FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Programmable generator of synchronous pulse sequences
Student:	Vojtěch Nevřela
Supervisor:	Ing. Jaroslav Borecký, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer engineering
Department:	Department of Digital Design
Validity:	Until the end of summer semester 2020/21

Instructions

Design and implement a generator of synchronous pulse sequences for a field-programmable gate array utilizing VHDL or Verilog language.

The work will include:
1. Research of existing solutions.
2. Choice of a suitable hardware platform for implementation.
3. Design and implementation of a generator with these requirements:
   - the generator will contain at least eight channels,
   - additional control signals for channels (the number will depend on the selected platform),
   - high pulse sequence resolution (less than 10ns),
   - programmable sequences (minimum 500 commands per channel),
   - configuration of individual sequences of all channels via a serial link,
   - playback of sequences will be triggered by an external signal.
4. Simulation and testing of the resulting design.

References

Will be provided by the supervisor.

doc. Ing. Hana Kubátová, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 11, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Programmable generator of synchronous pulse sequences

Vojtěch Nevřela

Department of Digital Design
Supervisor: Ing. Jaroslav Borecký, Ph.D.

June 4, 2020

Acknowledgements

I wish to express my thanks to my friend and the opponent of this thesis, Michal Dudka, who has provided me with the necessary facilities and knowledge to complete the electronics part of the work. I would also like to thank my supervisor for sharing his expertise in digital design and much-appreciated guidance. Last but not least, my thanks go to my father for his unceasing encouragement and support. Finally, I am greatly indebted to my partner and her family for providing me with a temporary place to stay during the pandemic of 2020, enabling me to work on this thesis in peace, leading to it's completion.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on June 4, 2020 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Nevřela, Vojtěch. *Programmable generator of synchronous pulse sequences.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Předmětem práce je analýza dostupných technologií a následná implementace programovatelného vícekanálového sekvenceru pulzů za využití hradlového pole. Předmětem praktické části je samotný vývoj řešení v jazyce Verilog a jeho simulace a následné nasazení na hradlové pole. Rozlišení dosahující 1 ns bylo dosaženo za pomocí specializovanáno formátování instrukcí. Zařízení bylo také doplněno komunikační aplikací.

Klíčová slova FPGA, Xilinx, asembler, generátor sekvencí pulzů, Verilog, serializér

Abstract

The objective of this thesis is to analyze of the available technology and the eventual implementation of a programmable multichannel pulse sequencer using a gate array. The goal of the practical part is the actual development using the Verilog hardware description language, simulation, and the hardware realization of said device. Resolutions reaching 1 ns were achieved using specialized instruction formatting. The device is also accompanied by a communication application.

Keywords FPGA, Xilinx, assembler, pulse train generator, Verilog, serializer

Contents

List of Figures

List of Tables

Introduction

The need for precise scientific equipment is rising as the research becomes more focused on the minuscule features of our universe with every following day. This equipment sometimes is not readily available and when it is, carries a hefty price tag. The institutions that require these precision instruments often have to develop them on their own. These appliances can be divided into measurement and stimulus generation devices. This paper is concerned only with the latter. One example of such a device, and also the subject of this thesis, is the pulse train generator, also called the pulse sequencer, pulse sequence generator, or many other takes on the same name.

Pulse train generators are used to generate stimuli for a variety of experiments that require pulses of precise widths to be administered to inputs at correct times. The precision required can even reach the magnitudes of picoseconds. This makes the task vitally unattainable by microprocessor-based systems whose speeds are currently about magnitude lower and therefore, require the use of more specialized technology.

The need for such a device was felt by the Department of Optics of the Faculty of Science at Palacký University. The order to design a similar device that would surpass their in-house microprocessor design (which will be discussed in the following chapter) was therefore created. The main issue of their design was the lack of high enough resolution and speed. The proposed design aims to improve on their design with the use of a field-programmable gate array, which is more suitable for the task due to its parallel nature akin to the parallel nature of the task at hand.

The objective of this thesis is the research of available solutions to this problem and available technology for improving on them. The goal is finding a compromise between price and performance, followed by the implementation of the said device. The implementation will be done in Verilog hardware description language and will also include supporting software to enable the appliance to be reasonably simple to interface with even for the researchers that will be using it.

In the first chapter, the already existing solutions will be discussed. After that, the problem will be analyzed from the perspective of memory, bandwidth, and physical size and pin count of the utilized device. At the end of this chapter, the analysis of the manufacturers and available hardware will be conducted, and the final hardware is selected. The third and most involved chapter is concerned with the design and implementation of the pulse box. The design is discussed from all points of view. In the last chapter, the simulations implemented during the development are demonstrated, and the device is tested in real hardware.

Chapter 1

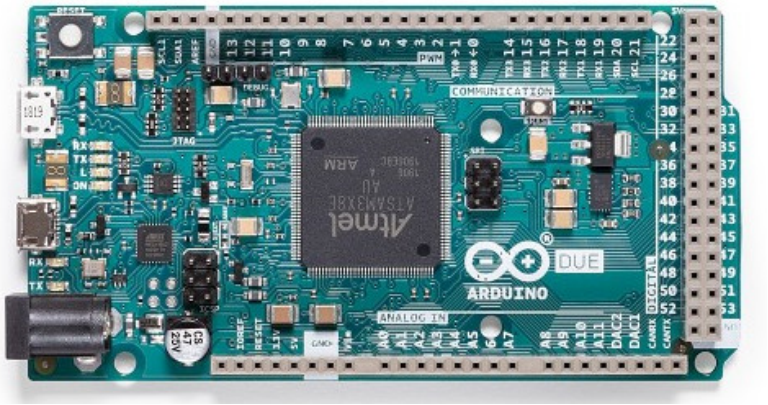
Previous work

1.1 In-house solutions

1.1.1 UPOL Design

At the Palacký University, an attempt to create such a device was made [1]. The device is based on a SAM3X microcontroller on an Arduino Due board. The central part of their design was a program capable of generating single purpose instruction sequences, which would toggle the output pins at the correct times. The timing was implemented using simple delay loops and interposed NOP instructions to improve precision. The main flaw of the design is the lack of speed due to the utilization of an MCU instead of dedicated logic. Arduino due runs on an 84 MHz clock [2]. This means that any changes to the output pins can happen no faster than that. This means the minimal theoretical output granularity is around 12 ns. This may prove insufficient for faster experiments.

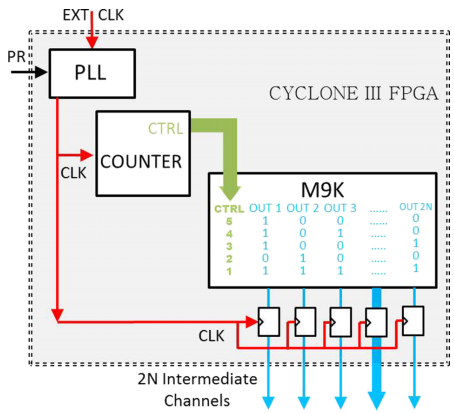
Figure 1.1: Arduino Due development board



1.1.2 Design by Ben Haylock and collective

Contrary to the previous, this design utilizes a Cyclone III FPGA [3]. The device, further called Haylock's device, incorporates a single memory unit composed of M9K primitives provided in the chip. The data is first preloaded in the chip memory and sent out in parallel after the assertion of an external trigger.

Figure 1.2: Schematic of pulse train generator by Ben Haylock and collective



It is also necessary to mention that the constraints for this design differ from ours in a few key points. Our design only needs to produce monopolar pulses only while Haylock's device is capable of generating bipolar pulses. Also, the required and variable channel power output is not a requirement for our design.

1.2 Commercial devices

1.2.1 Tektronix HFS 9000

In the 90s, Tektronix presented the HFS 9000 Stimulus system, aimed at various markets. It was supposed to be a universal and moddable platform for any experiment. This particular device is included here due to the reasonable pricing for the feature set provided. Despite being dated, the characteristics are still relevant for experiments of today. The channels are implemented using add-on cards HFS 9PG1 and HFS 9PG2. The speeds of the HFS 9000 greatly exceed the speeds expected from any in-house design. In the manual, the claim is that 1 ps resolution can be achieved [4]. This device fulfills the task requirements in all regards except for the number of channels. Also, using the word available may be an oversight. Due to the age of the said device, it is rather problematic to obtain one. Another drawback, should one consider using said device for their experiment, is the rather outdated user interface, consisting of a very simple text CRT display.

1.2.2 Keysight 81134A

The dual-channel Keysight 81134A and the single-channel version 81133A are the currently available instruments sold by Keysight. The devices boast output frequency range between 15 MHz and 3.35 GHz [5] but have only up to 2 channels. These instruments represent the group of commercially available devices very well. A complete package with full support, accompanying software for PC configurable from a PC with a price of refurbished models around 40,000 USD.

Figure 1.3: Keysight 81134A Pulse Pattern Generator

Chapter 2

Analysis

The requirements for the device were initially set very broadly. The only important parameter was the improvement over the design already used at UPOL. We set our design goals to following.

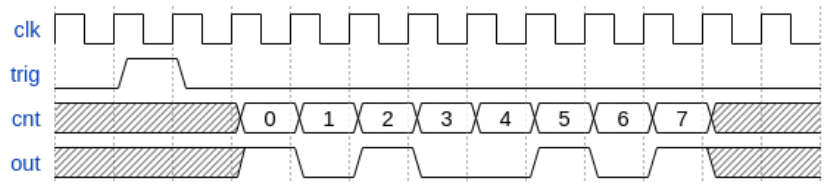
- 1 ns resolution[1]
- 16 output channels
- >1000 stored output state changes[2] for every channel
- 6 bit parallel outputs for some channels usable for connecting channel attenuators

2.1 Memory

Let us concern ourselves with the memory requirements first. There are two evident and naive approaches for storing the patterns in memory. The pattern may be stored directly or through time intervals instead.

The first approach implements the sequence storage as raw waveforms encoded as bits in the instruction memory. Every bit signifies the output state directly. In the following figure, the sequence generated from the value 0xA5 can be observed. The bits are read from the MSB first.

Figure 2.1: Output waveform for raw instruction storage scheme



[1]Minimal pulse or gap duration
[2]Pulses or gaps of minimal width

This approach is very naive. While simple to implement, much memory is wasted for sparse sequences. The only advantage could be the speed. The sheer simplicity of the design may increase the maximal clock frequency and, therefore, the resolution. Another problematic point would be adding the data for the parallel outputs. Due to the simplistic nature of the encoding, or more precisely, the lack thereof, the parallel output value would have come with every bit effectively multiplying the memory requirements 7-fold (1 bit for on/off, 6 bits for parallel value).

The other possible approach is to separate the sequences into instructions containing 3 fields, the output value, the time it should be displayed for, and the parallel output. This approach makes sure no memory is wasted for sequences with fewer transitions as only the transitions themselves are encoded. On the other hand, this means that narrow pulses or gaps would require whole multi-bit instructions for their realization, and if more dense pulse sequences are required, then memory is wasted.

Assuming 32 bit instructions, 1 bit dedicated to the output value, 6 bits to the parallel output value, and 25 bits for the duration, the sequence from the previous figure can be encoded using eight instructions. A single byte now suddenly takes up 256 bytes! It is obvious that implementing either of the naive approaches would lead to a severely inefficient encoding scheme. We can use the examples to roughly estimate the memory usage though. For the first scenario, the calculation is straightforward:

$$Memory = 1\text{bit} \cdot 16\text{ch} \cdot 1000\text{commands} = 16000\text{bit} = 15.615\text{Kbit}$$

And with parallel output data:

$$Memory = (1\text{bit} + 6\text{bit}) \cdot 16\text{ch} \cdot 1000\text{commands} = 112000\text{bit} = 109.375\text{Kbit}$$

Should the purely timed approach be used, the resulting memory usage approximation is as follows:

$$Memory = 32\text{bit} \cdot 16\text{ch} \cdot 1000\text{commands} = 512000\text{bit} = 500\text{Kbit}$$

The raw scheme is much more efficient for storing dense sequences but is wasteful when the sequences are sparse. The timed approach has exactly the opposite characteristics. This means a hybrid approach could be ideal and would allow the appropriate encoding scheme to be used for each use case. The exact implementation shall be discussed in greater detail in the design chapter.

2.2 Bandwidth

Calculating the total throughput will be simple as well. The worst case, independent of any encoding, is that the output bit would change on every clock edge. This means the data is presented to the outputs every nanosecond. The throughput for the 16 channels is then calculated as follows:

$$Throughput = \frac{1\text{b}}{1\text{ns}} \cdot 16\text{ch} = 16\text{Gbit/s}$$

Had the processing been done entirely in series, the hardware would need to be clocked at 16 GHz! This may seem like a colossal number, but it may not be as problematic as it may seem. The processing can be done in parallel at many levels. If every channel is processed independently, we already get a decrease to 1 GHz. If we process the data in bigger chunks, for example, bytes, and utilize some form of serialization at the end, the internal logic frequency drops to 125 MHz, a value completely reasonable for an FPGA. The details on how this was implemented will be discussed further in the design chapter.

2.3 Pin count

This design requires a significant number of output pins. There are a few considerations to take into account. The frequencies transmitted from each channel are very high, and therefore the signals can be easily impaired. The estimated pin counts for each case can be seen in the table 2.1.

Differential output	Attenuator	Pin count
No	No	19
Yes	No	35
No	Yes	115
Yes	Yes	131

Table 2.1: Pin counts required for various configurations

2.4 Available technology and hardware selection

Since the minimum pulse and gap width requirement was set to 1 ns, it was decided very early in the design process that FPGA would be the most suitable for the task. Using an MCU would inherently lead to design in many ways similar to the one realized at UPOL and would also lack the flexibility that we wish to achieve. Another major factor was the general availability of knowledge and hardware. Depending on the availability of resources and one's manufacturing facilities, one may decide to incorporate the chip into

	direct	daughterboard
Footprint	smaller	larger
Ease of design	harder	easier
Price	cheaper	more expensive
Support circuitry	power, communication, program flash, clock, ...	power, clock (depending on application)
Assembly	BGA packages require reflow soldering	depending on board can be hand soldered
Signal integrity	as good as possible	impaired by connectors

Table 2.2: Comparison between direct and daughterboard FPGA designs

the design directly or on a daughterboard. The first approach gives the designer the most flexibility but comes with a higher required level of electronics design knowledge. The second enables the designer to focus better on the task at hand but comes at the cost of lower feature and routing flexibility, accompanied by the bulkier size. For the design of the pulse box, the decision had to be made as well. In the table 2.2, we can see a more organized comparison.

In the end, the decision was made for us. The FPGAs of today come almost exclusively in BGA form. This means the pins are on the underside of the chip and can not be soldered with a handheld soldering iron. To affix the FPGA to the PCB, the chip is first fitted with solder balls and placed on the PCB. The entire assembly is then heated in a specialized oven. The chip is now fixed to the board. This technology is not readily available to us, and setting up a reflow soldering process would require a significant time investment. Because this was not an option, there were two other options left. Choosing an older FPGA (such as Cyclone IV from Altera/Intel), which is available in less dense and more flexible packages or the utilization of a daughterboard. The former was decided against due to the time constraints imposed on the design, thus leaving us with the second option.

2.4.1 FPGA Manufacturers and available hardware

In the FPGA space, there are two main manufacturers. Intel[3] and Xilinx. These two manufacturers hold over 80% of the market share [6]. The choice to utilize gate arrays from either of the manufacturers was made due to the well-established user space and general ease of access to documentation. Other manufacturers were not considered.

[3]until 2015 Altera

Intel FPGA portfolio

Intel currently markets 5 device families. Agilex, Stratix, Arria, Cyclone and Max [7]. The Agilex family is the Intel's newest offering, built on a 10 nm process that promises to do everything better than its predecessors both in terms of performance and power consumption. Since there was no need for either of those, the price increase was not justified.

Skipping ahead to the Max family, these FPGAs are Intel's offering when it comes to non-volatile, also called flash-based, FPGAs. The advantage of flash-based FPGAs is the very rapid start-up time and simpler electronic design as they do not need additional configuration memory. Because the chip does not need to pull the bitstream from a flash memory device, the operation may begin almost instantly after powered. This was not a requirement for our design either and has therefore ruled out this family due to the price increase as in the previous case.

The Stratix, Arria, and Cyclone are Intel's conventional FPGA families. The memory sizes commonly found in FPGA devices will not be a limiting factor as long as the most simple chips are avoided. The only requirement was the capability to output a signal of sufficient frequency. Because a signal of period as low as 1 ns must be transmittable, directly driving the FPGA pins from the fabric is out of the reach. This means the data has to be serialized in some fashion. For this exact reason, the chips are equipped with a so-called SERDES circuit [8], which is pre-built in hardware (not in FPGA fabric but directly in silicon) and can, therefore, be clocked to frequencies highly exceeding the ones attainable in the FPGA fabric. SERDES circuits are commonly found in all the families of the FPGAs, which in turn means that any device with sufficient memory and pin count can be used.

Xilinx FPGA portfolio

With Xilinx, the situation is somewhat similar. Their portfolio consists of the Virtex, Kintex, Artix, and Spartan families, sorted from the largest and most advanced to the smallest and cheaper. The families are further sub-divided into generations. As mentioned before, the preference was to utilize a more modern FPGA. This rules out the Spartan-6 family, currently the only supported 6th generation chip. The case with Xilinx is the same as with Intel. The vast majority of the devices are acceptable feature-wise, which means the deciding factor will be the availability of development kits and the support.

In regards to the support, both of the manufacturers run a forum full of employees and members who are more than willing to help with any aspect of the design. Also, the general availability of documentation is comparable.

Intel development kits

Kits equipped with Intel FPGA's are primarily manufactured by Intel themselves and by Terasic. At this point, it can be easily said that the vast majority of FPGAs manufacutred will be able to cater to our needs. The deciding factor will then have to be the form factor of the kits available.

The kits from Intel are of no real use to us. The intended use for these kits is the early prototyping and education. The boards are either in the format PCIe cards or user-friendly development kits with a lot of unnecessary features such as ethernet transceivers, displays, LEDs, and other. Other manufacturers do not have much more to offer, either. Terasic has a very similar offering, albeit with some boards that could be of use to us. [9]

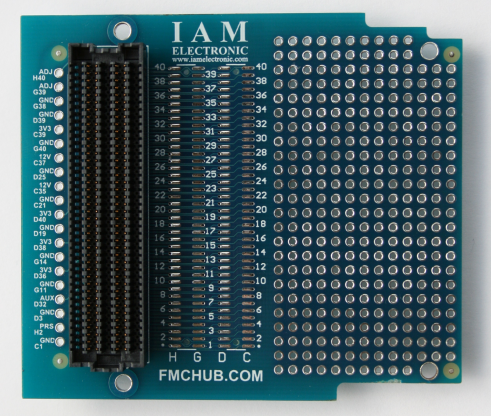
Figure 2.2: Terasic TR5 development kit

Setting the price aside, the TR5 development kit offers 2 HPC and 2 LPC FMC connectors and a FPGA more than sufficiently powerful. The main design disadvantage, and one of the reasons any solution involving FMC connectors was not implemented in the end, is the fact that the soldering techniques used for mounting such connectors are the same as for BGA chips. This means a commercially produced board would have to be used.

The LPC FMC connector contains 72 user utilizable signals, and the HPC variant has 200 [10]. This is a sufficient amount for our application. Breakout boards could then be used. The majority of available boards look alike to FPGA Mezzanine Card (FMC) LPC Breakout Board by IAM Electronic.

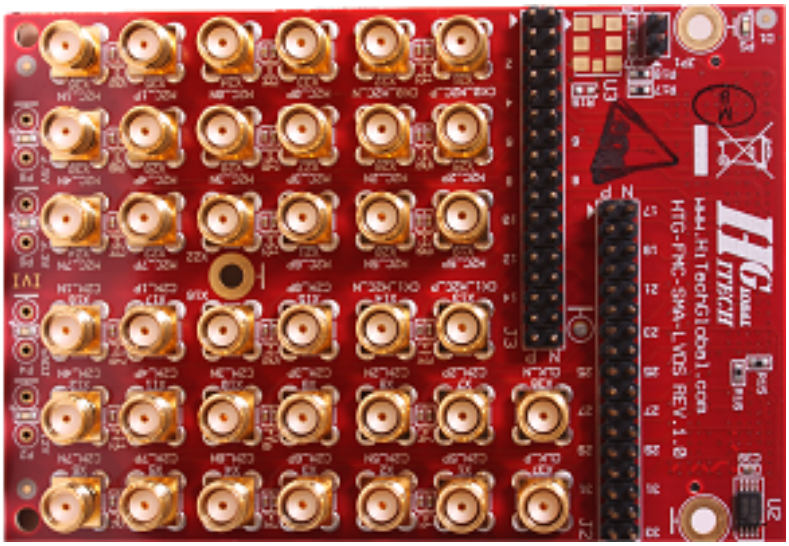
With these boards, the issue could be the signal integrity and the possibly haphazard nature of the final wiring. The focus was turned to breakout boards with SMA connectors. An example was selected from HiTech Global. While more suitable, the price is higher, and the number of outputs is only 32. This

Figure 2.3: IAM Electronic breakout board



means a single SMA card like this could cover only the channels and more outputs would be necessary.

Figure 2.4: 8-Port SMA / 34 Differential Pair FMC Module (Vita57.1) from HiTech Global

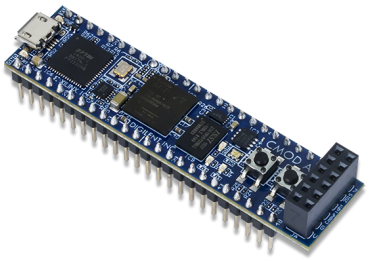


Xilinx development kits

The availability of suitable Xilinx based hardware is somewhat better. With kits manufactured by Xilinx themselves, the story is somewhat similar with the boards manufactured by Intel. Again, we find kits aimed primarily at the early prototyping stage of electronics development and the education sector.

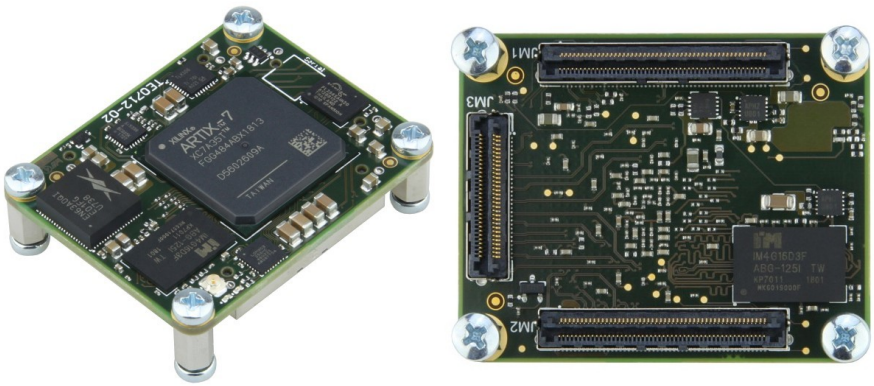
Looking at the other manufacturers, such as Digilent, we find that three types of boards are available. The first type is aimed strictly at the education sector. One can find devices such as the Basys or Nexys. The second type are boards, again, for prototyping. The third are system on module boards, such as the Cmod A7 from Digilent. The purpose of this board is to be easily connectable in a breadboard but can also be integrated into a more complicated design. The design idea implemented in this device is right for us, but the module has a lack of I/O pins we need, and the FPGA used has a very limited internal memory of only 225 KB.

Figure 2.5: Cmod A7 development board



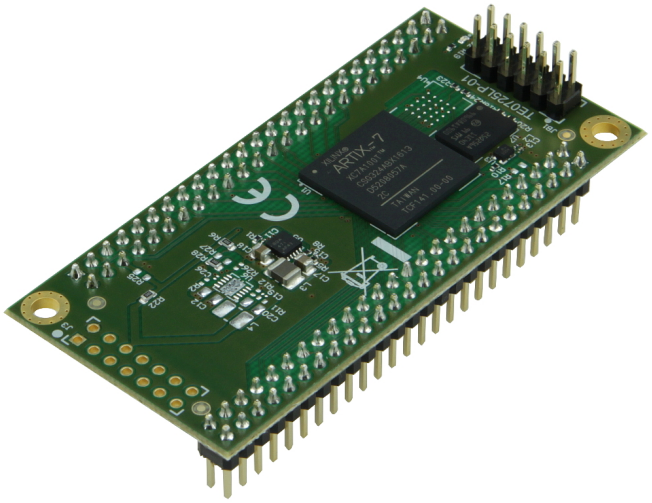
The manufacturer that made the most sense, in the end, was Trenz Electronic. The company manufactures a variety of development kits and a big part of that are systems on module. The modules can be divided into 2 distinct groups. The first group are modules equipped with the high-density Razor beam connectors. The modules are rather small and high density. This makes them suitable for use in larger volume devices than ours. An example of such a module is the TE0712 as seen in the Figure 2.6.

Figure 2.6: TE0712 FPGA module by Trenz Electronic



The next type of module is the TE0725. This module is equipped with standard 2.54 mm pin headers and can be therefore interfaced much more easily. Overall, the feature set of the two Trenz kits is the same. Both have the same FPGA and are equipped with a flash memory device. Both are also sold with an HyperRam chip, which is not going to be used in the design. The difference is only the form factor. For the prototype stage, this variant was used. In the future, should the signal integrity prove to be insufficient or the module too large, the electronics can be redesigned, and the FPGA code will be compatible.

Figure 2.7: TE0725 FPGA module by Trenz Electronic



Chapter 3

Design and implementation

In the first part of this chapter, the rough design of the entire hardware is presented. This is followed by actual design for the FPGA. In the end, the TE0725LP board with the Xilinx Artix-7 XC7A100T-2CSG324C FPGA was incorporated as the core element in the pulsebox. The device was chosen due to the reasonable size of the chip's BRAM and the wish to use a modern FPGA, which 7-Series from Xilinx definitely is. The hardware description is followed with chapters about the sequence assembler software and interface application.

3.1 Design approaches

There were multiple design approaches that could have been taken. Principally, the data independent of encoding or design strategy enters the pulsebox via a single communication interface connected to a personal computer and has to be stored in some way. Then, this data is split into the individual channels and is sent out. A point where the data splits must therefore exist and can be placed in 2 possible locations of the design, before the instruction memory or after.

Splitting the datapath after the memory would lead to a design where a single block of memory has to be read by multiple channels, often at the same time. This means the memory would have to have multiple read ports or be accompanied by some form of arbitration circuitry. Data buffering would also be necessary to make sure all the channels are supplied with data without interruption. The BRAM blocks found in the FPGAs are usually designed in a rigid fashion with a single read and write port set. Implementing multi-port ram with them is therefore impossible and would be accomplishable only with external memory device or a lot of wasted resources due to the necessity to duplicate the data between ports.

This leads us to the arbitration approach. The memory could be organized in larger words, and the data can then be cached. This approach can seem even

more suitable when the reading is purely sequential, and therefore general-purpose cache can be replaced with a simple FIFO. Also, the arbitration circuit could be replaced by an automaton, which would only make sure that none of the FIFOs go empty. This approach would be necessary if an external memory was used instead of on-chip BRAM. This could be, for example, due to the capacity required being too high. Another reason to implement a memory arbiter would be a design where all of the channels need to access the entire memory. The channels in our design access only their portion of memory and do not interfere with each other. This could become a problem if an optimization in the instruction set, which would enable the channels to read in the entire sequence memory, were introduced. Such optimization, for example, could be the ability to define pulse sequences (analogically to functions in standard programming) and permit access to them from all of the channels. For our use case, this does not, therefore, make much sense as a much simpler and direct approach is available

The other approach assigns a separate memory block to each channel. The most time-constrained and highly synchronous part of the design is the channel logic. The channels must be granted access to the instructions at any time, as the failure to do so would lead to skipping in the sequence currently played back. That is unacceptable. This solution takes the uncertainty, which would be introduced with a memory arbiter, outside the time-critical part of the design and into the write part. It is assumed that the operator of the pulsebox will upload the sequence data prior to starting the playback. The write part of the design is then limited only very vaguely and is only required to process the data in a reasonable time. Delays in writing lead only to longer user waiting times, and no data errors are generated. This is the design that will be implemented.

3.2 Design outline

The entire design was done in Verilog HDL utilizing the Xilinx Vivado software. The whole design, except the output serializer, runs on a single clock generated by a PLL in the FPGA. The design is divided into 16 identical channel blocks, which provide a write-only memory-mapped interface to the upper modules. The channels contain a 1024 word memory, which is being read sequentially as the pulse sequence progresses. As mentioned before, to lower the internal frequency and enable parallel processing, FPGAs commonly implement some form of serializer/deserializer logic. The Artix-7 is no different in that regard. This FPGA is equipped with the OSERDESE2 module, called oserdes or serializer from now on, on the majority of the pins. This module can be utilized to create signals that can easily be clocked at 1 GHz. This is precisely what is needed for this application. The serializer can be operated at various widths, 2, 3, 4, 5, 6, 7, 8, 10, or 14 bits to be exact, depending on

the operation mode. A decision had to be made in regards to which serializer width would be selected, 14-bit serializer width was deemed incompatible with the encoding scheme, which is built around 32 bit instruction length and will be discussed in detail in further sections. With 14 bits, one instruction could either contain two output 14-bit words leaving 4 bits empty or one output word and one 6-bit parallel output leaving 12 bits unused. The next logical step would be the 10-bit serializer width. This number was selected as its the highest number for which the count of wasted bits is considered reasonable.

Now that the oserdes width was selected, we can calculate the internal logic frequency required. First, let us pay attention to the clock used to drive the oserdes. Because we are using 10-bit width, the oserdes can be operated only in DDR mode per manufacturer's specification. This means that data is clocked out on every clock edge. The resulting clock must then be half of the desired output frequency, 500 MHz. Let us now turn our focus to the internal clock frequency. It can be calculated by dividing the serializer frequency by the width of the serializer and multiplying by 2 because every clock cycle, 2 bits are transmitted (a single bit on every clock edge). The resulting clock is then precisely 100 MHz. There is one last caveat with using the OSERDESE2. When the internal and output clocks are in sync, the manual states that the delay between the input and output may differ by one clock cycle and can not be controlled, this was solved with the clocks being generated from one PLL and slightly out of phase, so that no edges would coincide. This has proven satisfactory and has yielded deterministic results.

Now with the clocking scheme out of the way, we can concern ourselves with the logic design. To begin with, the communication protocol must be specified as it was the primary influence for the design in Verilog.

3.3 External interface

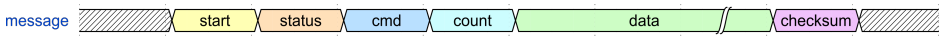
The only communication port beside the pulse outputs and a trigger input is a serial interface. Simplicity was the main concern here, and therefore a UART interface was chosen. To enable a modern PC not equipepd with a hardware RS232 port to communicate with the pulsebox, an interface chip had to be introduced. In the final design, the FT232 interface chip would be used. Using a chip from FTDI has one added benefit too. Besides working as a simple USB COM port interface in line with the standard specification, there are also custom libraries available from the manufacturer. With these libraries, a more advanced and professional interface program could be created. This program would not then have to rely on the user to select a valid port and speed for the device to communicate correctly. In the prototype, a different approach was taken. An ATmega32u4 mounted on an Arduino Pro Micro board was used. Using the default Arduino toolchain and libraries, the chip

simply forwards all the data sent via the USB COM port to one of its UART transceivers and vice versa.

With the lowest level of the communication interface described, let us now proceed to the description of the protocol itself. At first, a very general outline was laid out. The interface is a simple master-slave serial bus and implements a simple stop-and-wait protocol. All the communication is initialized by the PC (master) which, after sending a request, waits for an answer. Because the device can execute all of the commands during the duration of a single baud, it is capable of processing a continuous stream of commands. The simple protocol was selected to improve reliability during development and can be upgraded later. Because the protocol does never send more data than is the size of any buffer along the data path, no flow control is necessary. Implementing flow control would also have to be done on the protocol layer and not with simple Xon and Xoff bytes since the protocol does not work with text but binary data.

Now, we shall concern ourselves with the message format. Initially, the protocol was supposed to be designed such that the internal workings of the pulse box are covered by an abstraction layer. Various commands would be defined to upload the sequence, reset the device, retrieve the channel status, and manipulate the instrument in other ways. This command-oriented format can

Figure 3.1: Command oriented message format



be seen in figure 3.1. The command and answer message format would be the same to keep the protocol more streamlined. First, a start byte is sent, followed by a status byte used by the pulsebox to signal the result of the last operation (similar to a return code in programming). The status byte would always be 0 in the direction from PC to pulsebox. A command word is followed by a block of command-specific general data. This can be, for example, an address and data for writing into a channel sequence memory. The message is terminated with a simple checksum, calculated by summing all message bytes mod 256. While this message format is very versatile, it was soon realized that there is only one necessary and 2 handy commands that can be implemented. Before this realization, a message format specification was written. A reformatted version of the said specification can be found below. The complexity may not seem to be that high, but the possibilities of the protocol are not being utilized to the full extent. For example, the only command utilizing the arguments is the command 0x04.

Message format

#	Byte	Information
1	Synchronization	Start byte: 0x55
2	Status	Status code: 0x00-0xff
3	Function Code	Function code describing the~message: 0x00-0xff
4	Count	Number of bytes of arguments: n = 0x00-0xff
4 + 1	Argument	Argument byte [1]
4 + n	Argument	Argument byte [n]
5 + n	Checksum	Sum of all previous bytes: 0-0xff

Messages towards the~device always have status equal to 0x00

Status codes

Status code	Description
0x00	Ok
0x01	Wrong function code
0x02	Wrong checksum
0x03	Argument out of range

Messages

All messages (except for notification messages) are initiated by the~master (computer).
The~slave (pulse box) only responds.

Code and description	#cmd	#ans	Arguments command	Arguments answer
0x00 No operation	0	0	N/A	N/A
0x01 Retrieve pulse box status	0	3	N/A	0 -> Status bits* 1-2->Channel status
0x02 Stop playback and reset the~device	0	0	N/A	N/A
0x03 Start playback	0	0	N/A	N/A
0x04 Writes a word into command memory**	6	0	0-1 -> 14bit addr 2-5 -> 32bit val	N/A

- * bit 0 = running/halted, others for future use
- ** Reading back now possible as of now due to the~design of the~used memory blocks

The simplified protocol relies on exposing an addressable interface to the computer. The start byte, status word, and checksum are preserved, but the rest of the message has been changed. The status byte is now followed by an address and value having 16 and 32 bits, respectively. The operation (read or write) is now encoded in the status code.

Figure 3.2: Memory oriented message format

The channel memory has been assigned the lowest 16K of the address space. Above that, addresses for the reset and status bits are located. This can be observed in the updated protocol specification below. This is the version implemented.

Command format

#	Byte	Information
0	Synchronization	Start byte: 0x55
1	Command/Response	Command / Response byte
2-3	Address	16 bit address
4-7	Value	32 bit value
8	Checksum	Sum of all previous bytes: 0-0xff

All values are sent MSB first.

Command/Response codes

Status code	Description
0x00	Command: Read
0x01	Command: Write
0x02	Response: OK
0x03	Response: Address not defined
0x04	Response: Address read only
0x05	Response: Address write only
0x06	Response: Bad checksum
0x07	Response: Bad command
0xFF	Response: Logic error

For read operation, results are given in the~"Value" field accompanied by source address.

Address ranges

Start	End	#	Function
0x0000	0x03FF	1024	Channel 0 program
0x0400	0x07FF	1024	Channel 1 program
0x0800	0x0BFF	1024	Channel 2 program
0x0C00	0x0FFF	1024	Channel 3 program
0x1000	0x13FF	1024	Channel 4 program
0x1400	0x17FF	1024	Channel 5 program
0x1800	0x1BFF	1024	Channel 6 program
0x1C00	0x1FFF	1024	Channel 7 program
0x2000	0x23FF	1024	Channel 8 program
0x2400	0x27FF	1024	Channel 9 program
0x2800	0x2BFF	1024	Channel 10 program
0x2C00	0x2FFF	1024	Channel 11 program
0x3000	0x33FF	1024	Channel 12 program
0x3400	0x37FF	1024	Channel 13 program
0x3800	0x3BFF	1024	Channel 14 program
0x3C00	0x3FFF	1024	Channel 15 program
0x4000	0x4000	1	Control word
0x4001	0x4001	1	Status word

Special address spaces

Channel programs (Write only)

Direct access to channel program memory, instructions are executed from the lowest addresses first.

Control word (Read / Write)

Bits	Meaning
0	Reset/Running

Status word (Read only)

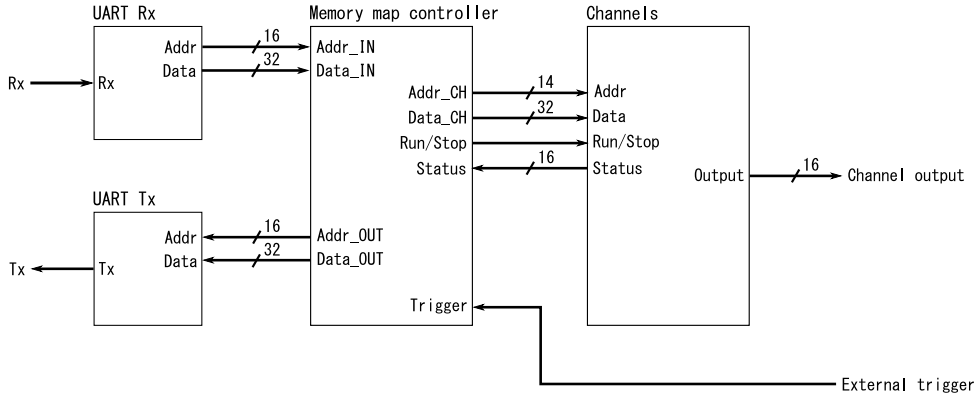
Bits	Meaning
0-15	Done status of individual channels 1->Done, 0->Running

The main advantage of the updated message format is the fixed message size. Before, when the message included a variable-length argument array, the receiving and transmitting circuitry had to contain a dedicated block of memory to work with the arguments. The automaton responsible for the reception of the messages was also made more complex. With the omission of the aforementioned variable-length element, all the control automaton has to do is to read nine bytes of data beginning with 0x55 from the serial line and store them in appropriate registers. Simple as that.

3.4 Top level design

The architecture, as implemented in the prototype, will be discussed in this section. All the design iterations and previous versions have their roots in the design of the communication protocol and the instruction format, and will therefore be discussed in their respective sections.

Figure 3.3: Pulsebox top level simplified schematic



The top module pictured in figure 3.3 can be divided into three distinctive parts. The communication interface located on the left side is responsible for handling the communication between the pulse box and the PC. On the receiving side, the serial data is first read and passed into the protocol parsing automaton in a parallel fashion. Then the address and data words are output to their respective buses, and necessary control signals are asserted. Had the message been parsed successfully, either the read or write signals are asserted. The memory map controller is then tasked with carrying out the command. In the case of a wrong command or bad checksum, the respective error signal is asserted, and the control is handled directly to the message transmitter. The transmitter then responds with a corresponding status code.

Now, let us return to the *Memory map controller* pictured in the centre of the schematic. Its main job is the uploading of the pulse sequences to the channel blocks. When an address between 0x0000 and 0x4000 is to be written, the data is passed into the channel data output alongside the address, and a write signal is asserted. The block also contains a register responsible for starting and stopping the channels. The reset can be controlled via a write to the lowest bit in the address 0x4000, or it can be unset through the external trigger signal. The last function of this block is to pass the channel status to the PC. Each channel provides a single bit output denoting its state. The user can then see if a channel is still running and replaying the programmed sequence or if it has finished (or has been stopped).

The block on the right contains all of the 16 channels and logic responsible for splitting the 14 bit input address bus into sixteen 10 bit busses for each channel. This means all the channel program memory can be accessed through a single unified bus. The details of the channel logic will be discussed in a separate section.

3.4.1 Communication application

To provide a user friendly interface to the pulse box, an interface application written in Python was created. The application enables the user to control the pulsebox from the comfort of a more user-friendly interface. As of the last version, only the command line interface is available, but the intent is to create a GUI application based on Tkinter or some other simple GUI framework. Instead of listing the functions manually, let us have a look at the help message incorporated in the software.

```
pulse_box_app v0.2.0 on linux
usage: app.py [-h] -c COM [-b BAUDRATE] [-p] [-s] [-w] [-r] [-u [UPLOAD]]
              [--start] [--stop] [--speedtest] [-a ADDRESS] [-d DATA]
              [-f FILE] [--nogui] [-v]
optional arguments:
  -h, --help            show this help message and exit
  -c COM, --com COM     set serial, on linux /dev/tty???, on windows COM?
  -b BAUDRATE, --baudrate BAUDRATE
                        baudrate
  -a ADDRESS, --address ADDRESS
                        address for manual R/W, 0x prefix for base 16
  -d DATA, --data DATA  data for manual write, 0x prefix for base 16
  -f FILE, --file FILE  program file for uploading
  --nogui               run without gui
  -v, --verbosity       console logging verbosity
tasks:
  -p, --ping            task: test communication with the~PB
  -s, --status          task: get PB status
  -w, --write           task: manual write to PB address
  -r, --read            task: manual read from PB address
  -u [UPLOAD], --upload [UPLOAD]
                        task: upload channel sequence, add comma delimited
                        number list to specify channels to be upload or "all"
  --start               task: start sequence playback
  --stop                task: stop sequence playback and reset channels
  --speedtest           task: test communication speed
```

The core functions that are utilized by the other ones are read and write. Provided with address and, in the case of write with data, the pulsebox 16 bit address space can be accessed. All checking, including read/write permissions and the availability of the memory location, is done in the FPGA and is signalled back into the application. This ensures that no matter what changes are made to the hardware design, as long as the communication protocol stays the same, the ability to interface with the hardware will remain on at least the most general level.

Now, let us proceed with an overview of the program capabilities. All the ping task performs is a read from address 0x4000. This was chosen arbitrarily. A successful response signifies communication has been established correctly. Start and stop commands, as their names suggest, affect the running state of the hardware. Asserting the external trigger has the same effect as the start function. The upload function is the main reason for the program. The input is a binary file that will be uploaded to the pulsebox precisely as is, with the only difference being the remapping of each 4 bytes into a single 32 bit word used in the program memory.

The last function, speedtest, was created because of the rather slow communication speeds noticed during the development. The speeds measured were about 32 Kbit/s, which is rather slow for UART with 2 Mbit baudrate. This function simply sends 4096 ping commands, and from the time the communication takes, it can calculate the real speed and estimate the time necessary

to upload the whole sequence to all of the channels. While this proved useful, the reason for the significant slowdown was not explained yet. To rule out any slowdowns caused by the PySerial library used, a simple, absolutely barebone program written in plain C was created. This program has shown exactly the same results. This meant the problem is somewhere in the OS or in the hardware. An oscilloscope was hooked up to the serial output of the interface chip, and it was noticed that there are wide gaps between the bytes. This is probably caused by the slow Atmega32u4 processor, not being able to process the data fast enough. Also, the communication with the USB device directly (when FT232 is used) instead of relying on a virtual COM port driver could improve the speed. Further investigation was not done in this matter as the speeds were satisfactory for this prototype stage.

3.5 Instruction specification and assembler

The inner workings of each channel were crafted around an instruction specification, which was done very early in the design process. As it was mentioned in the requirements and specifications chapter, there are multiple ways of encoding the sequences, and a combined approach was implemented.

At first, it was decided that the instructions would have a constant width, which was decided to be 32 bits. 8 bits would be way too small as the serializer width is higher than that. 16 bits seem plausible, but considering the attenuator data is 6 bits wide, there are no bits left for the differentiation between instruction types. 64 bits seemed like an overkill. Encoding up to 6 chunks without attenuators into one instruction seemed nice, but using this amount of bits for storing time interval seemed rather wasteful. The possibility of needing a sequence lasting just shy of 6 years (the longest possible interval that can be encoded with 54 bits) is negligible.

Now that we have set the instruction length, let us proceed with the enumeration of all the parts that have to be incorporated into the sequence. First, a small number of bits have to be dedicated to the identification of each instruction. Then, for the timed instruction, an output value, attenuator value, and the time have to be added. Incorporating the value into the attenuator (such that >0 means 1 and 0 means 0) would not work because the user might want to set the attenuator in advance without setting the output. For the raw instruction format, the bit patterns and attenuator must be added. After a few iterations, the protocol was defined per following specification.

Instruction encoding
====================

End instruction

Halts the~pulse box channel from reading any further instructions. Used as the last instruction in the~sequence.

Name	Bits	Values	Description
Null	0-31	0	Ends program

[0,1] Long instruction

This instruction sets output (value based on opcode) for time specified in *"Time"* part of the~instruction. Unit of time is 10 ns.

Name	Bits	Values	Description
OpCode	0-1	[0,1]	Select instruction. Value also denotes output
Time	2-25	*	Time the~output will be held for
Attenuator	26-31	*	Attenuator setting

[2] Precise instruction w/ attenuator

This instruction serializes the~bit pattern/s specified in *"Pattern n"* parts of the~instruction. Single bit sets the~output for 1 ns. Length of the~pattern may be selected by bit 22 to 10 or 20 bits.

Name	Bits	Values	Description
OpCode	0-1	[2]	Select instruction.
Pattern 1	2-11	*	Output pattern 1
Pattern 2	12-21	*	Output pattern 2
P2 Enable	22	*	Enable pattern 2 (Switch 10 or 20 bit pattern)
N/A	23-25	N/A	Not used
Attenuator	26-31	*	Attenuator setting

[3] Precise instruction w/o attenuator

This instruction serializes the~bit patters specified in *"Pattern n"* parts of the~instruction. Single bit sets the~output for 1 ns. The~length is always 30 bits. The~attenuator value is not affected by this instruction.

Name	Bits	Values	Description
OpCode	0-1	[3]	Select instruction.
Pattern 1	2-11	*	Output pattern 1
Pattern 2	12-21	*	Output pattern 2
Pattern 3	22-31	*	Output pattern 3

In the final format, the first two bits have two functions. The first is the identification of the instruction, and the second is the output value during the timed instruction. This way, all the four possible values are used, and a separate bit need not be used to set the timed instruction output. The timed instruction then follows the previously set outline. A 6 bit attenuator value is placed at the end of the instruction, and the 24 remaining bits in between are left to the time. This means the longest time sequence coverable by one instruction is calculated as follows:

$$M = 2^{24} \cdot 10\text{ns} = 167772160\text{ns} = 167.77216\text{ms}$$

This number is more than satisfactory for this application. Continuing with the raw instructions, it was soon evident that some of the bits would be inevitably wasted. This was partly mitigated by the split into two separate instructions. One enabling pattern lengths of 10 and 20 bits with attenuator, the second encodes 30 bit sequences at the cost of no attenuator setting due to all the 30 remaining bits being used. In the former instruction, the attenuator value is placed at the end of the sequence, in the same place as in the timed instruction. To distinguish between 10 and 20 bit pattern lengths, bit 22 is used.

While the instructions are not complicated, and the program could be assembled by hand, the effectiveness of such an approach is disputable. To prevent mistakes and countless hours of debugging resulting in considerable frustration of the user, an assembler program was created.

3.5.1 Assembler and programming language

The assembler was created in the C++ programming language and provides a basic command line interface. The version as of the making of this document is capable of ingesting custom, line-oriented programming language and output the bytecode, which can be directly uploaded to the program memory. The syntax was designed to mimic somewhat the syntax used by the assembly languages used in processors.

The program recognizes two instructions, *tim* and *raw*. Each of the instructions recognizes a varying list of comma-separated arguments. Let us proceed with an extract from the Instruction specification.

```
Instructions
============

The~instructions are in format: NAME comma,separated,argument,list

Used terms
----------

uint: C formatted unsigned integer with maximal specified size (octal format not supported)

tim - Long (timed) instruction
------------------------------

| Field | Name          | Values       | Description                                 |
| ----- |:-------------:|:------------:| ------------------------------------------- |
| 1     | Output value  | [0,1]        | Output during the~instruction execution     |
| 2     | Time          | 24bit uint   | Instruction execution time                  |
| 3     | Attenuator    | 6bit uint    | Attenuator value during execution           |

raw - Precise (raw) instruction
-------------------------------

10/20 bit variant with attenuator

| Field | Name           | Values        | Description                                 |
| ----- |:--------------:|:-------------:| ------------------------------------------- |
| 1     | # of bits      | [10,20]       | Number of following bits                    |
| 2     | Binary pattern | 10/20bit uint | Next 10/20 outputs                          |
| 3     | Attenuator     | 6bit uint     | Attenuator value during execution           |

30bit variant w/o attenuator

| Field | Name           | Values        | Description                                 |
| ----- |:--------------:|:-------------:| ------------------------------------------- |
| 1     | # of bits      | 30            | Number of following bits                    |
| 2     | Binary pattern | 30bit uint    | Next 30 outputs                             |
```

The program is written in C++14 consistent with OOP principles. Each instruction is represented by a dedicated class derived from a common base class. The software is therefore ready for the possible addition of new instructions. The software comes with a makefile capable of building and installing the software. The binary is installed in /usr/bin directory, which is a location consistent with the Unix filesystem structure. Let us now see an example source be compiled into corresponding bytecode.

```
$>cat example.pbasm
    tim     1,0xaa,0x05
    tim     0,0xaa,0x05
    raw     10,0b1000000101,0x05
    raw     20,0b10000001011000001101,0x05
    raw     30,0b10000001011000001101000011101
$>pboxasm -vv example.pbasm -o bytecode.bin
pboxasm version 0.1.0
[INFO]: Reading source from "example.pbasm"
[DBG]: line 0 = tim 1,0xaa,0x05
[DBG]: line 1 = tim 0,0xaa,0x05
[DBG]: line 2 = raw 10,0b1000000101,0x05
[DBG]: line 3 = raw 20,0b10000001011000001101,0x05
[DBG]: line 4 = raw 30,0b10000001011000001101000011101
[INFO]: Writing binary to "bytecode.bin"
[DBG]: line 0 -> 0x140002a9 bin:|000101|00000000000000010101010|01|
[DBG]: line 1 -> 0x140002a8 bin:|000101|00000000000000010101010|00|
[DBG]: line 2 -> 0x14000816 bin:|000101|0000|0000000000|1000000101|10|
[DBG]: line 3 -> 0x14605836 bin:|000101|0001|1000000101|1000001101|10|
[DBG]: line 4 -> 0x8160d877 bin:|100000|0101|1000001101|1000011101|11|
[INFO]: All done!
$>hd bytecode.bin
00000000  14 00 02 a9 14 00 02 a8  14 00 08 16 14 60 58 36  |.............`X6|
00000010  81 60 d8 77                                       |.`.w|
```

First, the listing of the source can be seen. The compiler is then called with increased output verbosity to show all the steps during compilation. The source is parsed and then compiled into binary form. A hexdump of the output binary follows. One can observe the relationship between the hexdump and the compiler output. Each of the 32 bit chunks corresponds to individual instructions. The encoding has been made big-endian to enable the user (and mainly the programmer during the development of this tool) to easily distinguish each instruction in the hexadecimal listing. The byteorder of the messages transmitted on the serial line has been derived from this decision as well.

3.6 Pulse channel design

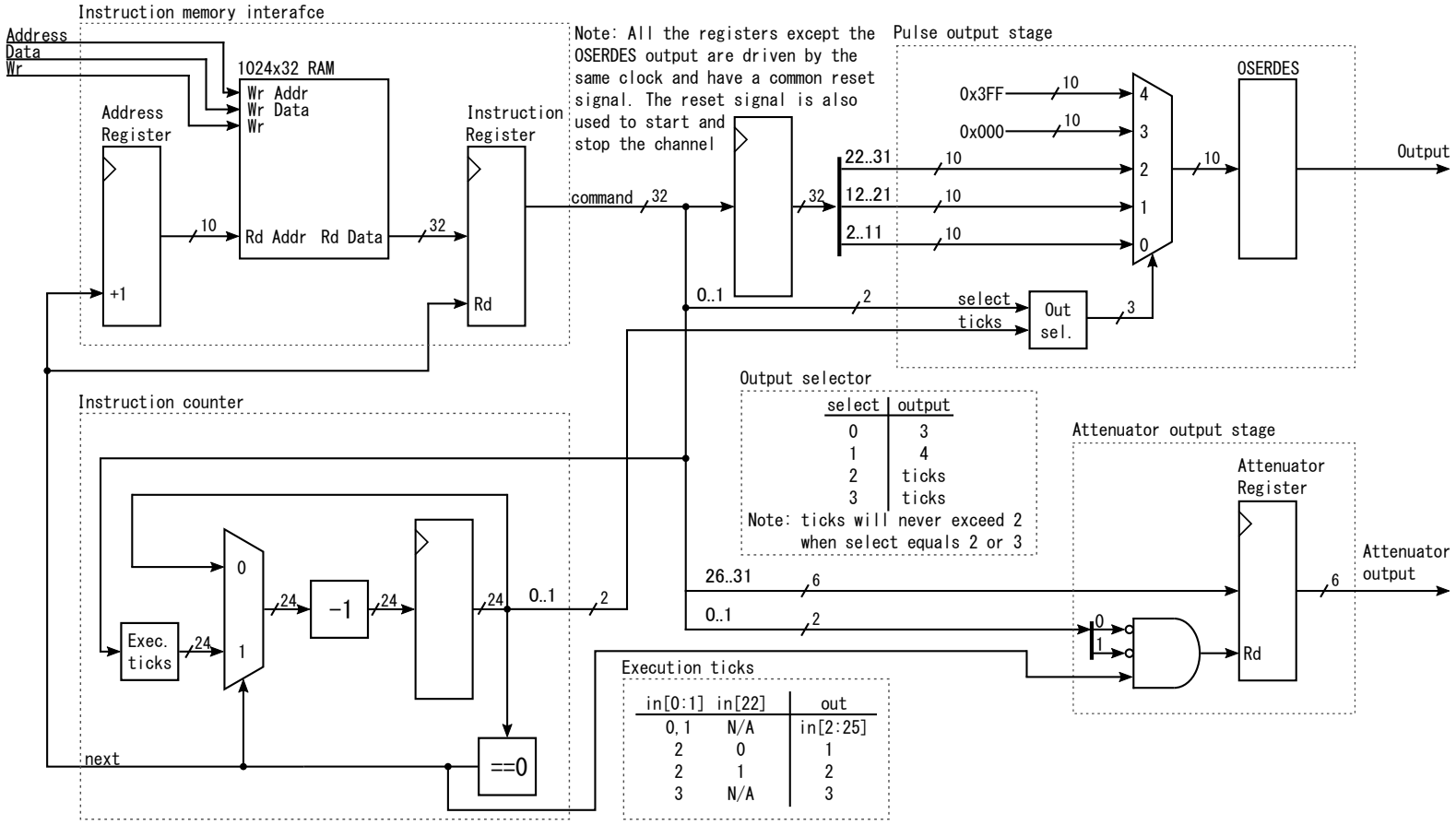
The architecture of the pulse channel has been heavily influenced by all the research, followed by a substantial amount of trial and error. In the following schematic, the different parts of the channel can be observed. In the top left corner, the write interface of the internal RAM can be observed. It is available externally and can be used at any time, independent of the running state. Besides the reset signal, this is the only interface controlling the channel.

After the deassertion of the reset signal, the first instruction is loaded. The possible issue of loading 0x00000000 from the instruction register right after reset is mitigated by the fact, that even though the channel is stopped and address register is zeroed, the instruction register, instead of being ze-

roed as well, keeps storing the provided value. Therefore when the reset has been asserted for at least 2 clock cycles before, the instruction register will contain the instruction at the start address. With the circuit being started, let us now turn our focus to the timing part of the circuit. When the *next* signal is asserted, the address register is advanced, and the instruction register stores a new value. The counter also stores a new time interval to be counted down. The interval is either constant (in the case of raw instructions) or taken from the instruction (for timed instructions). The counter then counts down the interval, and when the value reaches 0, *next* is asserted, and the cycle repeats.

New data reaches the serializer on every clock cycle; this is assured by an output multiplexer controlled by a block of dedicated combinatorial logic. At idle or in the reset state, the serializer is set to output only zeroes. Because the data inside the serializer is processed in some form of a pipeline, there is a notable delay between the internal enable signal being asserted and the first data leaving the module. This delay, presumably to set some internal states and to fill the pipeline, is longer than the delay between the data input and output when the module is running. To make use of this, the module is never stopped, and only the data on the input is changed. When the channel is stopped, the serializer is merely passed zeroes.

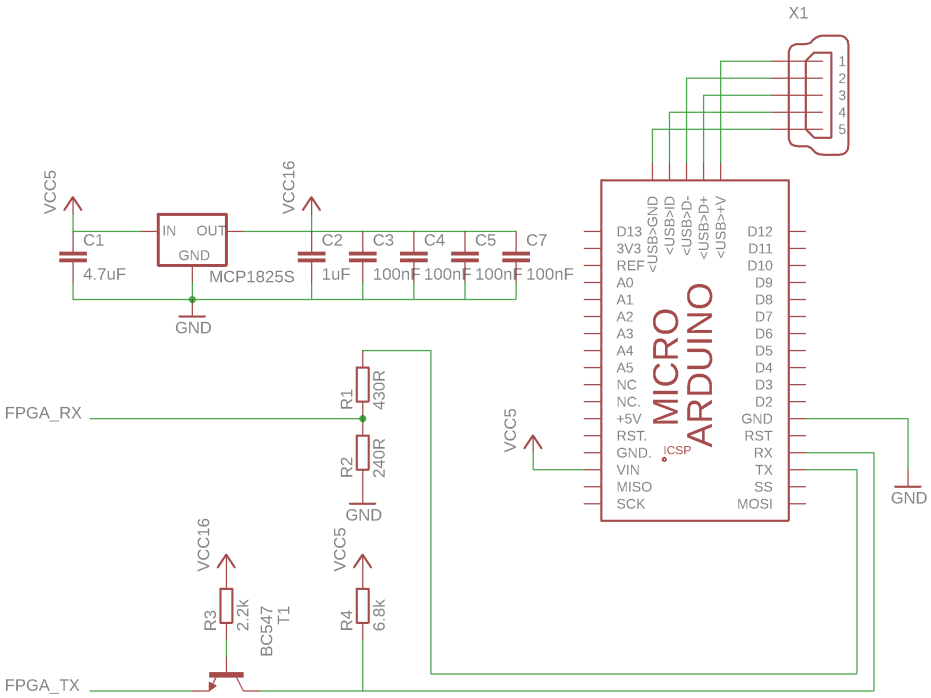
Figure 3.4: Pulsebox channel schematic



3.7 Electronics design

Initially, the development of the instrument was done on a Basys3 board from Digilent. Able to be powered from USB and equipped with a suitable FT232 chip from FTDI, it enabled the development of the portion of the design responsible for the communication. It was also used to test the channel using the integrated logic analyzer.

Figure 3.5: Prototype motherboard schematic



To implement the device on the daughterboard from Trenz, which was to be integrated into the final system, a prototype motherboard had to be created. The board contains the sockets for the daughterboard, Arduino Pro Micro as the UART interface and power source, linear regulator converting the USB 5V power into 1.8V used by the FPGA, and some level shifting circuitry for the UART. The programming interface is supplied by the XUP USB-JTAG cable from Digilent connected directly to the daughterboard. To access the outputs from the FPGA, an oscilloscope probe was directly attached to the daughterboard pins. LVDS was selected as the I/O standard for the pulse outputs as it was readily available in the FPGA and more than capable of desired output speeds.

The board was built by hand on a piece of protoboard, also sometimes called perfboard or Veroboard. Because the construction was rather flimsy and prone to shorts when misplaced on conductive items commonly found on

a workbench, the prototype was equipped with a 3D printed backing place inspired by the ones fitted to Basys3 boards at FIT CTU.

Chapter 4

Simulation and testing

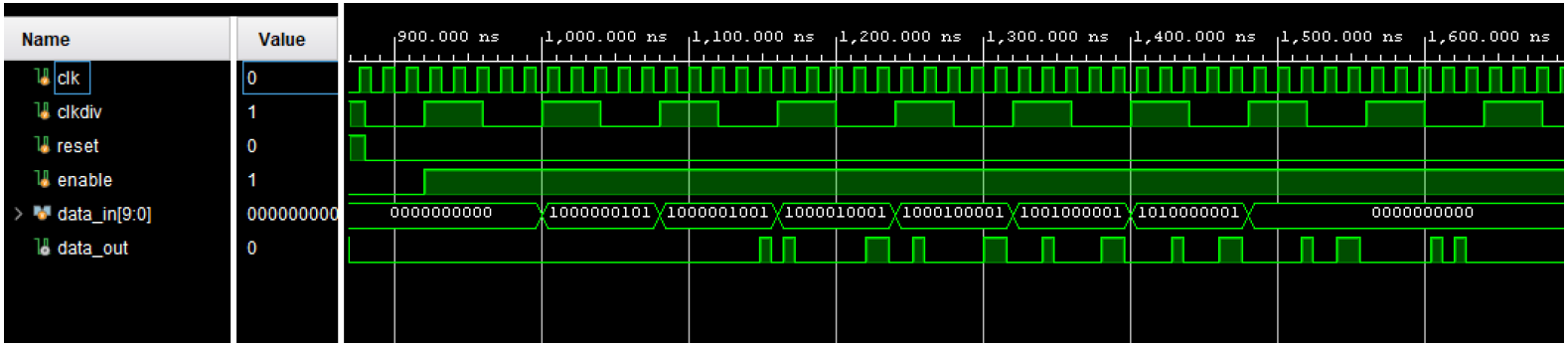
During the development of the pulsebox, all the modules utilized inside the channel module were tested and simulated. This resulted in the discovery of numerous bugs and mistakes early in the process and simplified the assembly and testing of the entire channel. Let us start from the smallest components and work our way up to the whole channel.

4.1 Module simulations

Serializer module

First, the serializer is observed. In the beginning, the reset signal is deasserted, and the enable signal is asserted. During normal operation, the enable signal is always active. It was found empirically that this does not affect the function of the serializer. After a delay, the data is seen on the output. It can be observed that the data is clocked out LSB first. The output data is synchronous with the faster *clk*, and the input is synchronous with the slower *clkdiv* clock.

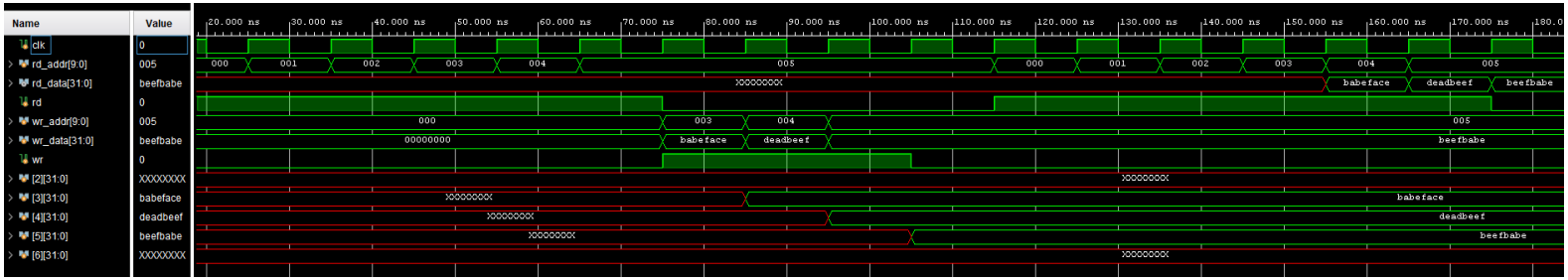
Figure 4.1: Serializer simulation



Memory module

The memory simulation is rather simple. The last five signals display the contents of select memory addressed that will be used in the simulation. At first, all the values are correctly undefined. This is further checked by reading the addresses at the beginning. After that, the addresses 1, 2, and 3 are written. The specified locations in the memory now have valid values stored. The memory is read, and correct values emerge with a single clock delay from the *rd_data* port. The undefined values in red are related to simulation only. When implemented in hardware, the memory will most probably contain all zeroes by default, as no form of initialization is performed.

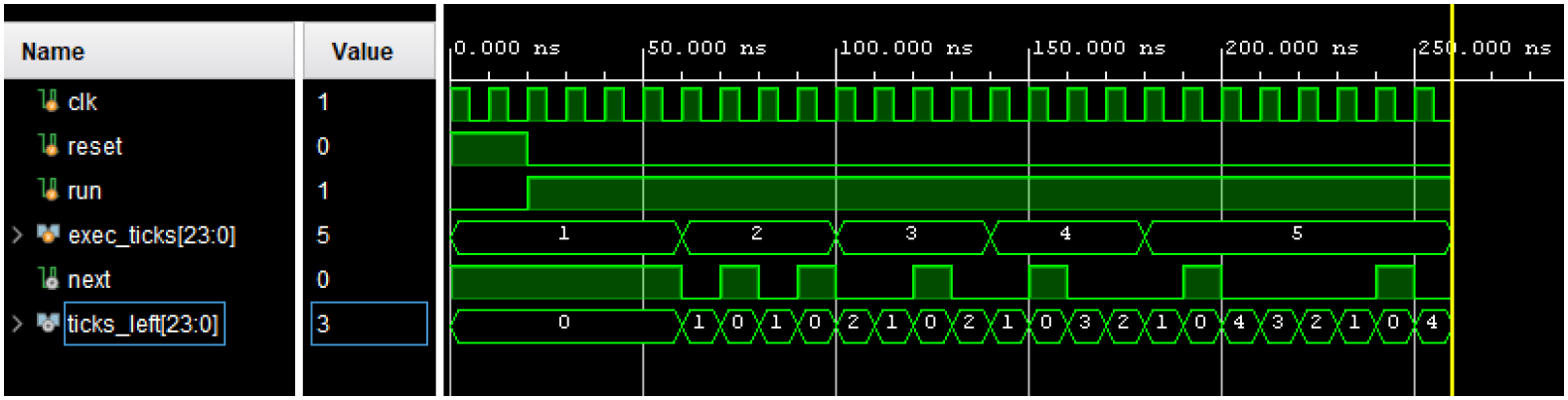
Figure 4.2: Channel memory simulation



Channel counter module

The counter is the most important but also one of the simpler parts of the channel. Its job is to count down until the stored value is zero, generate a signal which will advance the whole circuit, and load a new value from the instruction. This can be seen in the simulation well. At first, the reset signal is deasserted, and the run signal is asserted. In the final design, they are tied together with a simple not gate. Because the value stored is zero when the counter is held reset, after the deassertion, a new value is immediately loaded. Then the cycle carries on as described.

Figure 4.3: Channel counter simulation



4.2 Channel simulation

To test the functionality of the channel, the following sequence was devised. The sequence is divided into 4 distinct parts, always separated by a 10 ns gap during which the output is set to 1. The example was designed to mimic the transmission of the capital letter U (0x55 in hex) at varying baudrates. The value was chosen because of the alternating bit pattern. The value 0xAA could have been chosen as well, but that would make the transmission symmetric, which was undesirable for clarity. A signal with 3 ns bit width is followed by 2 ns and one 1 ns signal. The last is signal with 10ns bit lengths.

```
tim     1,1,0
raw     30,0b111000000111000111000111000111
tim     1,4,0
raw     20,0b11000011001100110011,0
tim     1,4,0
raw     10,0b1001010101,0
tim     1,4,0
tim     0,2,0
tim     1,1,0
tim     0,1,0
tim     1,1,0
tim     0,1,0
tim     1,1,0
tim     0,1,0
tim     1,2,0
tim     1,8,0
```

The sequence makes use of all of the instructions available in the pulse box. All of the figures in this section are part of one simulation. First, the sequence must be loaded into the channel. This can be seen in the Figure 4.4.

Figure 4.4: Pulse channel setup simulation

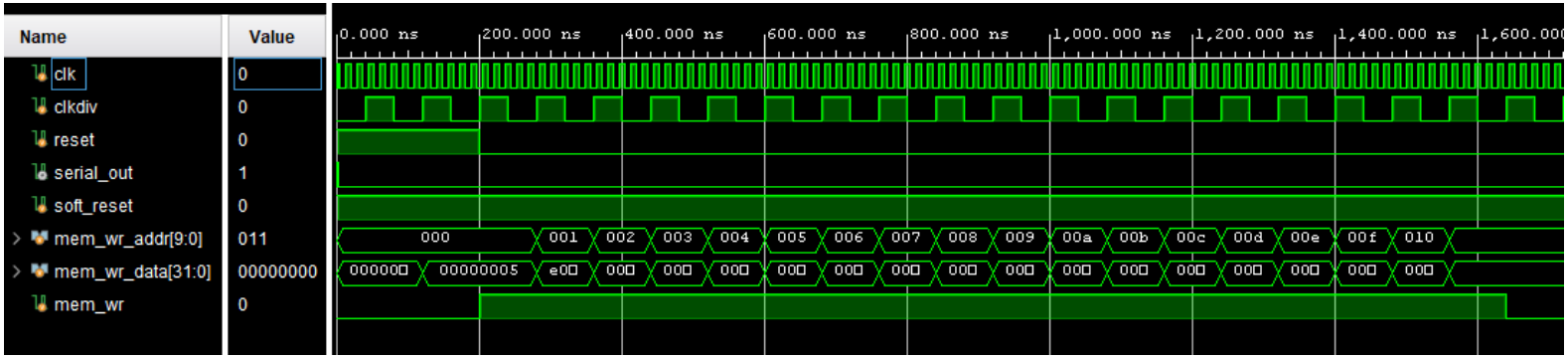
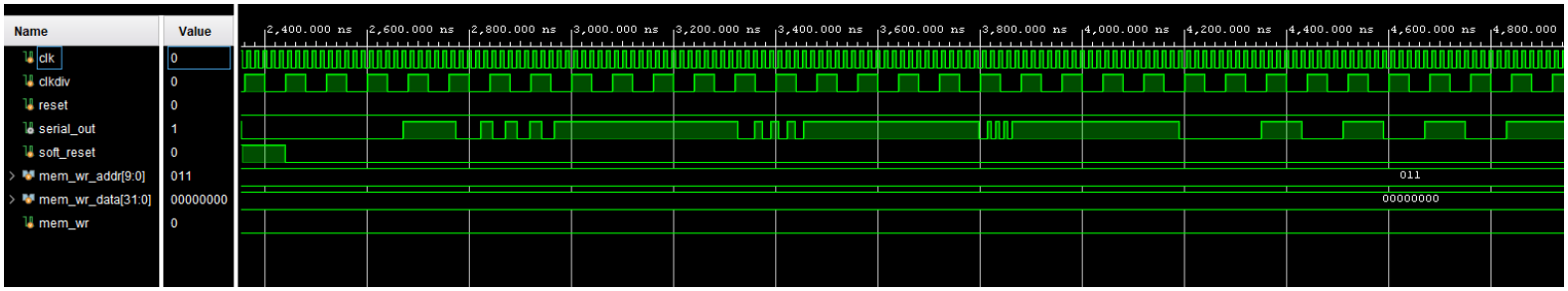


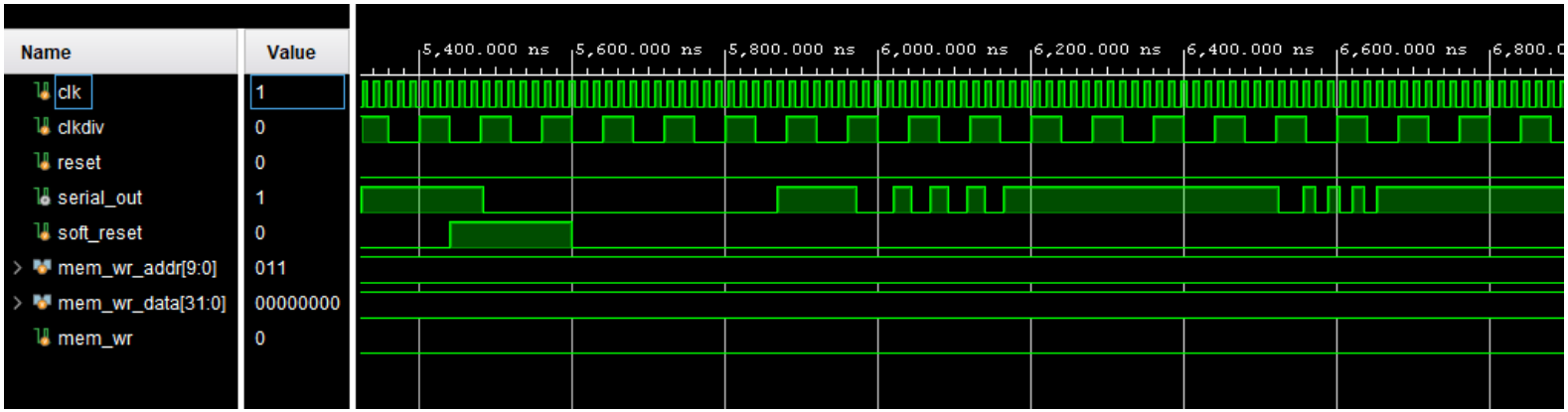
Figure 4.5: Pulse channel running simulation



The logic is held reset while the address and data are presented on their respective buses. The write signal is asserted, and the memory stores the value. This has been shown before in the memory simulation. Next, the pulses are played back.

After the deassertion of the reset signal, the channel springs to life, after a delay caused by the serializer, the data can be seen on the output. The programmed pattern can be seen in the Figure 4.5. To demonstrate the channel can be restarted by the asserion of *soft_reset* signal, the Figure 4.6 is provided.

Figure 4.6: Pulse channel restart simulation

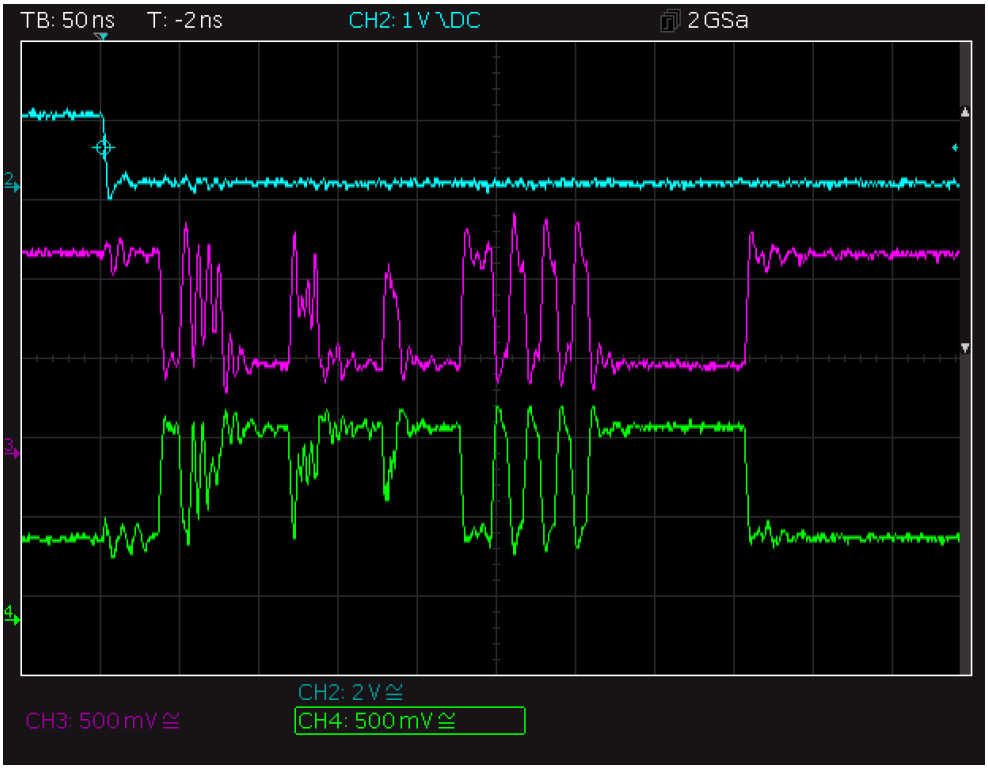


After the reset is deasserted again, the channel starts playing the sequence from the start in the same fashion as before.

4.3 Hardware testing

Seeing the channel perform correctly on a behavioral simulation is very useful, but actually implementing the hardware in the FPGA and executing the sequence is much more valuable. In the oscilloscope capture in Figure 4.7, 3 signals are present. The cyan signal at the top is the reset, the green, and purple signals are the positive and negative components of the differential output.

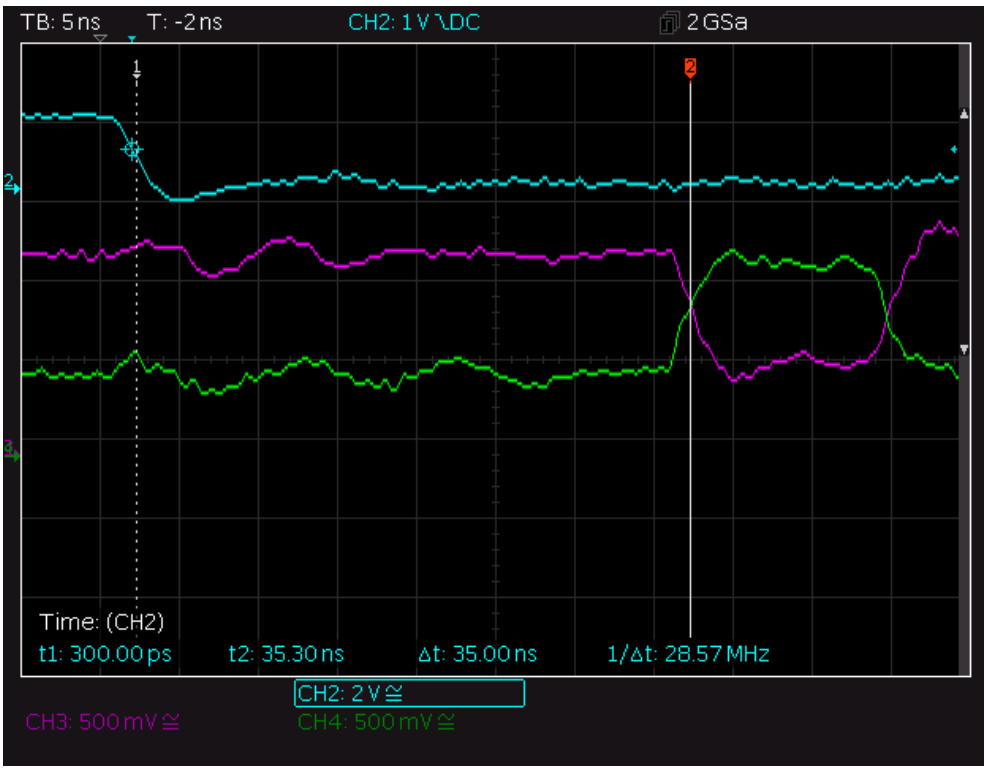
Figure 4.7: Output waveform oscilloscope capture



After the deassertion of the reset signal, the output waveform appears after a delay. The delay as seen in Figure 4.8 is about 35 ns. The delay, as seen in the simulation is about 16 periods of the fast clock. This equals to about 32 ns. The slight difference may be attributed to the possible time imprecision of basic behavioral simulation or general lack of measuring precision. The trace length may affect the time difference only minimally as the trace lengths on the PCB vary maximally in the order of centimeters, and the propagation delay will, therefore, be in the order of low hundreds of picoseconds.

Although the signal is heavily distorted due to the use of an oscilloscope with lower than necessary bandwidth,

Figure 4.8: Output waveform start delay oscilloscope capture



Conclusion

The thesis was focused on the design of a pulse train generator utilizing an FPGA. For the creation of a design that would be of use in a real scientific environment, the research of existing solutions and available technology had to be done. This research resulted in the selection of appropriate development platform and paved the way for the final design. To make the device to usable for end users, a communication application written in Python and assembler written in C++ were introduced.

The result is a functional prototype that can be interfaced using a custom protocol implemented over the standard serial interface and expandable to use a more professional interface incorporating FTDI D2XX drivers. The prototype does not yet have the attenuators added. The device can be programmed with binaries created by a custom assembler. The source files contain a straightforward line organized assembly language made specifically for the project. In the end, all the requirements outlined in the requirements section were met. The device is capable of outputting pulse sequences clocked at 1 GHz using specialized instruction formatting. The interface application is able to upload program binaries, control the reset state, and read the status of each channel.

The development was faced with multiple issues, mainly in regard to the electronic design. When the design was to be tested on the FPGA board that would be in the final device, a temporary motherboard had to be fabricated to provide the FPGA with power and all the necessary interface. Another batch of issues came from using Xilinx Vivado, which I never used before. The rather steep learning curve has caused numerous delays, mainly at the start of the design process.

In the future, the focus should be on creating the final electronics. The current state is that the device can communicate, and the outputs can be measured with an oscilloscope. The device is not yet ready for full incorporation into the scientific process. Another part would be the design of the GUI for the communication application. Currently, only the console interface is avail-

able. As the researchers in other fields that IT can not be expected to be well versed in the use of the command line, the creation of a GUI would be reasonable. Lastly, writing the programs for the device is not a simple task. Having a graphical program where one could draw the output waveforms and receive the compiled binary that can be uploaded would make great sense. This functionality could also be incorporated into the communication application.

Working on instruments that would be used in scientific research and would indirectly help us understand the inner workings of our universe brings me great joy. Advancing the boundaries of human knowledge has always been one of the major forces that propelled me forward with my education and subsequent career. Designing this very device may be the first step in that pursuit, and I can only imagine where it will lead me.

Bibliography

[1] HOŠÁK, Radim, JEŽEK, Miroslav, "Arbitrary digital pulse sequence generator with delay-loop timing," *Review of Scientific Instruments*, vol. 89, no. 4, 2018.

[2] Arduino, "Arduino due." [online], urlhttps://store.arduino.cc/arduino-due). accessed 2020/05/01.

[3] HAYLOCK, Ben, LENZINI, Francesco, KASTURE, Sachin, FISCHER, Paul, STREED, Erik W., LOBINO, Miroko, "Nine-channel mid-power bipolar pulse generator based on a field programmable gate array," *Review of Scientific Instruments*, vol. 87, no. 5, 2016.

[4] Tektronix, "Hfs 9000 stimulus system user manual." [online], https://www.tek.com/hfs9003-manual/hfs9000-user-manual. accessed 2020/05/01.

[5] Keysight, "81134A Pulse Pattern Generator, 3.35 GHz, dual-channel." [online], `https://www.keysight.com/en/pd-1000004569%3Aepsg%3Apro-pn-81134A/pulse-pattern-generator-335-ghz-dual-channel)`. accessed 2020/05/02.

[6] Hardware Bee, "List of fpga companies." [online], `https://hardwarebee.com/list-fpga-companies/`. accessed 2020/05/02.

[7] Intel, "Intel® fpgas and programmable devices." [online], `https://www.intel.com/content/www/us/en/products/programmable/fpga.html`. accessed 2020/05/013.

[8] Intel, "Altera LVDS SERDES IP Core UserGuide." [online], `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/archives/ug_altera_lvds-17-0.pdf`. accessed 2020/05/14.

[9] Intel, "Stratix V GX Device Family - TR5 FPGA Development Kit." [online], https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=158&No=1001). accessed 2020/05/22.

[10] FMCHUB, "VITA 57 FPGA Mezzanine Card (FMC) SIGNALS AND PINOUT OF HIGH-PIN COUNT (HPC) AND LOW-PIN COUNT (LPC) CONNECTORS." [online], https://fmchub.github.io/appendix/VITA57_FMC_HPC_LPC_SIGNALS_AND_PINOUT.html. accessed 2020/05/28.

Appendix A

Acronyms

MCU Microcontroller unit

FPGA Field programmable gate array

NOP No operation (processor instruction)

CRT Cathode ray tube

MSB Most significant bit

BRAM Block RAM

HDL Hardware description

GUI Graphical user interface

BGA Ball grid array (chip package type)

LPC Low pin count (connector)

HPC High pin count (connector)

FMC FPGA Mezzanine Card

RAM Random access memory

LSB Least significant bit

MSB Most significant bit

Appendix B

Contents of enclosed CD

- `README.txt` ..... file with CD contents description
- `app` ..... directory with application souce code
  - `speedtest` ..... C based speedtest program
- `compiler` ..... assembler source code and compiled binary
  - `build` ..... assembler build directory
  - `example` ..... pulse box program example directory
  - `include` ..... assembler header directory
  - `source` ..... assembler source directory
  - `Makefile` ..... assembler makefile
  - `README.txt` ..... assembler readme, listing of 3rd party libraries
- `documentation` ..... documentation directory
  - `xilinx_docs` ..... Xilinx document directory
  - `communication.md` ..... communication protocol specification
  - `instructions.md` ..... instruction specification
  - `trenz_kit_pinout.pdf` ..... development kit pinout
- `source` ..... pulse box Verilog implementation and simulation source
  - `ip` ..... IP core source directory
  - `simulation` ..... simulation source directory
- `thesis` ..... thesis LaTeX source code and images
- `BP_Nevrela_Vojtech_2020.pdf` ..... thesis text in PDF format