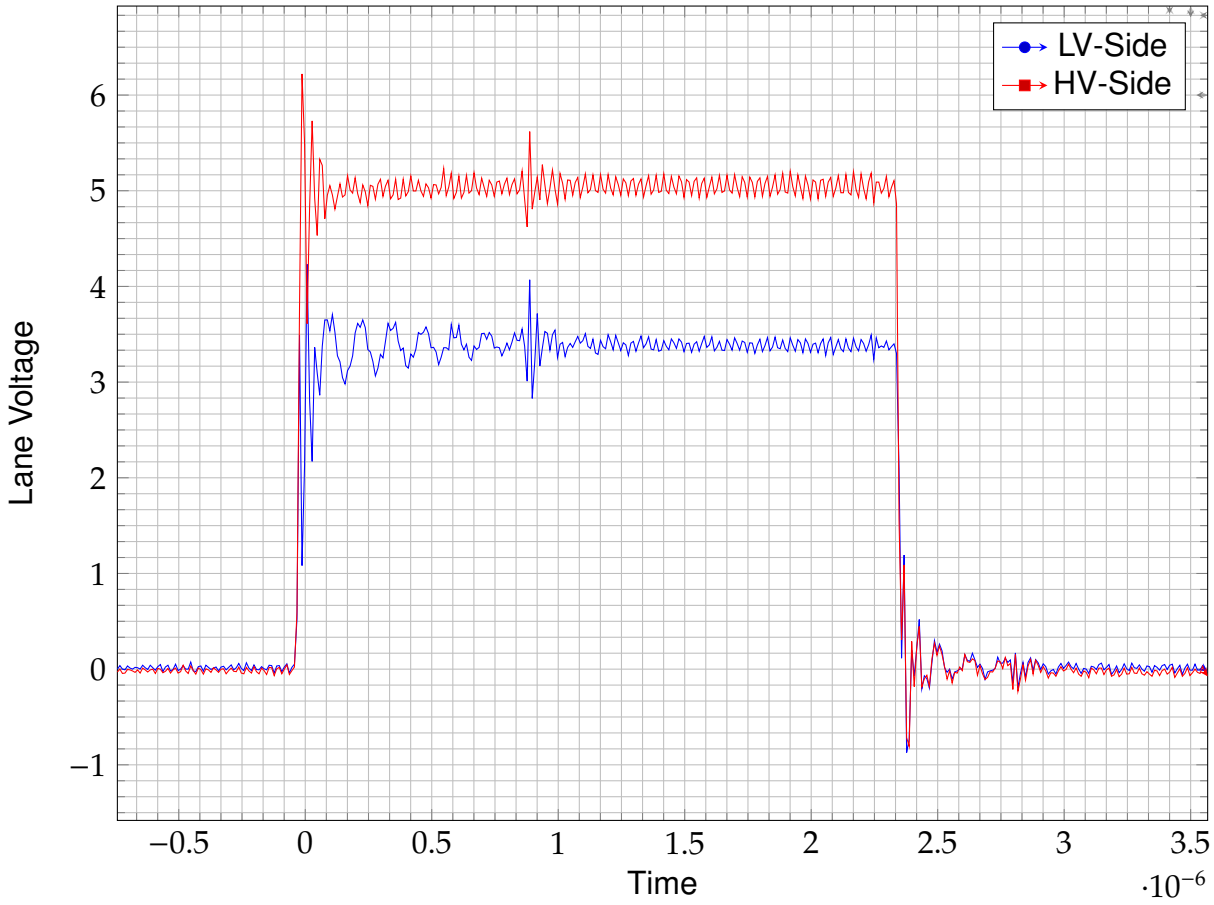

Figure 30: 5V to 3.3V conversion using the level shifter

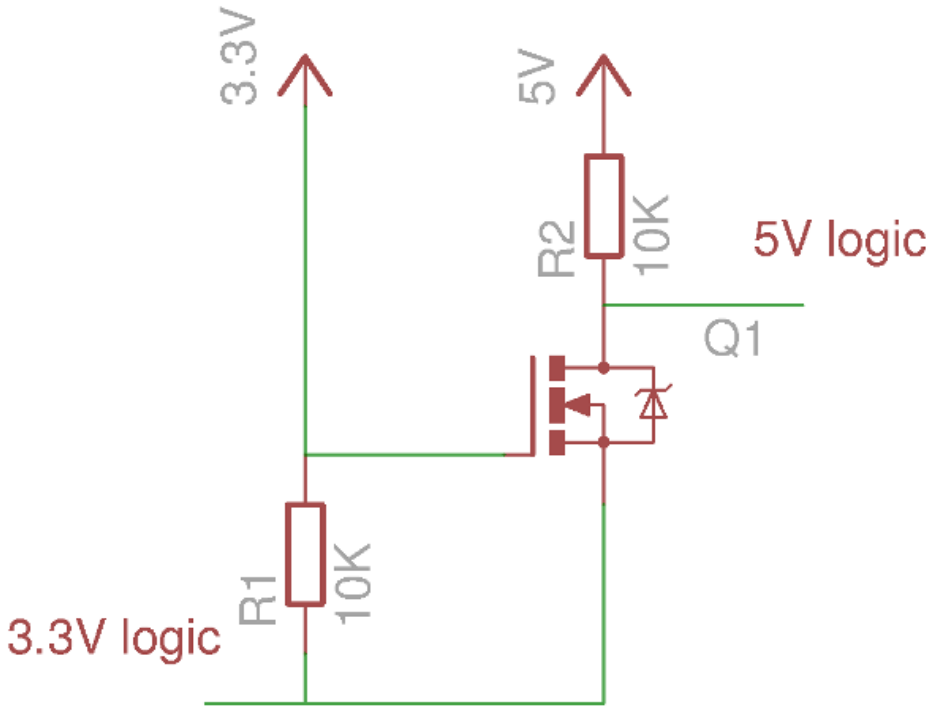

Figure 31: The internal schematics of the level shifter[23]

4.8.1 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 32. 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. [24].

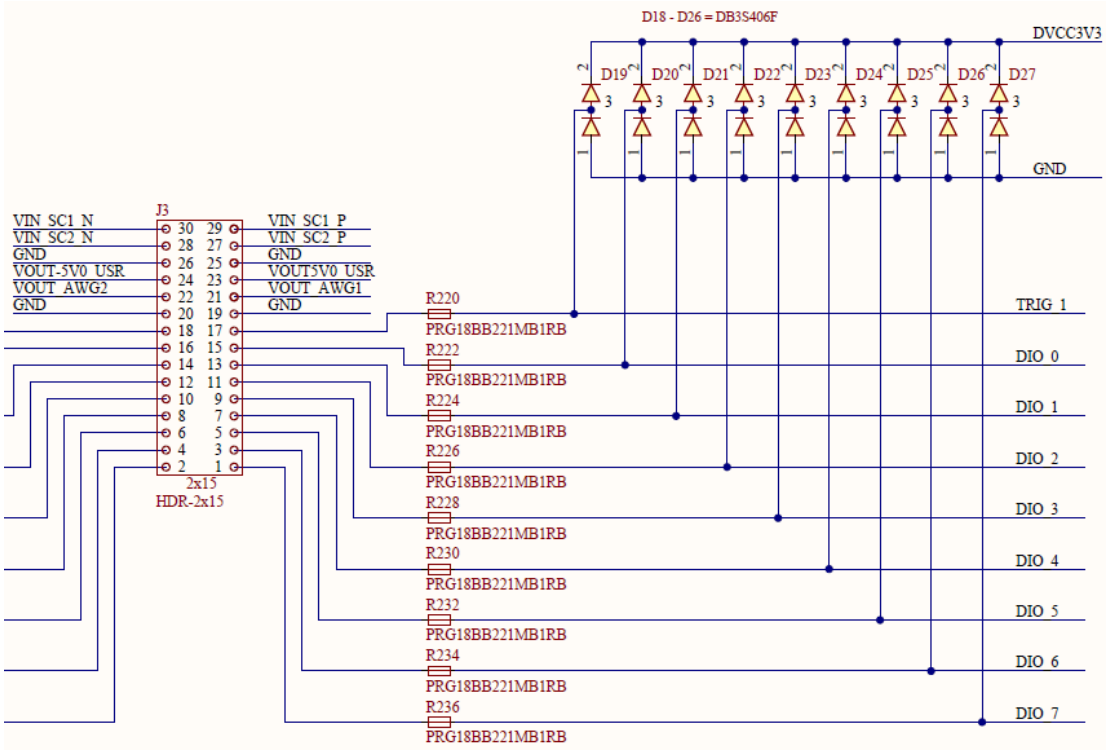

Figure 32: The internal clamping diodes of the Analog Discovery 2[15]

4.8.2 Final Module

The final module can be seen in figure 33 without the FPGA attached. The blue modules below are the level shifters.

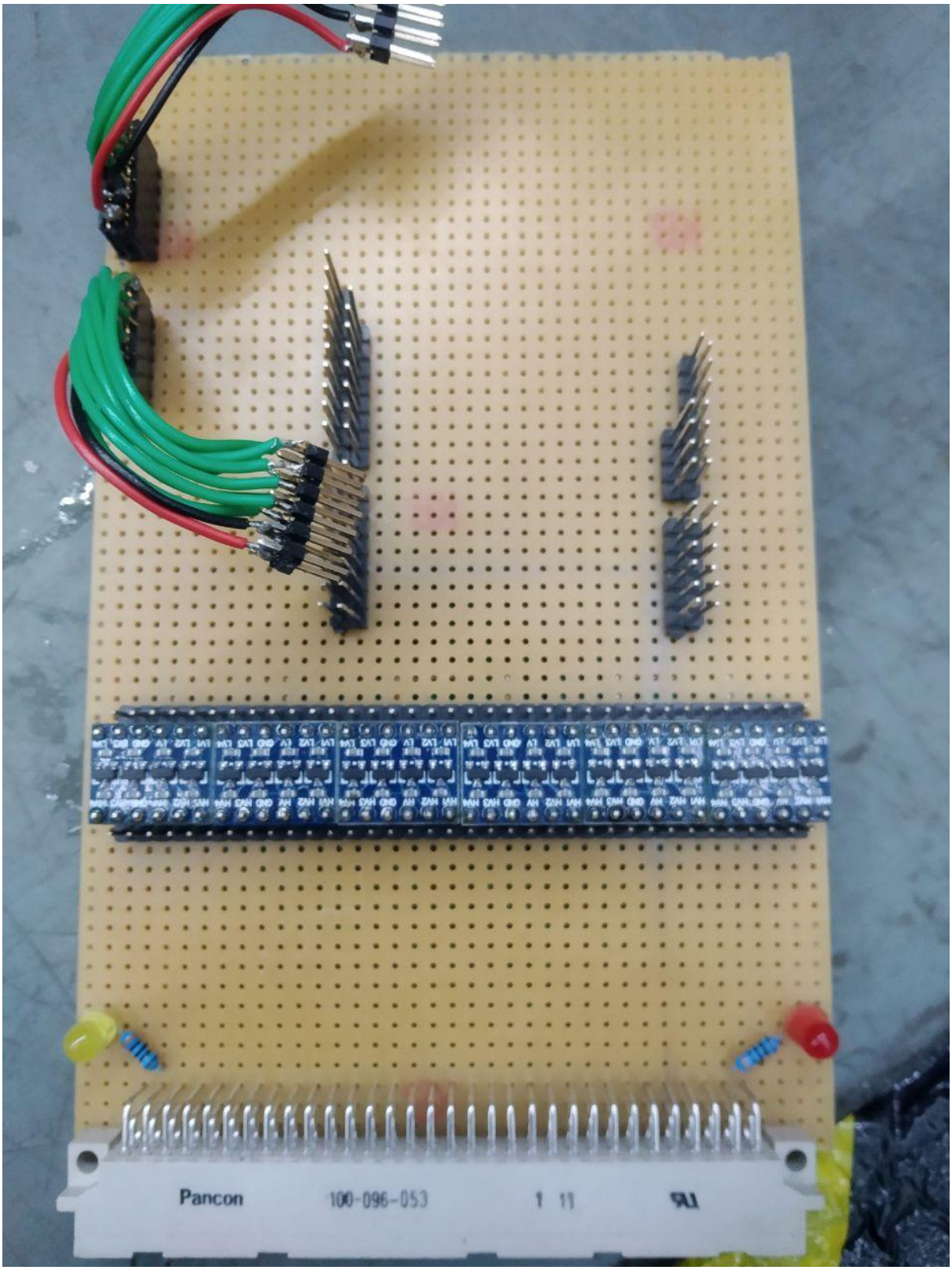

Figure 33: The final FPGA interface module with the level shifters

5 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.

5.1 General Implementation details

5.1.1 General definitions and pinout of the AVR

Like the examples seen before, 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 7

```
1  /* Copyright (C) 2020 tyrolyean
2   *
3   * This program is free software: you can redistribute it and/or modify
4   * it under the terms of the GNU General Public License as published by
5   * the Free Software Foundation, either version 3 of the License, or
6   * (at your option) any later version.
7   *
8   * This program is distributed in the hope that it will be useful,
9   * but WITHOUT ANY WARRANTY; without even the implied warranty of
10  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
11  * GNU General Public License for more details.
12  *
13  * You should have received a copy of the GNU General Public License
14  * along with this program.  If not, see <http://www.gnu.org/licenses/>.
15  */
16
17 #ifndef _AVR_H_TEXT
18 #define _AVR_H_TEXT
19
20
21
22 #define F_CPU 16000000UL
23 #include <avr/io.h>
24
25 /* Shift values for the peripherials on the control bus PORTL */
26
27 #define MR_SHIFT      0
28 #define WR_SHIFT      1
```

```
29 #define RD_SHIFT      2
30 #define CS_UART_SHIFT 3
31 #define CS_DAC_SHIFT  4
32
33 #define ADDR_REG      PORTK
34 #define DATA_REG      PORTF
35 #define CTRL_REG      PORTL
36
37 #define ADDR_DDR_REG  DDRK
38 #define DATA_DDR_REG  DDRF
39 #define CTRL_DDR_REG  DDRL
40
41 /* Included here to prevent accidental redefinition of F_CPU */
42 #include <util/delay.h>
43
44 /* Time it takes for the bus lanes to become stable for read and write
      access */
45 #define BUS_HOLD_US 1
46
47 void set_addr(uint8_t addr);
48
49 #endif
```

Listing 7: The avr.h header file

The in listing 7 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 macro 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[20] to a level which effects regular operation.

5.1.2 Read and Write routines

The set_addr function is the same as in the UART example code in listing 1 and has therefore been omitted, execept for its definiton in the avr.h file in listing 7. 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.

[20] BER...Bit Error Ratio

5.1.3 UART and DAC update polling

The AVR constantly polls the DAC and UART modules for updates as can be seen in listing 8. The routine_MODULE functions poll their respective modules for updates as can be seen in listings 9 and 10. 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.

```
1
2  int routine(){
3      routine_dac();
4      routine_uart();
5      routine_game();
6          return 0;
```

Listing 8: The routine function looped by the main

```
1  void routine_uart(){
2
3      uint8_t received = read_from_uart(UART_REG_LSR);
4      if(received & 0x01){
5          received = read_from_uart(UART_REG_TXRX);
6          ingest_user_char(received);
7          if(received == '\r'){
8              writechar_16550('\n');
9          }
10         writechar_16550(received); /* Echo back */
11     }
12
13     return;
14 }
```

Listing 9: The routine function for the UART

```
1  void routine_dac(){
2
3      uint8_t received = read_from_dac(0x00);
4      if(!(received & (0x01<<0))){
5          feed_dac();
6      }
7      return;
8  }
```

Listing 10: The routine function for the DAC

5.1.4 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 infinite loop. This loop can be seen in listing 8 and in figure 34.

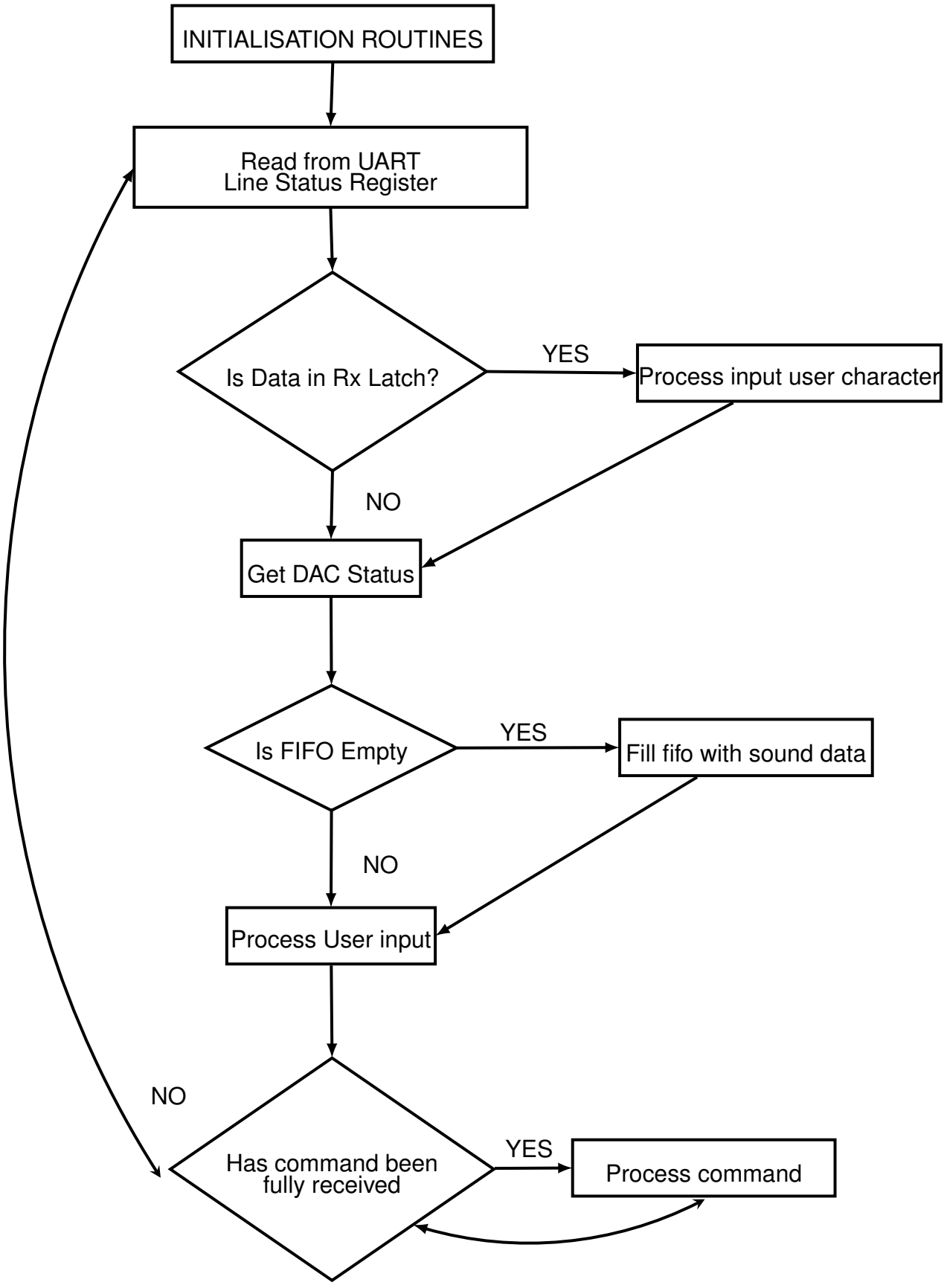

Figure 34: A Flow-Chart of the program execution path

5.2 DAC sound generation

5.2.1 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 exampl: 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[14] which results in 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 during runtime. In order to do that it has 6 modes in which it can operate, as can be seen in Listing 11:

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 11:

- 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

```
/* 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 11: The DAC operation modes

```
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 12: The DAC waveform generation code

5.2.2 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 [21] 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 13, and the sound update function, which actually updates the current tone and is responsible for playing a track in listing 14. The output of an example track can be seen in figures 35 and 36.

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

Listing 13: 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
```

[21] ISR...Interrupt Service Routine

```
 */
void update_sound(){

    static uint16_t audio_time = 0;
    static size_t tone_pointer = 0x00;
    static struct tone_t current_tone = {DAC_MODE_SILENT, 0,0};
    if(current_track == NULL){
        /* ABORT */
        audio_time = 0x00;
        return;
    }
    audio_time++;
    static const struct tone_t * old_track = NULL;

    if(audio_time >= current_tone.length ||
        current_track != old_track){

        if(old_track != current_track){
            tone_pointer = 0;
            audio_time = 0x00;
            old_track = current_track;
        }
        memcpy_P(&current_tone,&(current_track[tone_pointer]),
            sizeof(current_tone));

        if(current_tone.length == 0){
            tone_pointer = 0;
            memcpy_P(&current_tone,&(current_track[tone_pointer]),
                sizeof(current_tone));

        }

        dac_mode = current_tone.waveform;
        dac_frequency_deviation = current_tone.frequency_deviation +
            global_frequency_offset;
        audio_time = 0x00;
        tone_pointer++;
    }
    return;
}
```

Listing 14: The sound update function

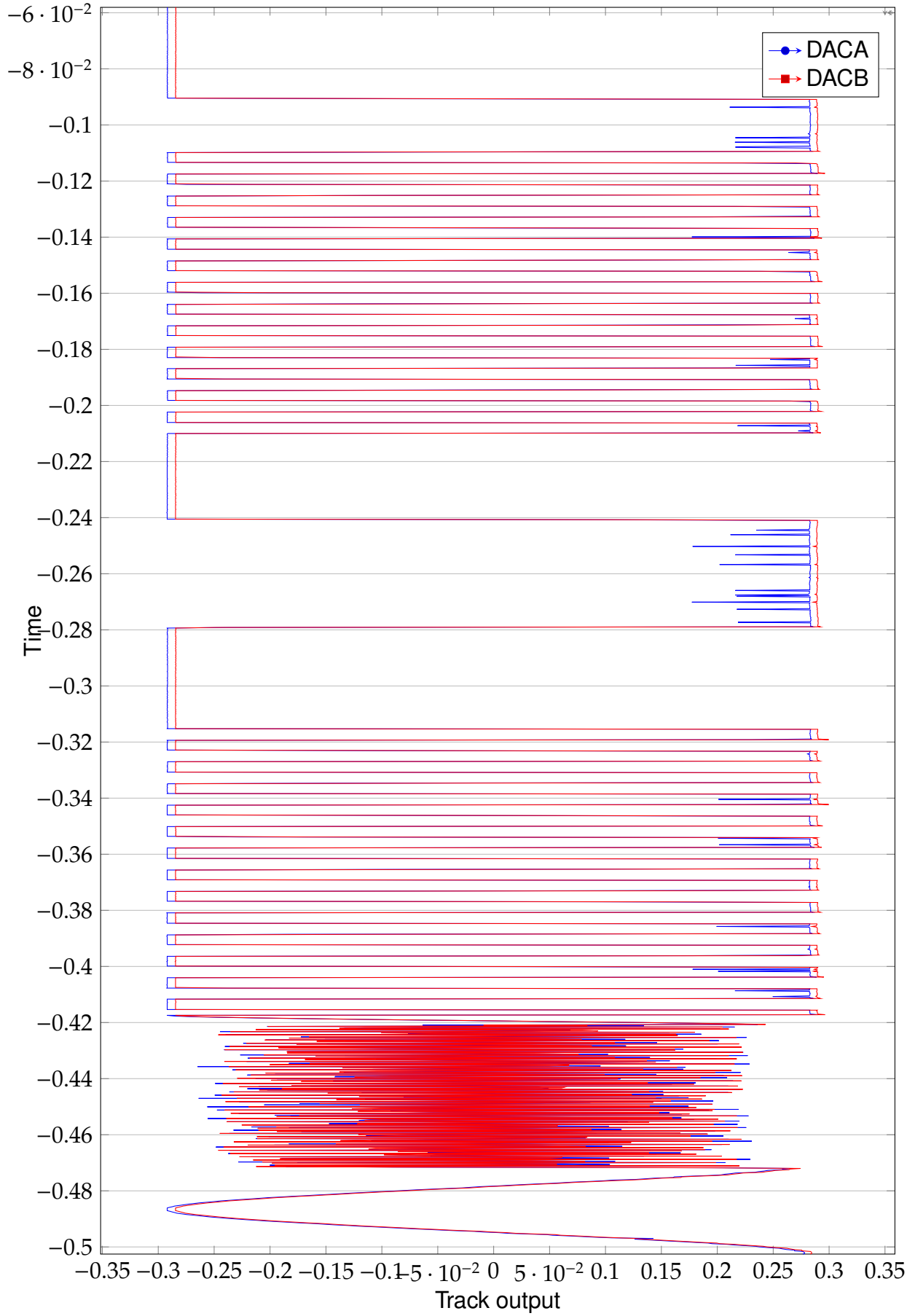

Figure 35: The output of an example track part 1

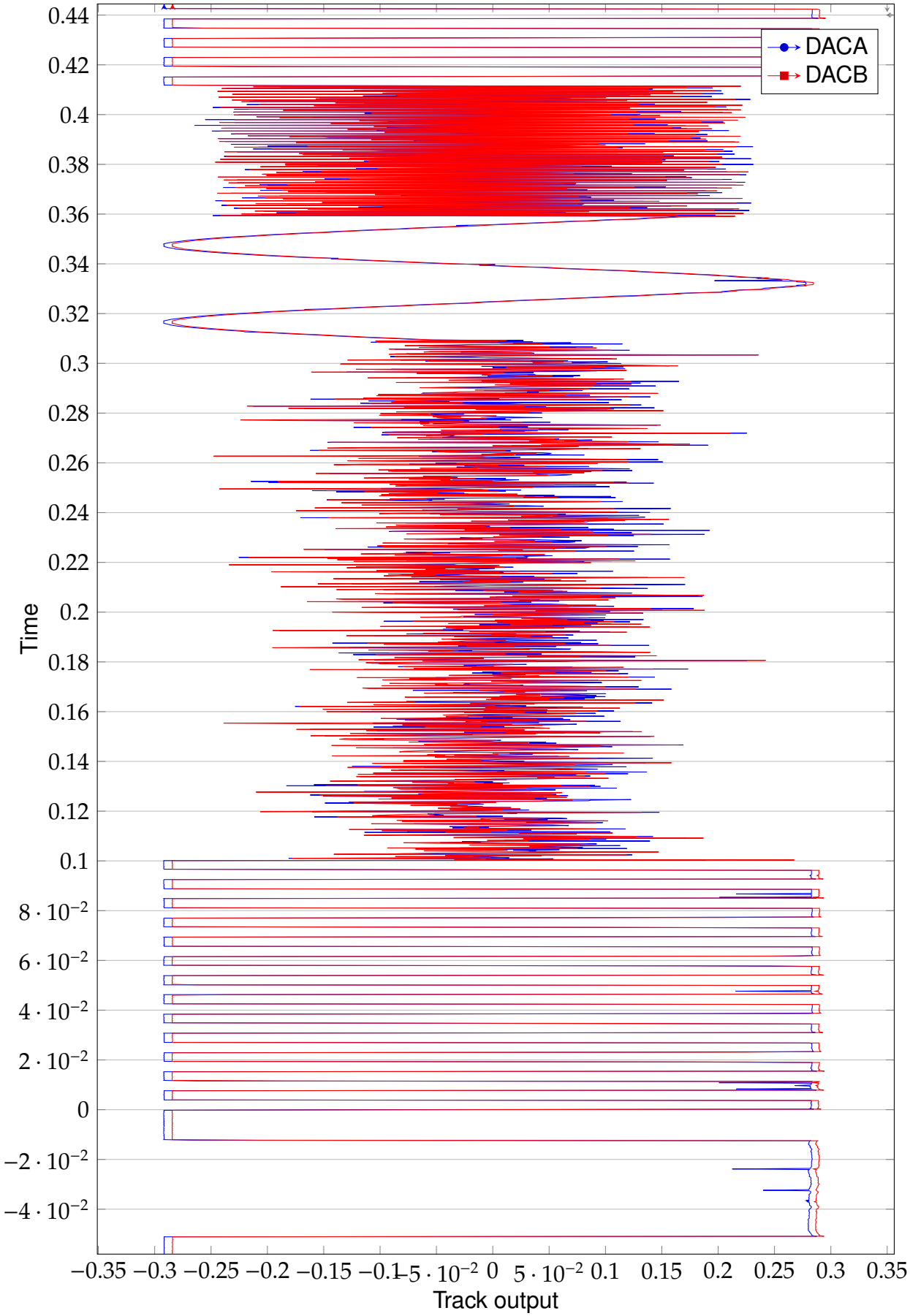

Figure 36: The output of an example track part 2

5.2.3 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.

5.3 User command interpretation

5.3.1 Command structure and parsing

As in other text adventures [25] 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 9 and 15.

As one command is parsed, each part is required to be separated by an empty space character, which is ascii code 32 [26]. 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 16

In listing 9 the comment echo back can be seen. The write_char function, writes it's parameter to the user., in this case the input sent by the user. This is done to write what the user typed out to the terminal as otherwise one would not be able to see what has been typed on any VT100 compatiable terminal[3] or terminal emulator.

```
1 void ingest_user_char(char in){
2     if(in == 0x7F /* DELETE CHAR */){
3         command_buffer[command_buffer_pointer--] = 0x00;
4 
5     }else{
6         command_buffer[command_buffer_pointer++] = in;
7     }
8     return;
9 }
```

Listing 15: The character ingest function

The in Listing 15 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 [26] 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 [3].

```
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 16: The command parsing function

5.3.2 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 17, the use item function is passed the command buffer[22] 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, because after command parsing, the string is overwritten with zeros as seen in Listing 16.

```
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);
```

[22] which is an address in memory

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

Listing 17: The command execution routine

5.4 Gameplay

The game itself plays like a regular game with limtations set in direction. Players can search for items in each room and grab the found items as can be seen in figure 37. The general gamplay is perfomred via altering the map data and the strings output to the user.

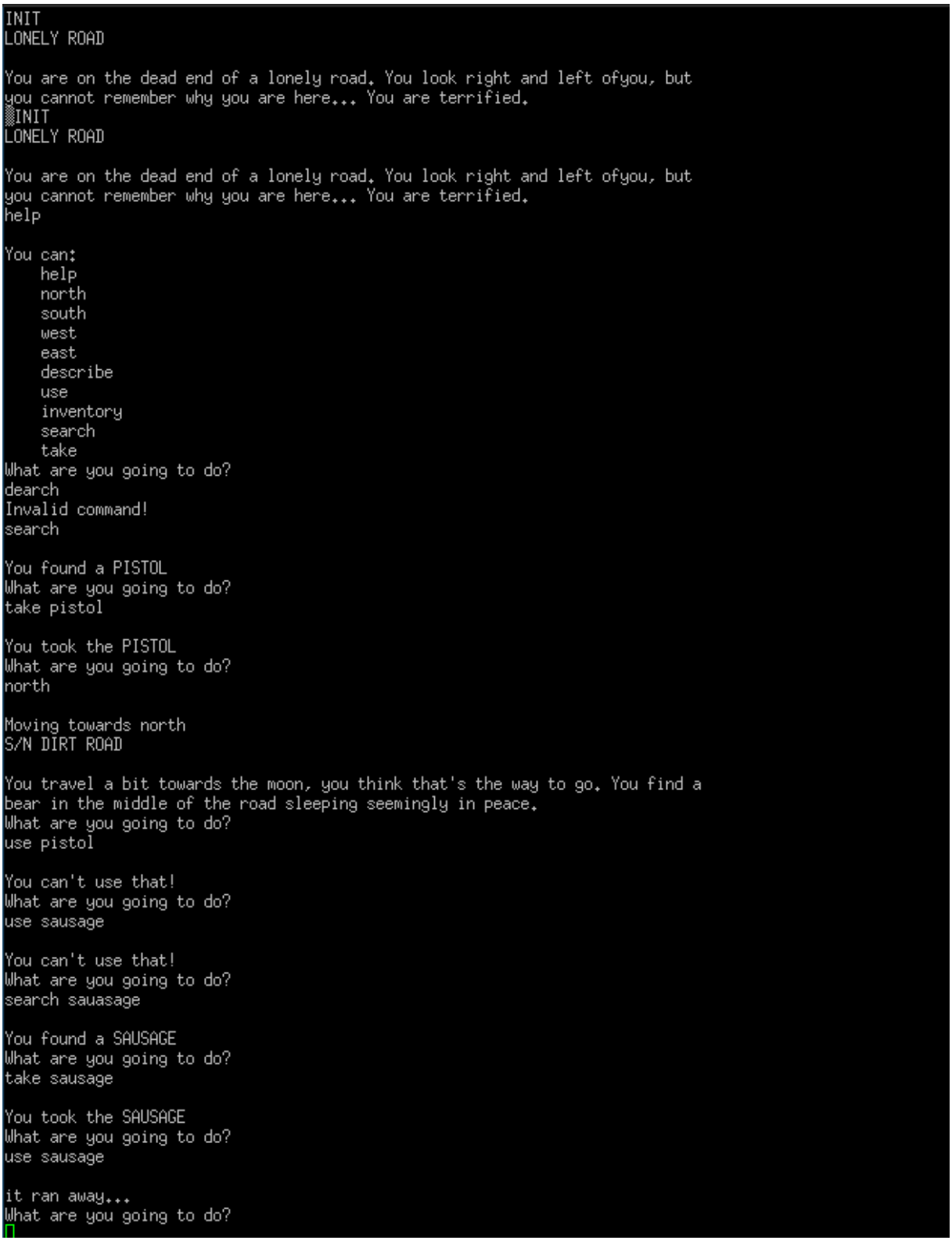

Figure 37: A regular beginning of the game

5.4.1 Memory constraints

The AVR has 8kB of internal SRAM, which are used for stack and heap [14]. 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[27]. 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[28]. In listing 12 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[28] 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.

5.4.2 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.

5.4.3 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 and the outcome changed.

5.4.4 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 18 and the state diagram in Figure 38.

```
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 18: The computer FSM

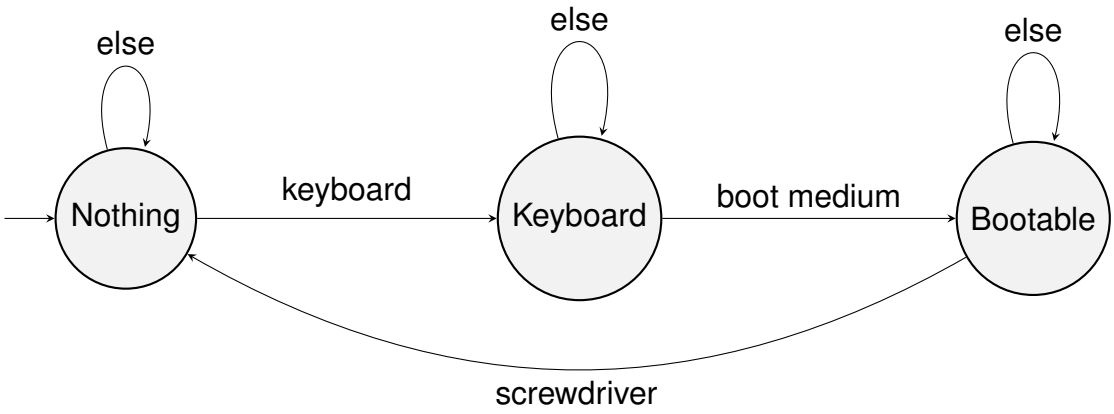

Figure 38: A state diagram of the computer state machine

6 FPGA Development

The project started out with the desire to build a CPU from scratch. Examples such as The NAND Game [29] and Ben Eater's Breadboard Computer series [30] 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 makes 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[23] containing an Altera Cyclone III[24] 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[25] with a Xilinx Artix-7[26] FPGA was ordered from Digilent.

A comparison between the two FPGAs themselves can be seen in Table 6, a comparison between the peripherals on the development boards in Table 7.

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

Table 6: Comparison between Altera and Xilinx FPGAs

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

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

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

[26] 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

Table 7: Comparison between the peripherals on Terasic and Digilent FPGA development boards

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 an HDMI-compatible DVI interface that is accessible through one of the high-speed PMOD connectors.

6.1 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 strong type system [31], it was selected as the language of choice for the project.

To refresh the reader's memory on the VHDL language, and as a quick guide for the tools involved in this project, see Appendix B.

6.1.1 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.

6.1.2 Free Software Tools

A somewhat recent development is the creation of Free Software FPGA toolchains. A breakthrough was achieved by Claire (formerly Clifford) Wolf in 2013 with yosys [32], [33], a feature-complete Verilog synthesis suite for Lattice's `iCE40` FPGA series. Since then, both yosys and place-and-route tools like nextpnr [34] 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 [35], a bridge between GHDL [36] and yosys allowing VHDL to be synthesized just the same as Verilog, and Dave Shah added Xilinx support to nextpnr [37]. The latter was preceded by many months of volunteer work reverse-engineering the Xilinx bitstream format as part of *Project X-Ray* [38].

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 and ghdlsynth as a bridge between them, 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

7 The Core

The core implements the rv32i architecture as specified by the RISC-V standard [39].

As can be seen in 39, it is constructed according to the traditional stages of a 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.

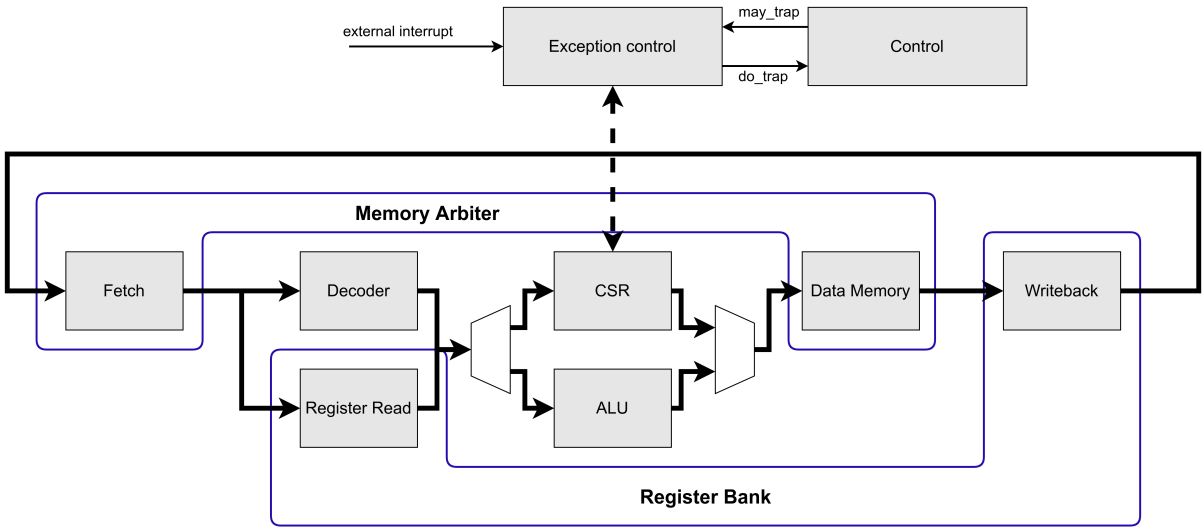

Figure 39: Block diagram of the CPU core

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.

7.1 Control

```
1  entity control is
2      generic (
3          RESET_VECTOR : yarm_word
4      );
5      port (
6          clk   : in std_logic;
7          reset : in std_logic;
8
9          fetch_enable    : out std_logic;
10         fetch_ready     : in std_logic;
11         fetch_instr_out : in yarm_word;
12
13         decoder_enable         : out std_logic;
14         decoder_instr_info_out : in instruction_info_t;
15
16         registers_data_a : in yarm_word;
17         registers_data_b : in yarm_word;
18
19         alu_enable_math : out std_logic;
20         alu_math_result : in yarm_word;
21         alu_valid       : in std_logic;
22         alu_enable_cmp  : out std_logic;
23         alu_cmp_result  : in compare_result_t;
24
```

```
25          csr_enable           : out std_logic;
26          csr_ready            : in std_logic;
27          csr_data_read        : in yarm_word;
28          csr_increase_instret : out std_logic;
29
30          datamem_enable         : out std_logic;
31          datamem_ready          : in std_logic;
32
33          alignment_raise_exc : out std_logic;
34          alignment_exc_data  : out exception_data_t;
35
36          registers_read_enable  : out std_logic;
37          registers_write_enable : out std_logic;
38
39          -- TRAP CONTROL
40
41          may_interrupt     : out std_logic;
42          -- the stage that will receive an interrupt exception
43          interrupted_stage : out pipeline_stage_t;
44
45          do_trap     : in std_logic;
46          trap_vector : in yarm_word;
47
48          trap_return_vec : in yarm_word;
49          return_trap     : out std_logic;
50
51          -- instruction info records used as input for the respective stages
52          stage_inputs : out pipeline_frames_t
53      );
54  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.

7.2 Decoder

```
1   entity decoder is
2       port (
3           clk    : in std_logic;
4           enable : in std_logic;
5
6           async_addr_rs1 : out register_addr_t;
7           async_addr_rs2 : out register_addr_t;
8
9           alu_muxsel_a    : out mux_selector_t;
10          alu_muxsel_b    : out mux_selector_t;
11          alu_muxsel_cmp2 : out mux_selector_t;
12
```

```
13          csr_muxsel_in : out mux_selector_t;
14
15          instr_info_in  : in instruction_info_t;
16          instr_info_out : out instruction_info_t;
17
18          raise_exc : out std_logic;
19          exc_data  : out exception_data_t
20      );
21  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

7.3 Registers

```
1   entity registers is
2       port (
3           clk   : in std_logic;
4
5           read_enable  : in std_logic;
6           write_enable : in std_logic;
7
8           addr_a : in register_addr_t;
9           addr_b : in register_addr_t;
10          addr_d : in register_addr_t;
11
12          data_a : out yarm_word;
13          data_b : out yarm_word;
14          data_d : in yarm_word
15      );
16  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.

7.4 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.

7.5 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;

```

```
27          do_trap     : in std_logic;
28          return_m_trap : in std_logic;
29          mepc_in       : in yarm_word;
30          mcause_in     : in yarm_trap_cause;
31          mtval_in      : in yarm_word;
32
33          raise_exc : out std_logic;
34          exc_data  : out exception_data_t
35      );
36  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.

7.6 Memory Arbiter

```
1   entity memory_arbiter is
2       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.

7.7 Exception Control

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

        fetch_raise_exc : in std_logic;
        fetch_exc_data  : in exception_data_t;

        -- synchronous exceptions
        decoder_raise_exc : in std_logic;
        decoder_exc_data  : in exception_data_t;

        csr_raise_exc : in std_logic;
        csr_exc_data  : in exception_data_t;

        alignment_raise_exc : in std_logic;
        alignment_exc_data  : in exception_data_t;

        datamem_raise_exc : in std_logic;
        datamem_exc_data  : in exception_data_t;

        -- interrupts
        global_int_enabled : in std_logic;
        interrupts_enabled : in yarm_word;
        interrupts_pending : in yarm_word;

        -- stage inputs for return address + trap value (instruction)
        stage_inputs      : in pipeline_frames_t;
        interrupted_stage : in pipeline_stage_t;

        may_interrupt : in std_logic;
        do_trap       : out std_logic;
        trap_cause    : out yarm_trap_cause;
        trap_address  : out yarm_word;
        trap_value    : out yarm_word
    );
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.

8 SoC Peripherals

The complete FPGA design consists not only of the CPU core, but a number of components that allow it to operate as well as to communicate with the outside environment. They are connected to the core using a shared 32-bit bus.

8.1 UART

The easiest way to communicate with an embedded system is usually through a serial interface. To ensure the best compatibility with existing software, a National Semiconductor 16550 UART was reimplemented from scratch instead of creating a new design. Thus, the modules's functionality and design can be found in the 16550's datasheet [4].

8.2 DVI graphics

As can be seen in Figure 40, the graphics module consists of several subcomponents:

- The VGA timing generator creates the impulses and counters necessary to drive a VGA-, DVI- or HDMI-based display
- The text renderer draws text characters onto the screen using a built-in font ROM
- The TMDS encoder frontend converts the internal parallel signals into a set of high-speed serial streams necessary for DVI or HDMI.

8.2.1 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 colour red, green and blue) scan across the screen a single horizontal line at a time while being modulated by the colour 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

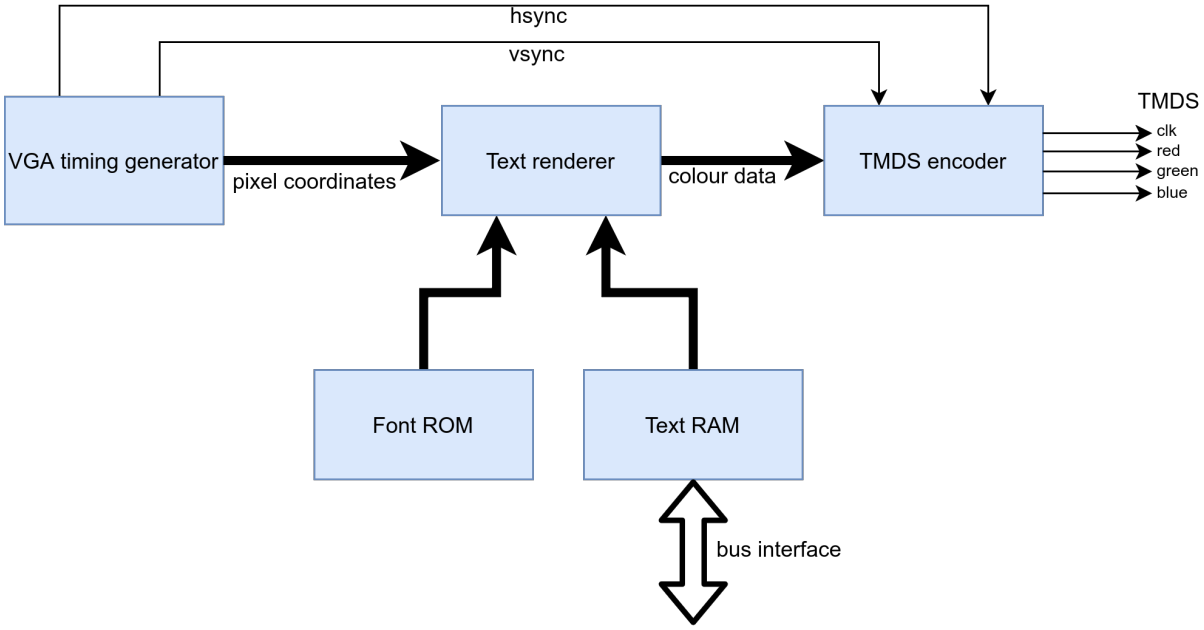

Figure 40: Block diagram of the video core

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 as visualized in Figure 41, 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.

8.2.2 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.

As can be seen in Figure 42, 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 within 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

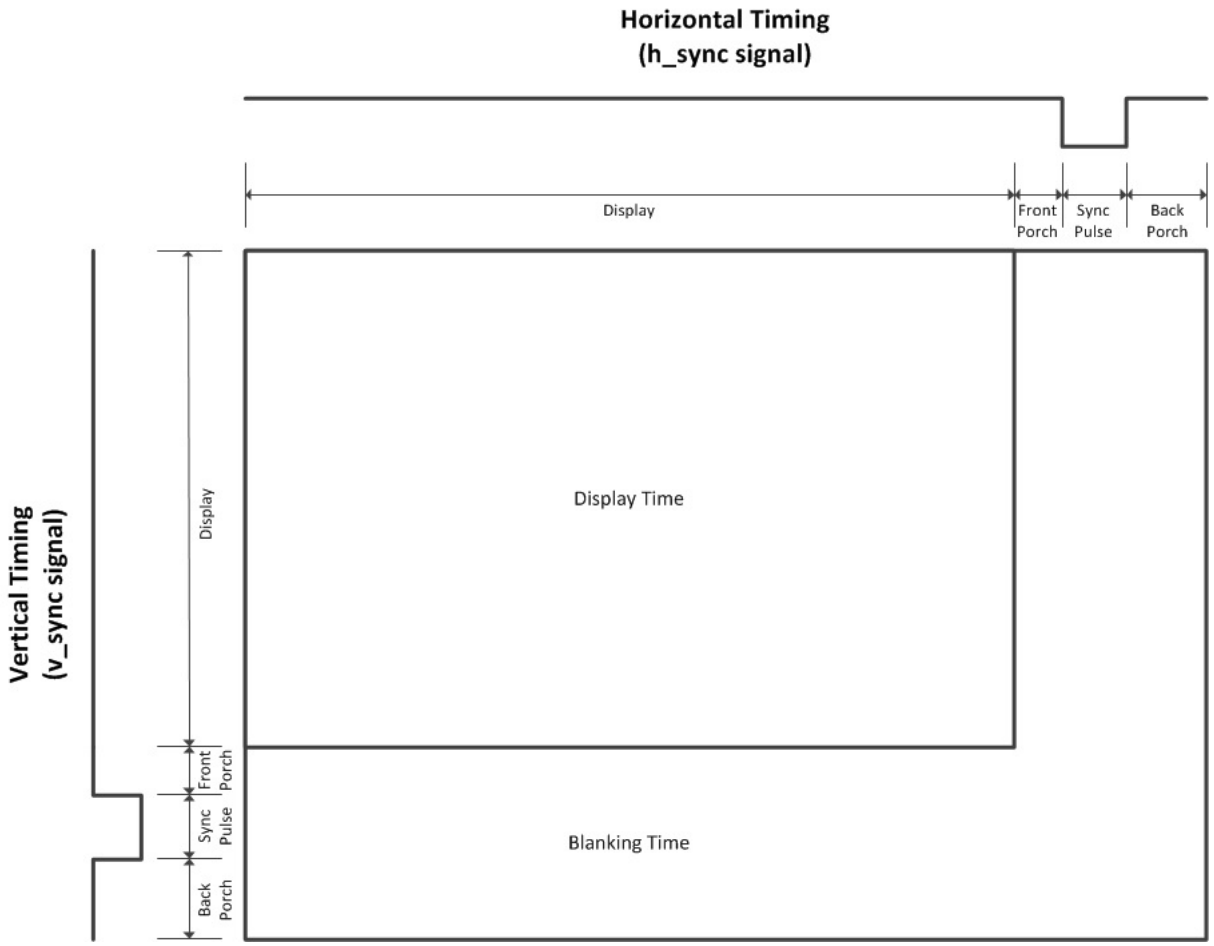

Figure 41: Diagram of VGA timing intervals

ROM to determine the colour of the pixel.

8.2.3 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 colour 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.

8.3 Ethernet

The Arty development board contains an RJ-45 Ethernet jack connected to an Ethernet PHY. The PHY handles the physical connection to an copper twisted pair Ethernet

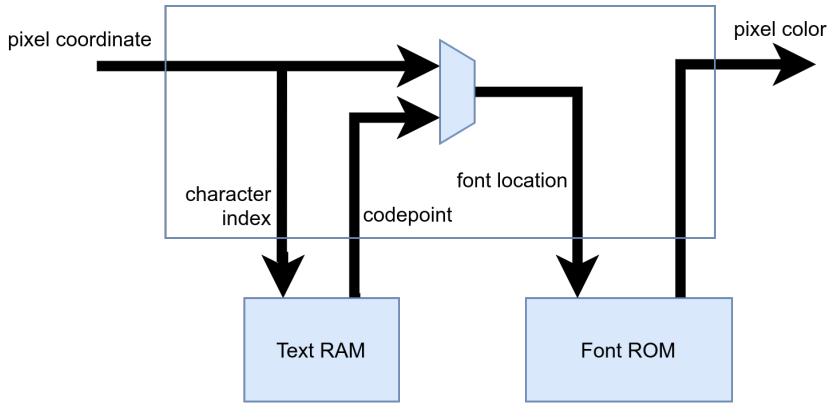

Figure 42: Block diagram of the text renderer

network (Layer 1 of the OSI model) and exposes a standardized media-independent interface (MII) to the FPGA. The LiteEth core [40], which is released under a Free Software license, is used to integrate the Ethernet interface into the SoC.

8.4 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.

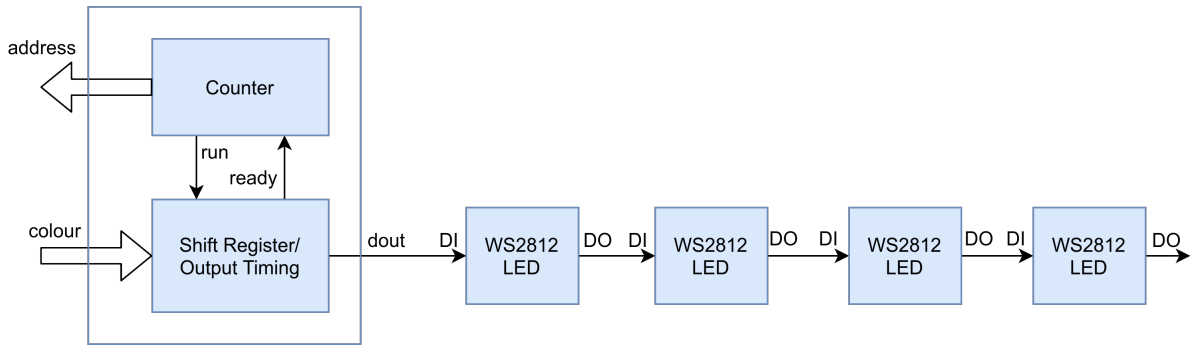

Figure 43: Block diagram of the WS2812 driver

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

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 colour value (8 bits each for red, green, blue), all following bits are passed through unmodified.

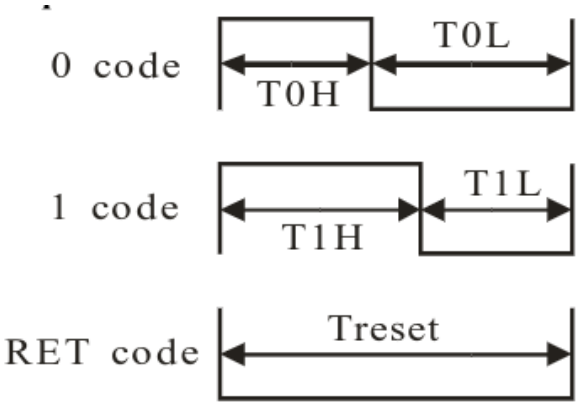

Figure 44: Timing diagram for the WS2812 serial protocol

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 Figure 44.

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

8.5 DRAM

The Arty A7 development board contains a 256MB DDR3 memory module. Since the FPGA only contains about 1.8MB of block RAM, some of which is already reserved for various hardware functions (e.g. the text buffer and WS2812 driver), the external memory is absolutely necessary to run larger programs.

Interfacing with DDR3 memory is notoriously difficult, requiring complex logic on both physical and logical layers. For this reason, the Free Software LiteDRAM core [41] is used to integrate the entire memory interface into the SoC. While irrelevant to the SoC, it can still be considered a slight peculiarity that the LiteDRAM core actually contains an entire separate RISC-V core to coordinate initialization of the memory.

8.6 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. Lastly, the bus must be slowed down. While the internal bus runs at a frequency of 50 MHz, a reasonable frequency for the external circuitry is around 1 MHz. To achieve this, a clock divider is used to only change the state of the external bus interface every 64th clock cycle, resulting in an effective bus speed of under 1 MHz.

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.

9 Software

9.1 Bootloader

The CPU loads its machine code from an FPGA-internal block RAM. The initial value for this RAM is part of the bitstream, and if any changes to it are required, the entire project has to be resynthesized. Because this takes upwards of 5 minutes, a different solution was created: a fixed bootloader is encoded into the block RAM, which is able to read additional program code (the payload) from the UART at runtime and store it to available memory. After the transfer is complete, it simply jumps to the base address of the payload and continues execution from there. When the current payload exits or a hardware reset is actuated, a new program can be loaded instantly.

Because many subroutines are used in both the loader and the payload, duplicating them in the payload would be a waste of space. Using custom linker scripts and compiler flags, the payload is linked against the functions in the loader. Whenever a loader function is called from the payload, execution jumps to bootloader code, executes the requested actions and then returns to the payload.

9.2 Drivers

Several components required writing functions to make them easier to use. Some are as simple as writing a value to a specific memory location:

```
1 void set_rgb_led(size_t num, uint32_t color) {
2     ((volatile uint32_t*)ADDRESS_RGB_LEDS)[num] = color;
3 }
```

Listing 26: Function to set the colour of an RGB LED on the Arty board

Others, like the function to write a character to the screen are more complicated and use further subroutines:

```
1  void vga_putchar(screen_t *s, unsigned char c) {
2      switch(c) {
3          case '\n':
4              set_cursor_pos(s, s->row + 1, 0);
5              break;
6
7          case '\b':
8          // DEL
9          case 0x7F:
10             if (s->col > 0) {
11                 set_cursor_pos(s, s->row, s->col - 1);
12             }
13             if (c == 0x7F) {
14                 set_curr_char(s, ' ');
15             }
16             break;
17
18         default:
19             set_curr_char(s, c);
20             set_cursor_pos(s, s->row, s->col + 1);
21     }
22 }
```

Listing 27: Function to write a character to the screen

10 Testing

10.1 RISC-V Compliance Tests

The RISC-V Compliance Test Suite [42] 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) [43] 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.

10.2 Formal Verification

While carefully selected simulation is useful to uncover bugs and to ensure they can't happen again (regression testing), it never offers complete certainty - it is simply impossible to manually cover all possibilities of inputs. With formal verification, the circuit under test is expressed using a mathematical model and an algorithm (a SAT solver) ensures that certain manually-selected criteria are always fulfilled. A detailed explanation of the algorithm can be found in [44].

As an example of formal verification, the core's ALU (subsection 7.4) has been extended with a formal verification definition, which can be seen in Listing 28. Skipping over some helper logic in the beginning, the first statements add assumptions about the entity's input signals. These are rules that must be obeyed by designs using the component, otherwise the correct function cannot be guaranteed (and is indeed unproven).

Below these assumptions, a process is used to calculate the expected result whenever a calculation is requested. While most operations are implemented the same as in

the main entity and thus have little value as a known-good comparison value, the bit-shift operations are implemented incrementally in the main ALU and directly in the verification; thus, the less resource-intensive ALU implementation can be confirmed to function exactly like this more expensive method.

Afterwards, the first assertions actually happen: these are the theorems that the formal verification suite will prove to be correct. The only two proven statements that are actually relevant to users are that when a result has been computed, it will equal the value computed using the aforementioned process, and that a computation will always finish eventually.

Finally, two more assertions are used to give hints to the formal verification algorithm, specifically the induction step. It is sometimes very difficult or even impossible to arrive at a successful induction; these assertions can be proven trivially, eliminating a number of potential scenarios that would otherwise make a successful complete proof impossible.

```
1  formal: block
2      signal prev_a : MATH_WORD;
3      signal prev_b : MATH_WORD;
4      signal prev_operation : alu_operation_t;
5
6      signal expected_result : MATH_WORD;
7
8      signal has_run : std_logic := '0';
9
10     signal prev_enable_math : std_logic;
11 begin
12     default clock is rising_edge(clk);
13
14     process(clk)
15     begin
16         if rising_edge(clk) then
17             prev_a <= a;
18             prev_b <= b;
19             prev_operation <= operation;
20             prev_enable_math <= enable_math;
21         end if;
22     end process;
23
24     -- assume inputs won't change while calculation is ongoing
25     assume always not valid -> (
26         a = prev_a and
27         b = prev_b and
28         operation = prev_operation
29     );
30
31     -- assume the "run" input is active at least until the result is valid
32     assume always prev_enable_math -> (enable_math or valid);
33
34     process(clk)
35     begin
36         if rising_edge(clk) and enable_math = '1' and valid = '1' then
37             has_run <= '1';
38
```

```
39              expected_result <= (others => '0');
40              case operation is
41                  when ALU_AND =>
42                      expected_result <= a and b;
43                  when ALU_OR =>
44                      expected_result <= a or b;
45                  when ALU_XOR =>
46                      expected_result <= a xor b;
47                  when ALU_SHIFTL =>
48                      expected_result <= std_logic_vector(shift_left(
49                          unsigned(a),
50                          to_integer(unsigned(b(4 downto 0)))
51                      ));
52                  when ALU_SHIFTR_L =>
53                      expected_result <= std_logic_vector(shift_right(
54                          unsigned(a),
55                          to_integer(unsigned(b(4 downto 0)))
56                      ));
57                  when ALU_SHIFTR_A =>
58                      expected_result <= std_logic_vector(shift_right(
59                          signed(a),
60                          to_integer(unsigned(b(4 downto 0)))
61                      ));
62                  when ALU_ADD =>
63                      expected_result <= std_logic_vector(signed(a) + signed(b));
64                  when ALU_SUB =>
65                      expected_result <= std_logic_vector(signed(a) - signed(b));
66              end case;
67          end if;
68      end process;
69
70      -- When a result has been computed, it must be correct
71      assert always valid and has_run -> math_result = expected_result;
72      -- Eventually, a result will always be available
73      assert always enable_math -> eventually! valid;
74
75      -- Hints for induction
76      assert always not valid -> enable_math;
77      assert always not valid and (
78          operation = ALU_SHIFTL or
79          operation = ALU_SHIFTR_L or
80          operation = ALU_SHIFTR_A) -> not (or current_b(current_b'left downto 5));
81  end block;
```

Listing 28: Formal verification block for the ALU

11 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

12 List of Figures

13 List of Tables

14 Listings

References

[1] Free Software Foundation: What is free software? URL: https://www.fsf.org/about/what-is-free-software (visited on 03/31/2020).

[2] J. Rutledge B. Olyha: TrackPoint Engineering Specification Version 4.0 Serial Supplement. IBM Corp. 1998. URL: https://web.stanford.edu/class/ee281/projects/aut2002/yingzong-mouse/media/Serial%20Mouse%20Detection.pdf.

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

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

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

[6] Unknown author: IEEE 1284: Parallel Ports. Lava Computer MFG Inc. 1998. URL: https://www.lavaports.com/wp-content/uploads/white_papers/ieee1284_parallel_ports.pdf.

[7] Dan Cox: Overview of Audio Codec '97. Intel Corporation. 1996. URL: http://euc.jp/periphs/AC97_OVR.PDF.

[8] Various: IBM Personal Computer AT Technical Reference. International Business Machines Corporation. Mar. 1984. URL: http://euc.jp/periphs/AC97_OVR.PDF.

[9] John von Neumann: 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.

[10] Chris Knebel Ian Kaneshiro Josh Knebel Nathan Riopelle: VGA Student Presentation. University of Michigan. URL: https://www.eecs.umich.edu/courses/eecs373/Lec/StudentF18/VGA%20Student%20Presentation.pdf.

[11] Unknown Author: I2C-bus specification and user manual. NXP Semiconductors N.V. Apr. 2014. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf.

[12] Unknown Author: VESA Enhanced Display Data Channel (EDDC) Standard. Video Electronics Standards Association. Dec. 2007. URL: https://glenwing.github.io/docs/VESA-EDDC-1.2.pdf.

[13] Unknown Author: PCA9564 Parallel bus to I2C-bus controller. Philips Semiconductors. Sept. 2006. URL: https://www.nxp.com/docs/en/data-sheet/PCA9564.pdf.

[14] 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.

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

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

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

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

[19] 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.

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

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

[22] Ethan Winer: The Audio Expert: Everything You Need to Know About Audio. Focal Press, 2013. URL: https://books.google.com/books?id=TIf0AwAAQBAJ&pg=PA107#v=onepage&q=-%2010%20dbv&f=false.

[23] 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/.

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

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

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

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

[28] 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.

[29] Olav Junker Kjær: The Nand Game. URL: http://nandgame.com (visited on 03/29/2020).

[30] Ben Eater: Building an 8-bit breadboard computer! 2016. URL: https://www.youtube.com/playlist?list=PLowKtXNTBypGqImE405J2565dvjafglHU (visited on 03/29/2020).

[31] Klaus Fricke: „Digitaltechnik - Lehr- und Übungsbuch für Elektrotechniker und Informatiker“. In: Springer Vieweg, 2013. Chap. 15.3. DOI: 10.1007/978-3-8348-2213-0.

[32] Johann Glaser Clifford Wolf: „Yosys - A Free Verilog Synthesis Suite“. 2013. URL: http://www.clifford.at/yosys/files/yosys-austrochip2013.pdf (visited on 03/29/2020).

[33] Various Contributors: Yosys - Yosys Open SYnthesis Suite. URL: https://github.com/YosysHQ/yosys (visited on 03/29/2020).

[34] Various Contributors: nextpnr - a portable FPGA place and route tool. URL: https://github.com/YosysHQ/nextpnr (visited on 03/29/2020).

[35] Tristan Gingold: ghdlsynth-beta. URL: https://github.com/tgingold/ghdlsynth-beta (visited on 03/29/2020).

[36] Tristan Gingold: ghdl. URL: https://github.com/ghdl/ghdl (visited on 03/29/2020).

[37] David Shah: nextpnr-xilinx. URL: https://github.com/daveshah1/nextpnr-xilinx (visited on 03/29/2020).

[38] SymbiFlow: Project X-Ray. URL: https://github.com/SymbiFlow/prjxray (visited on 03/29/2020).

[39] The RISC-V Instruction Set Manual - Volume I: Unprivileged ISA. 2019. URL: https://content.riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf (visited on 03/29/2020).

[40] Florent Kermarrec: LiteEth. URL: https://github.com/enjoy-digital/liteeth (visited on 03/29/2020).

[41] Florent Kermarrec: LiteDRAM. URL: https://github.com/enjoy-digital/litedram (visited on 03/29/2020).

[42] Lee Moore Jeremy Bennett: RISC-V Compliance Task Group. URL: https://github.com/riscv/riscv-compliance (visited on 03/29/2020).

[43] GitLab CI/CD. URL: https://docs.gitlab.com/ee/ci/ (visited on 03/29/2020).

[44] Clifford Wolf: Formal Verification withSymbiYosys and Yosys-SMTBMC. URL: http://www.clifford.at/papers/2017/smtbmc-sby/slides.pdf (visited on 03/29/2020).

[45] Tony Bybell: GTKWave. URL: http://gtkwave.sourceforge.net (visited on 03/29/2020).

[46] Gwenhael Goavec-Merou: openFPGALoader. URL: https://github.com/trabucayre/openFPGALoader (visited on 03/29/2020).

Appendices

A Projektmanagement

A.1 Schlussfolgerung / Projekterfahrung

Aus der Projektimplementierung konnten viele Lehren gezogen werden. Messungen welche mittels des Analog Discovery durchgeführt wurden sind bis zu ungefähr 1MHz Frequenz gut zu gebrauchen werden danach jedoch sehr stark fehlerhaft. Alle Bauteile in THT Bauform zu verwenden vereinfachte Messungen am Steckbrett erheblich, jedoch werden diese bei hohen Frequenzen unzuverlässig. Viele Implementationsdetails wurden durch mündlich übergebene Hinweise verbessert was zeigt wie wichtig zwischenmenschliche Kommunikation in technischen Bereichen ist.

A.2 Projektplanung

A.3 Projektterminplanung

A.3.1 Meilensteine

Brauns Tabelle 8 zeigt die zu Projektbeginn festgelegten Meilensteine.

Datum	Meilenstein
21.10.2019	Pflichtenheft, Grobdesign, Testplan, Core-Grundstruktur
17.12.2019	Komplettes Core-Simulationsdesign
21.01.2020	Simpler SoC (core+memory+LEDs) und Implementierung in FPGA
18.02.2020	Anbindung an diskrete Peripherie
10.03.2020	UART-Bootloader

Table 8: Meilensteine Brauns Armin

Plank Tabelle 9 zeigt die zu Projektbeginn festgelegten Meilensteine. Der Meilensteininhalt wurde nach der Aufgabenstellung zugeteilt, die Meilensteintermine wurden vom Betreuer festgelegt.

Datum	Meilenstein
22.10.2019	Pflichtenheft, Grobdesign, Testplan, Beschaffung der Unterlagen
10.12.2019	Serielle Schnitstelle
14.01.2020	8-Bit-Parallelport
12.02.2020	Dokumentation
10.03.2020	4-Bit-DAC mit R-2R-Netz

Table 9: Meilensteine Plank Daniel

A.3.2 Work time reference

Brauns Table 10 shows the times worked.

Date	Duration [h]	Task
2018-12-11	3	Create Quartus project, implement first proof-of-concept design
2018-12-19	3	ALU design and corresponding test bench
2019-01-18	5	First processor prototype capable of running programs
2019-01-20	2	Preliminary firmware build system
2019-01-28	6	VGA generator prototype
2019-01-29	6	VGA text renderer
2019-02-04	4	Control and Status Registers
2019-02-07	12	16550 compatible UART
2019-02-09	5	UART boot loader
2019-02-09	1.5	Build system improvements
2019-02-18	6	Unify simulation and synthesis SoC entities
2019-02-19	2	Debug and fix text renderer timing issues
2019-02-19	2	Add interrupts to UART
2019-02-19	1	Handle UART interrupts in payload
2019-02-20	4	Makefile-based build system
2019-02-20	0.5	Documentation
2019-02-21	1	Debug and fix VHDL simulation warnings
2019-02-26	2	Diagnosing and fixing core bugs
2019-02-26	3	Exception control unit
2019-02-26	2	Illegal instruction exceptions
2019-03-01	0.5	Breakpoint + environment call exceptions
2019-03-01	1	Build system improvements
2019-03-04	1	Misc bug fixes
2019-03-10	2	Diagnose and fix interrupt related processor bug

2019-03-11	3	CSR illegal instruction exceptions
2019-03-12	1	Preparations for pipelining core
2019-03-21	0.5	Switch to bare-metal compiler toolchain
2019-04-03	0.5	Build system improvements
2019-04-04	5	Design changes to improve timing
2019-04-05	4	Port design to Arty A7 and Vivado
2019-04-06	3	Port design to Arty A7 and Vivado
2019-04-06	3	TMDS generator frontend
2019-04-07	1	Add DVI output to Arty SoC
2019-04-11	0.5	Simulation tooling
2019-04-19	1	Split into core and soc respositories
2019-04-19	3	Cleanup after repo split
2019-04-21	4	Text renderer work
2019-04-22	4	Vivado project generator script
2019-04-22	1	Target Arty S7
2019-04-25	3	Colors in text renderer
2019-05-07	3	Firmware tooling
2019-05-07	3	UART resiliency
2019-05-08	4	Software ring buffer
2019-06-05	4	WS2812 driver
2019-07-04	2	Vivado DDR3 IP
2019-07-10	3	DDR3 interface
2019-07-13	6	memory development
2019-07-14	6	memory debugging
2019-07-19	3	Vivado in-circuit debugging
2019-07-25	5	Test HDMI output in hardware
2019-07-26	5	Add parity to UART
2019-09-19	1	Remove DE0/Quartus support
2019-10-04	1	UART Modem
2019-10-04	3	VHDL-2008 memory simulation model
2019-10-10	2	new core
2019-10-13	5	new core
2019-10-14	1	new core
2019-10-19	6	new core
2019-10-22	2	Merge new core
2019-10-31	5	Unify simulation and vivado SoC entities
2019-11-14	1	Fix `to_integer` simulation warnings

2019-11-21	5	DDR3 simulation entity
2019-12-01	3	Investigate Free toolchains
2019-01-02	0.5	Code cleanup
2019-01-12	4	Work toward free toolchain
2019-01-18	5	Toolchain testing and debugging
2019-01-23	1	UART improvements
2019-01-24	6	Switch whole project to Free toolchain
2019-01-25	3	Memory self-test routines
2019-01-25	0.5	Prepare ALU for mul/div
2019-02-01	7	Simplify core
2019-02-02	5	Compliance tests and core bug fixing
2019-02-02	2	GitLab CI
2019-02-04	1	Update toolchain
2019-02-08	5	Investigate LiteEth ethernet core
2019-02-09	5	Develop missing LiteEth features
2019-02-11	4	Add LiteEth to SoC
2019-02-16	2	LiteEth debugging firmware routines
2019-02-18	4	LiteEth simulation model
2019-03-01	2	Dependency updates
2019-03-02	2	Merge synthesis and simulation socs
2019-03-06	2	External bus interface
2019-03-06	3	Test external bus
2019-03-09	3	Debug UART boot
2019-03-15	1	Remove Vivado support
2019-03-28	4	Documentation
2019-03-28	1	Refactor ALU
2019-03-29	2	Documentation
2019-03-29	3	Add formal verification
2019-03-30	3	Documentation
2019-03-31	4	Documentation
2020-04-01	**SUM**	277h

Table 10: Work time reference - Brauns

Plank Table 11 shows the times worked.

Date	**Duration [h]**	**Task**

2019-09-06	4.25	start of thesis document
2019-09-07	2.25	planning of thesis
2019-09-20	1	planning part two, input into database
2019-09-23	0.5	corrections in database
2019-09-25	0.5	discussions with supervisor about deadlines
2019-09-27	0.25	reformatting and discussion about database entry
2019-10-11	2	tests and high level design for MS1
2019-10-12	3.75	gather PDFs for MS1
2019-10-16	2.5	tests and high level design for MS1
2019-10-17	2.5	tests and high level design for MS1
2019-10-20	4.25	tests and high level design for MS1
2019-10-22	3.5	Finalisation tests and high level design for MS1
2019-12-08	4.75	Download thesis template and implement
2020-01-03	6.75	Planning and early schematics of serial module
2020-01-04	2	Parallel port layout
2020-01-08	3.75	Serial console breadboard test
2020-01-11	2.5	Attempting interaction with 16550
2020-01-18	4.5	Attempting interaction with 16550 nailing down errors
2020-01-18	3	Attempting interaction with 16550
2020-02-25	1	Help partner with hosting tar.gz file
2020-01-26	6.25	Attempting interaction with 16550 no output
2020-02-01	3	Attempting interaction with 16550 quartz doesn't oscillate
2020-02-07	5.5	Attempting to make 1.8432MHz oscillators oscillate
2020-02-08	3	Oscillation succeeded... finaly
2020-02-09	7.75	Transmit character in serial via 16550
2020-02-10	4	Serial console eurocard
2020-02-11	5	Serial console and arduino eurocard
2020-02-12	5	Serial console and arduino eurocard
2020-02-13	4	Serial console and arduino eurocard testing
2020-02-14	6	Serial console and arduino eurocard code
2020-02-15	3.5	Serial console and arduino eurocard code
2020-02-18	3.5	ECHO! Program
2020-02-19	3.5	DAC schematic and breadboard beginning
2020-02-20	2.25	DAC driver simulation attempt
2020-03-01	3.25	Level shifter test and verification
2020-03-04	2	DAC fifo breadboard

2020-03-08	7.5	breadboard final test DAC and FIFO and eurocard
2020-03-10	4.75	DAC module test and sine generation code
2020-03-11	4.25	textadventure start
2020-03-12	4.25	textadventure polling dac and 16550
2020-03-13	4.5	finalisation of everything in school COVID-19
2020-03-14	5	textadventure DAC mode implementation
2020-03-15	4	textadventure sound routines
2020-03-17	4	textadventure gameplay
2020-03-18	6	documentation
2020-03-19	4	documentation
2020-03-20	3	documentation
2020-03-21	1	textadventure gamplay
2020-03-22	0.5	textadventure gamplay
2020-03-23	6.25	documentation
2020-03-24	6.75	documentation
2020-03-25	7.25	documentation
2020-03-26	7	documentation
2020-03-27	5.75	documentation
2020-03-28	4.5	documentation
2020-03-29	6.5	documentation
2020-03-30	9.75	documentation
2020-03-31	0	documentation
2020-04-01	**SUM**	229.5h

Table 11: Work time reference - Plank

B 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.

B.1 Prerequisites

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

`ghdl` **[36]** to analyze, compile, and simulate the design

`gtkwave` **[45]** to view the simulation waveform files

`yosys` **[33]** to synthesize the design

`ghdlsynth-beta` **[35]** to synthesize the design

`nextpnr-xilinx` **[37]** to place and route the design

`Project X-Ray` **[38]** for FPGA layout data and bitstream tools

`openFPGALoader` **[46]** to load the bitstream onto the FPGA

B.2 Creating a design

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

```vhdl
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)
    );
end counter;

architecture behaviour of counter is
    signal count : unsigned(7 downto 0) := (others => '0');
begin
    proc: process(clk)
    begin
        if reset then
            count <= (others => '0');
        elsif rising_edge(clk) and enable = '1' then
            if direction = '1' then
                count <= count + 1;
            else
                count <= count - 1;
            end if;
        end if;
    end process;

    count_out <= std_logic_vector(count);
end behaviour;
```

counter.vhd

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

```
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

B.3 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)
gtkwave counter_tb.ghw counter_tb.gtkw
```

Listing 31: Commands required to simulate the counter design

B.4 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]
```

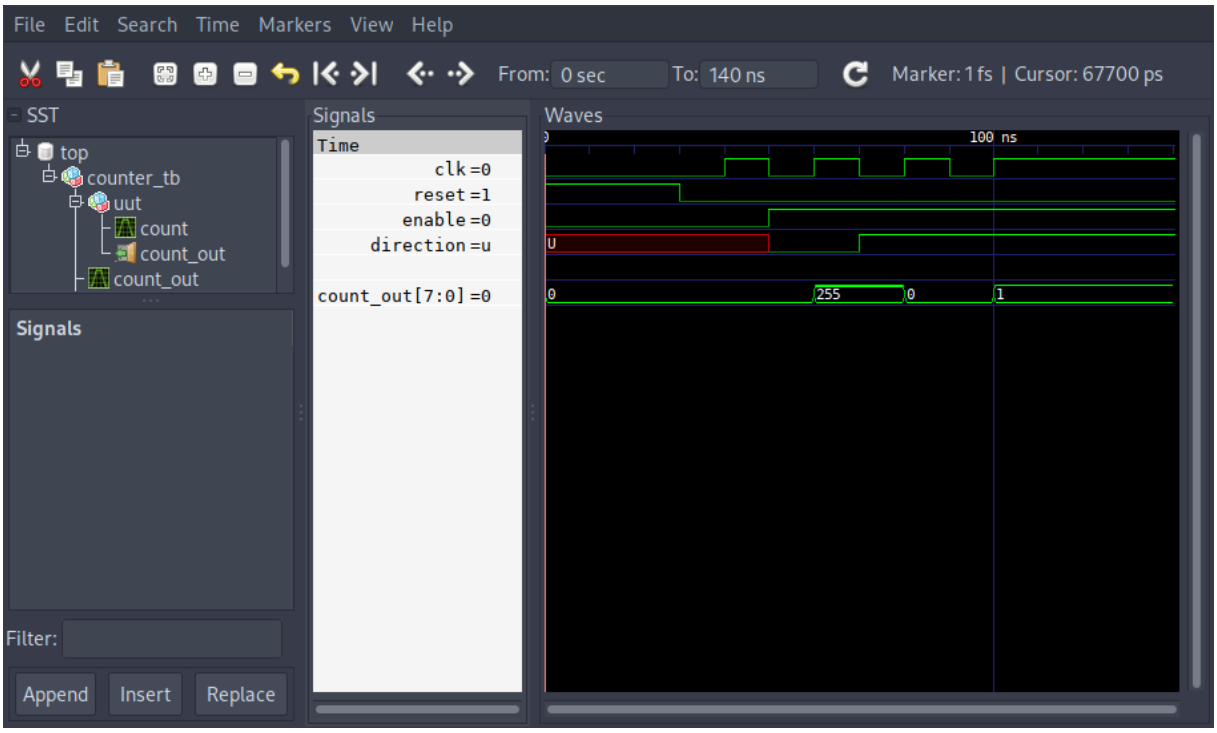

Figure 45: Screenshot of the counter test bench waveform in GTKWave

```
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 33: 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.