FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of master's thesis

Title:	Multichannel USB time-to-digital interface
Student:	Bc. Vojtěch Nevřela
Supervisor:	Ing. Jaroslav Borecký, Ph.D.
Study program:	Informatics
Branch / specialization:	Design and Programming of Embedded Systems
Department:	Department of Digital Design
Validity:	until the end of summer semester 2024/2025

Instructions

Design and implement a multichannel time-to-digital converter connected to a PC via USB3 interface based on the TDC-GPX2 chip. The chip samples the inputs and generates a timestamp for each pulse detected. The timestamps are aggregated in a FPGA and sent to PC, where an application saves them to disk. The FPGA is also responsible for sending configuration from PC to all the other chips (TDC, reference DAC). The device should provide at least 4 channels and the interface between the TDC and FPGA should run at least at 100 MHz SDR. Time synchronization should be based on an externally provided clock signal.

Electronically approved by prof. Ing. Hana Kubátová, CSc. on 21 September 2023 in Prague.

Master's thesis

MULTICHANNEL USB TIME-TO-DIGITAL INTERFACE

Bc. Vojtěch Nevřela

Faculty of Information Technology
Department of Digital Design
Supervisor: Ing. Jaroslav Borecký, Ph.D.
January 10, 2024

Citation of this thesis: Nevřela Vojtěch. *Programmable generator of synchronous pulse sequences.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

List of code listings

I would like to express my thanks for great cooperation and help to Petr Obšil, who the device is designed for and who will use it in his research. Next, I wish to express my thanks to my friend 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, I would like to express my thanks to my partner for her unending patience.

Declaration

In Prague on January 10, 2024

Abstract

The objective of this thesis is the development of a four channel time to digital converter device connected to a PC via USB 3.0 interface. The implementation was done in SystemVerilog HDL, tested using Xilinx Vivado ILA, and deployed on an FPGA SoM connected to custom hardware. The resulting device is capable of ingesting samples at 2.5 MS/s per channel and storing them in a file on a PC.

Keywords FPGA, time-to-digital, USB

Abstrakt

Předmětem práce je vývoj 4 kanálového time to digital konvertoru připojeného k PC skrze USB 3.0 rozhraní. Implementace je vyhotovena v SystemVerilogu, otestována za pomoci Xilinx Vivado ILA a nasazena na FPGA SoM který je připojen k vlastnímu hardwaru. Výsledné zařízení je schopno přijímat vzorky s kadencí 2.5 MS/s na kanál a ukládat je do souboru na PC.

Klíčová slova FPGA, time-to-digital, USB

Abbreviations

ADC	analogue to digital converter
BRAM	block RAM (memory type in FPGA)
DDR	double data rate
FIFO	first in first out (buffer)
GIL	global interpreter lock
GUI	graphical user interface
I/O	input/output
JTAG	Joint Test Action Group (interface standard)
LRU	least recently used
MRCC	multi region clock capable (pin)
PLL	phase-locked loop (clock signal generation)
RAII	resource acquisition is initialization (C++ technique)
S	sample (unit)
SDR	single data rate
SoM	system on module
SRCC	single region clock capable (pin)
TDC	time to digital converter
TVC	time to voltage converter

Introduction

Many fields need precise time measurement facilities to perform research. Fields ranging from optics, optoelectronics, quantum physics, telecommunication, and other high-tech areas of research and engineering benefit from such implements. The experiments and measurements performed often generate electric pulses from various sources, which have to be sampled precisely such that an analysis can be performed on the arrival times and valuable statistical data can be extracted. This is performed using time-to-digital converter (TDC) instruments. The issue with commercially available TDCs is that they are either not readily available in a user-friendly package with all the supporting software (usually embeddable chips) or that they are too expensive (ready-made instruments).

There exists a plethora of readily available chips capable of time-tagging incoming events. With sufficient supporting hardware and software, a user-friendly TDC can be created, which is also the primary goal of this thesis.

A TDC device will be successfully developed and tested alongside accompanying software.

Chapter 1

Theory of time-to-digital converter operation

In computer engineering, time to digital converters (TDC) represent a critical component in measuring and analysing time intervals. TDCs play a pivotal role in diverse applications, ranging from high-precision time-of-flight measurements in radar and lidar systems to time-based signal processing in communication networks. The fundamental objective of a TDC is to transform analogue time intervals into digital representations, facilitating precise temporal measurements in electronic systems.

The idea of a time to digital converter is to capture occurrences of events in time and to generate corresponding timestamps. This behaviour enables the processing of said pulses to be moved into the digital domain as well as allowing the events to be processed after collection instead of processing them on-the-fly.

The history of TDCs can be traced back to the mid-20th century when researchers began exploring methods to digitize time for scientific and industrial applications. Early implementations involved analogue techniques, but the advent of digital technology in the latter half of the century spurred the development of more accurate and reliable digital TDCs. As semiconductor technologies advanced, TDCs evolved to meet the increasing demand for higher precision and resolution in various time-sensitive applications.

In contemporary computer engineering, TDCs have become integral to numerous cutting-edge technologies, particularly in medical imaging, telecommunications, and autonomous systems. The demand for enhanced temporal resolution and accuracy has led to ongoing research and development in the TDC domain.

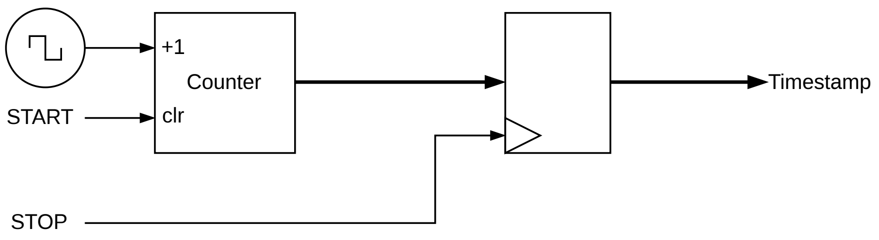
There exist many approaches for generating time stamps from events, some of which will be discussed further. They can be divided into 2 main groups/types of measurement which we will call coarse and fine.

For the purpose of this chapter, we assume that an event occurrence is denoted by a 0 to 1 transition.

1.1 Coarse measurement

In our context, by saying coarse, we mean that the time resolution of the TDC is no better than the period of the clock signal used for sampling the input signal. A simple method comes to mind. A monotonic counter incrementing each clock cycle is used. Upon the detection of an event, the value currently stored in the register is latched, and its output is the next timestamp, as shown in Figure 1.1. Should the difference between two events be detected, the counter can be cleared upon the detection of the *START* event.

Figure 1.1 Coarse measurement example schematic



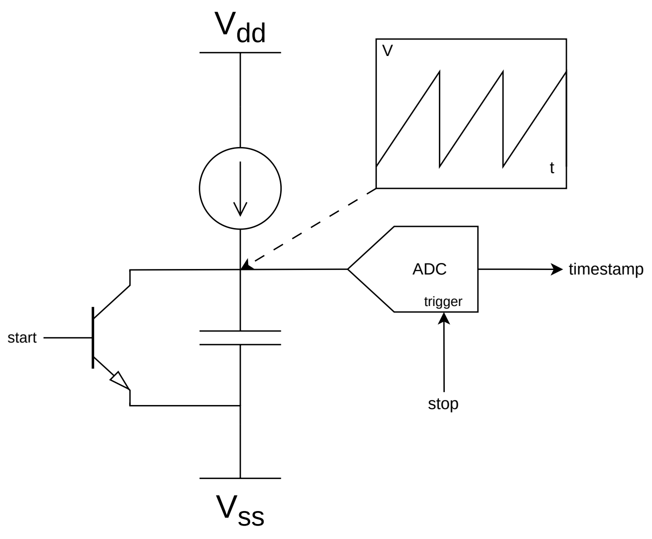
The biggest disadvantage is the limitation by the clock speed which is in turn limited by the capability of the technology used to implement the logic. While it is possible to use more modern technology and processes to attain higher resolution, there exist methods to reach resolutions higher than the period of the utilized clock.

1.2 Fine measurement approaches

1.2.1 Analogue method

One of the earlier methods of creating a high precision is through a time-to-voltage (TVC) converter. The idea of this approach is similar to the simple sampling counter discussed in the previous section, albeit with voltage instead of a numeric representation. A rising sawtooth-shaped signal is generated through the means of a capacitor, a constant current source, and a means to discharge the capacitor periodically. When the time between events should be measured, the *START* event opens the normally closed transistor/switch, and the capacitor begins to charge. When the *STOP* event occurs, the current voltage is sampled and converted to a numeric value using an ADC as shown in Figure 1.2, after which the transistor/switch closes again, discharging the capacitor, which prepares the circuit for the subsequent measurement.

Figure 1.2 Analog TDC illustration schematic



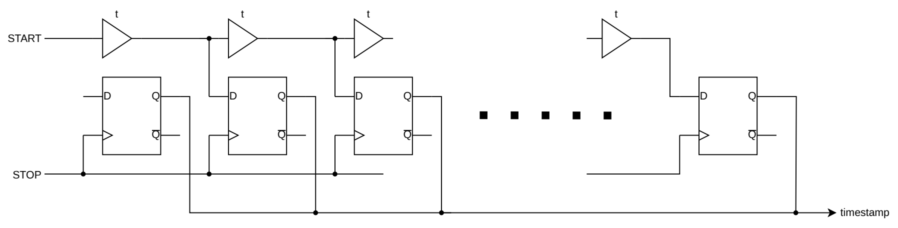
Because, instead of a discrete counter, a continuous voltage is used, it may seem intuitive that the device has an infinite theoretical precision limited only by the resolution of the used ADC. This is not entirely true. In electronic design, it is always a balancing act between precision and resource utilization both in terms of power and silicon size [1]. Soon, we start running into issues with high power consumption, which creates another problem as the circuit heats up and, therefore, as the electrical parameters change. Another caveat comes from the analogue nature of the circuit. External noise reaching the device through the power supply, connected signals, or the surroundings may have a non-negligible effect on the precision. This means extra care must be given to filtering the connected signals and power lines and shielding the entire device.

The analogue method is limited by the time it takes to fully charge the capacitor. This time is the maximal time difference between the start and stop events.

1.2.2 Digital delay line method

The simplest asynchronous method to capture time is based on a chain of precise delay elements accompanied by the same number of registers. This arrangement is constructed such that the time between two events can be measured. For our purposes, we call these two start and stop events.

Figure 1.3 Digital delay line TDC schematic



When the start event occurs, the rising edge propagates through the delay elements, gradually filling the entire chain as depicted in Figure 1.3. During the propagation of the said rising edge, the stop event occurs, triggering a read operation on all of the registers, effectively capturing the current state of the propagation. The index of the furthest register containing a 1 multiplied by the time delay of the used elements produces the delay between the two events.

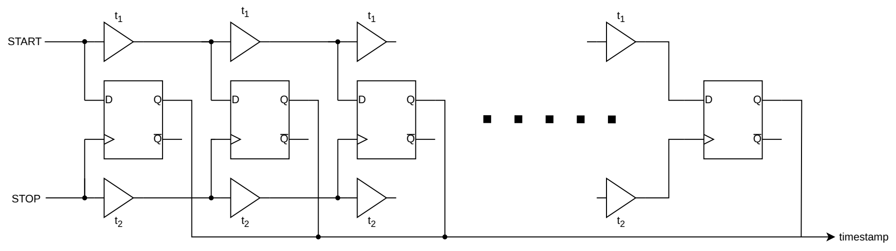
This method is limited in maximal time between the start and stop events. This time equals the sum of delays introduced by the buffer chain.

1.2.3 Vernier method

Further improvement of the delay line method is the Vernier method, which borrows the method to gain precision from the Vernier scale used in metrology. The method may be implemented either in the time domain [2] with two detuned oscillators and a coincidence detector or, more practically, by augmenting the delay line method as described above with another sequence of delay elements having different time delay values [3].

When a start event occurs, it begins, like in the digital delay line method, to propagate through a chain of delay elements connected to the data inputs of a chain of registers. The difference between the former method and this method is that as the stop event propagates, the registers sequentially instead of simultaneously, and at some point in the chain, a change from 1 to 0 can be observed. This is not unlike matching markings on a Vernier caliper, and the value is extracted similarly. However, this method requires the delay of the elements in the $STOP$ chain to be smaller than the ones in the $START$ chain to ensure the stop pulse is able to catch up with the start pulse. This is illustrated in Figure 1.4 where $t_2 < t_1$.

Figure 1.4 Vernier TDC schematic

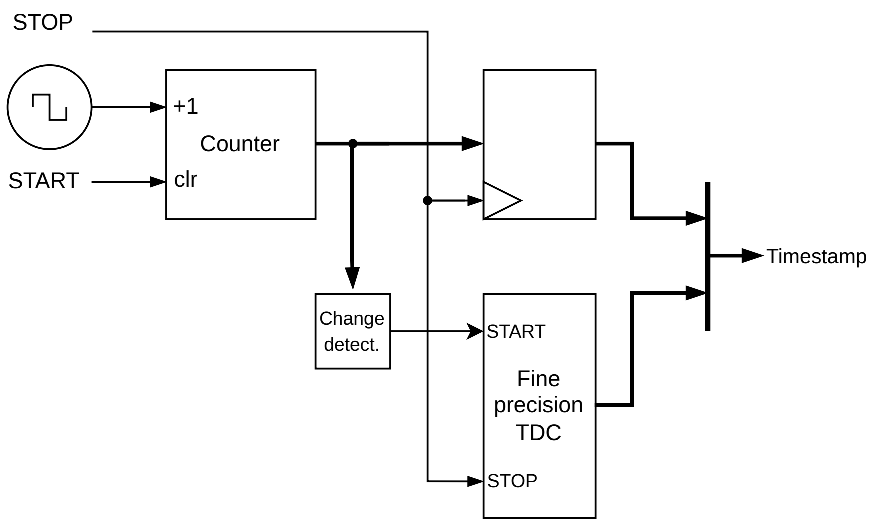


Like the delay line TDC, this method is also limited regarding the maximal time difference between start and stop events.

1.3 Hybrid approaches

To remedy the maximal time interval limitations found in the fine measurement approaches, a combination of the two can be used as shown in Figure 1.5. First, a free-running counter TDC is set up. This section generates the coarse timestamp as the number of elapsed reference clock cycles. Additionally, a TDC capable of sub-period accuracy is used. The maximal measurement length of this TDC is set such that it matches the period of the reference clock. This secondary TDC is then re-triggered (start signal is asserted) each clock cycle of the reference clock. When an event occurs, both the coarse and the fine TDCs are sampled, and the resulting timestamp is formed by the concatenation of the coarse timestamp and the time timestamp (after converting the fine TDC timestamp to match the encoding).

Figure 1.5 Hybrid TDC illustration schematic



Chapter 2
Existing solutions

There are currently many commercial solutions on the market today that can fulfill the requirements outlined in the assignment. The issue is with the price, which becomes too high should the TDC devices possibly be needed in more significant numbers, as in our case.

qutools - quTAU

Qutools GmbH is a company specializing in quantum optics and quantum technologies. They are recognized for providing advanced solutions for time-correlated single-photon counting (TCSPC) and time-tagging applications, which are essential in scientific research, particularly in the field of quantum optics. Qutools' devices often incorporate Time-to-Digital Converters (TDC).

The quTAU series is one of Qutools' product lines specifically designed for time-correlated single-photon counting (TCSPC) applications. quTAU is an 8-channel TDC device connected to a PC via a USB 2.0 interface. This interface limits its bandwidth compared to its competitors. It can be controlled via C/C++ or through the Labview software. [4]

Figure 2.1 qutools quTAU [4]

The listed key features on qutool website are:

- 8 input channels (LV)TTL (with hardware extension: user-defined threshold from -2 ... +3V)
- typ. 81 ps resolution (bin size)
- USB 2.0 interface
- Compact and easy-to-use
- Graphical user interface and device drivers for Windows and Linux
- Example software for C/C++ and Labview

PicoQuant

PicoQuant is a company that specializes in providing solutions for time-resolved fluorescence spectroscopy and single-molecule detection. They are known for their advanced time-correlated single-photon counting (TCSPC) devices, which often incorporate Time-to-Digital Converters (TDC) for high-precision time measurements.

Their portfolio offers a wide range of devices with the number of channels ranging from 1 to 64 and resolutions up to 1 ps with sub-ns time between incoming samples. This is summed up in the Table 2.1.

Figure 2.2 PicoQuant PicoHarp330 [5]

Table 2.1 PicoQuant TDC portfolio [5]

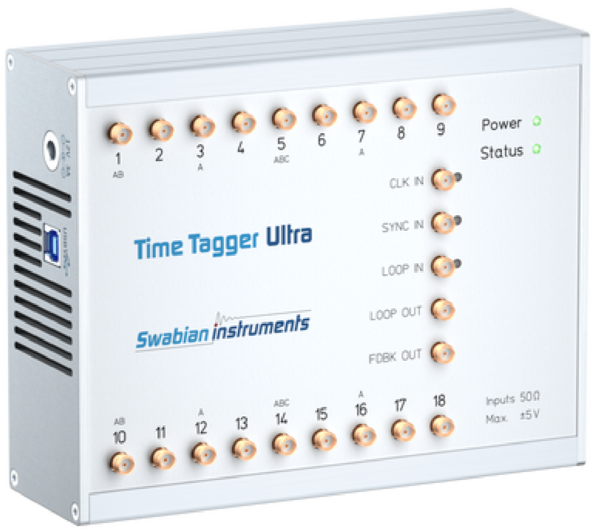
	HydraHarp 400	MultiHarp 160	MultiHarp 150	PicoHarp 330	TimeHarp 260
Number of detection channels besides common synch channel	2, 4, 6, or 8	16, 32, 48, or 64	4, 8, or 16	1 or 2	1 or 2
Minimum bin width	1 ps	5 ps	5 ps (P) 80 ps (N)	1 ps	25 ps (PICO) 250 ps (NANO)
Dead time	<80 ns	<0.65 ns	<0.65 ns	<0.68 ns	<25 ns (PICO) 2 ns (NANO)
Interface	USB 3.0	USB 3.0, FPGA Data Interface	USB 3.0	USB 3.0	PCIe 2.0 x1

Upon request, PicoQuant stated that “Time tagger prices start in higher 4-digits € region but can go to very high 5-digits depending on timing resolution and channel number.”

Swabian Instruments

Swabian Instruments is a company known for its expertise in high-performance instrumentation for quantum optics, time-correlated single-photon counting (TCSPC), and related applications. Swabian Instruments provides a range of products, including Time-to-Digital Converters (TDC), that are designed for precise time measurements in scientific and research settings.

Figure 2.3 Swabian Instruments Time Tagger Ultra [6]



Regarding the price, Swabian Instruments states that it mostly depends on the timing jitter of the instrument and that the prices range from a few thousand € for devices with jitter of 100 ps or more and up to a few tens of thousands € for devices with RMS jitter below 5 ps. This was stated for devices with 4 channels.

ID Quantique

ID Quantique is a French company specializing in quantum-safe security solutions, quantum key distribution (QKD), and quantum random number generation. The company is a pioneer in quantum key distribution, a technology that uses the principles of quantum mechanics to enable secure communication by distributing cryptographic keys between parties. Also, their portfolio contains high-performance photon detectors for use in quantum information processing and other applications alongside supporting devices. This includes the ID1000. The ID1000 is an integrated time tagging, coincidence correlation, and delay/pulse generation solution [7]. The declared parameters are 100 MHz sample rate per channel with 1 ps resolution and less than 4 ps jitter.

Figure 2.4 ID Quantique ID1000 [7]

In response to the pricing inquiry, ID Quantique stated, "The price of our ID1000 is from 10k to 18k CHF depending on the options". This amounts to roughly 11000 € to 20000 €.

Chapter 3

Analysis and approach selection

To start off the analysis, some requirements for the design have to be set. The device must be capable of sufficient bandwidth, which we defined as continuous 1 MS/s on 4 channels. We have to ensure that the interface from the TDC chip to the supporting logic, the supporting logic itself, and the interface to the PC are capable of sufficient throughput.

3.1 TDC-GPX2 - mode of operation

To avoid confusion, it is a good idea to mention that the abbreviation TDC and device are used for both the TDC-GPX2 chip and the entire object of the thesis. In this section, only the chip is being discussed.

The entire device is based around the TDC-GPX2 chip from ScioSense. This device was selected as it is the flagship device from ScioSense despite being affordably priced. There was no reason not to pick the most feature-packed device. The chip has 4 channels capable of ingesting up to 32 MS/s with minimal 20 ns pulse spacing when used independently and up to 70 MS/s with minimal 5 ns pulse spacing when two channels are paired [8].

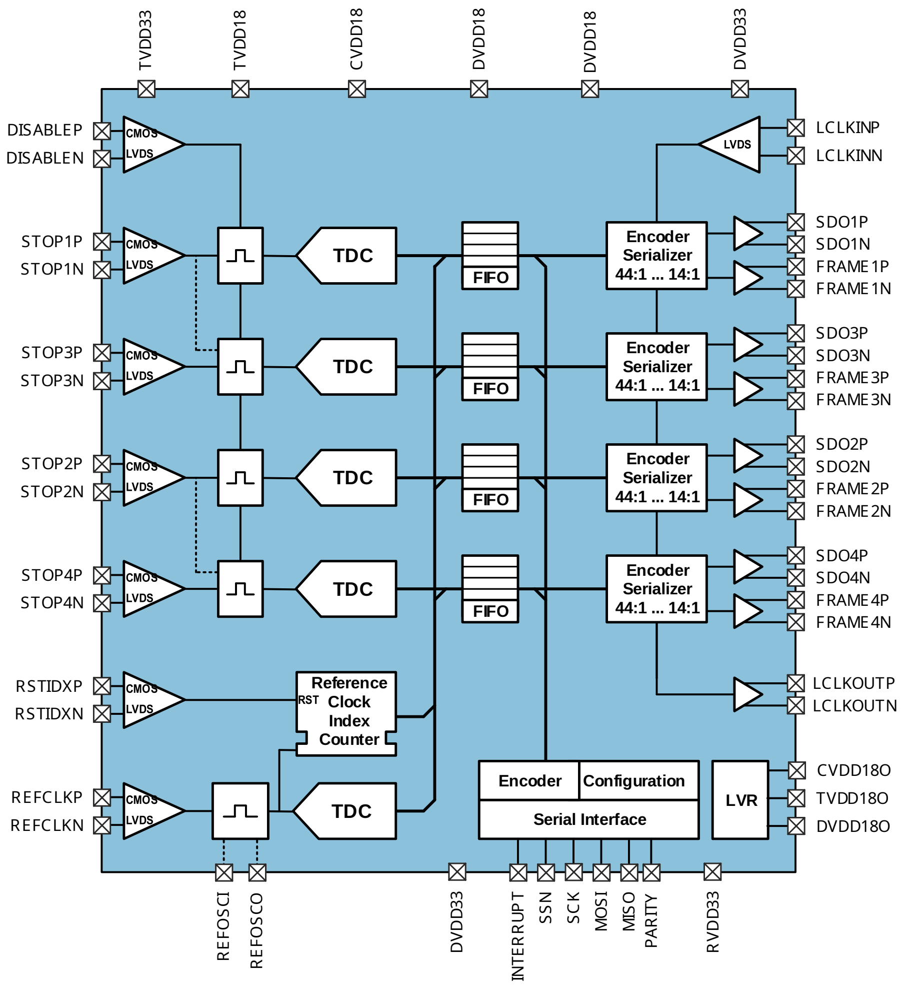
The time measurement capabilities of the chip are based on a reference clock provided through a crystal oscillator or on an external clock source. The frequency of this clock must be between 2 MHz and 12.5 MHz. The chip separated the measured time into two halves. The first half, called *reference clock index* by ScioSense, describes the number of reference clock cycles since reset/overflow. To gain higher precision, the clock is internally subdivided into a configurable number of time slots. This way, precision in the order of picoseconds can be achieved. This second half of the measured time is called *stop result*.

Upon the arrival of the pulse, these two values are sampled and stored in an internal FIFO, ready to be read out by the supporting logic. Each channel is equipped with a serializer, which turns the time data from the FIFO into a stream of bits sent to the external device alongside a *frame* signal, which indicates the start of a transaction.

The chip is configured via an SPI interface, allowing for, albeit rather slow, readout of the sample data. The number of bits used for the reference clock index, stop result, and communication interface parameters must be configured through this interface.

The communication interface is also rather capable. It can be run in SDR and DDR modes and with up to 200 MHz clock frequency. To receive the data, the supporting logic must provide a clock signal, which the TDC uses to run the output serializer and buffers. This signal is looped through the device and output back. This delayed clock is then used to sample the outbound bit streams.

Figure 3.1 TDC-GPX2 block schematic [8]



3.2 Hardware selection

A suitably powerful interface and a compatible device must be selected to transfer the data from the TDC to a PC. Regarding the interface, either USB 3.0 or PCIe comes to mind. Lower USB standards may be sufficient for the base case outlined at the start of the chapter. On the other hand, the TDC chip is way more capable than that, and it would be a shame to limit ourselves too much. PCIe interface could be suitable as the speeds greatly exceed those of USB, but the intention is to create a standalone device that could be plugged into any PC. Using PCIe would either bind us to a fixed computer or necessitate the utilization of Thunderbolt. In the end, USB 3.0 was selected.

Now, the task was to pick a suitable interface device. At first, the idea was to use a processor-based device. Microcontrollers with USB 3.0 support or with sufficiently fast I/O to enable an FTDI or similar device to proxy the communication are readily available. The outlined speeds can also be passed through a moderately modern MCU. The issue lies with the speed of the interface to the MCU. The TDC chip suggests using a rather fast SDR/DDR interface (up to 200 MHz range), which could prove problematic for the I/O of microcontrollers. If a slower interface was used, a problem with lost samples during denser pulse bursts could arise as the FIFO found inside the TDC is only up to 16 samples deep.

Another option was an FPGA, which would have no issue ingesting data at these rates as it is common to run their interfaces at these speeds using provided logic primitives.

In order to have better performance margins and more parallel processing power in the event that some internal processing becomes necessary in the future, it was decided that an FPGA would be more suitable.

3.2.1 FPGA selection

To avoid the necessity to design an entire complex PCB that would support the powering, provisioning, and interfacing of an FPGA, it was deemed better to use a premade module which would facilitate all the necessary circuitry, and the focus could be put on the development of more relevant parts of the device.

There are multiple FPGA module vendors on the market today such as Digilent, Trenz Electronic, Terasic or Opal Kelly. It is also necessary to keep in mind that the module must either have sufficient I/O to connect to a USB 3.0 interface or to include the interface itself on board.

Regarding the vendors, only Trenz Electronic and Opal Kelly have suitable modules for our design. In the end, Opal Kelly was selected as Trenz Electronic products were not possible to buy during the time of development. This has proven to be a good choice because of the maturity and quality of Opal Kelly's libraries and supporting software.

Opal Kelly sells a plethora of devices from which the XEM7310-MT-A75 module was selected as it strikes a good balance between price, I/O count, and FPGA size. This module includes a Cypress FX3 USB 3.0 interface, and the communication is abstracted by an Opal Kelly-developed library, which streamlines the communication. Another benefit of this device is the possibility to upgrade to XEM7310-MT-A200 if the built-in memory is not sufficiently large.

3.2.2 Interface library

Opal Kelly provides a software platform for FPGA integration to a PC called Front Panel. Front Panel also provides functionality for the configuration of the FPGA, including bitstream upload, external reset, and clocking in some FPGA modules. The platform is responsible for abstracting away the complexities of USB communication and communication with the Cypress USB interface chip and the user design.

After the FPGA has been initialised, the communication may commence. The bitstream must be Front Panel enabled. This means that an instance of an `okHost` module must be present in the FPGA and connected to the correct pins of the FPGA. This module is part of the Front Panel HDL library, which also contains other modules for communication that are connected via the `okHost` interface. On the PC side, the user can either use the Front Panel application or the API. The former can be used to specify a user-friendly GUI interface utilizing buttons, hex-digit displays, sliders, and others. The latter is used in conjunction with user code, allowing for the development of a custom application that can, for example, perform further processing of received data. This is how the communication is done in this case, and the details are further discussed in the Software chapter.

The library provides the following communication primitives. Except for the Register Bridge, all of them are available both in *In* and *Out* variants. The direction is respective to the FPGA.

Wire

A 32-bit asynchronous wide signal is used for transferring values that are not used for triggering or which change infrequently. In the *In* direction, when the PC updates the value, all the signals are updated at once. In the *Out* direction, the FPGA is free to update the signals at any time. When a change is detected, the interface updates the new value in the PC, where it can be asynchronously read. It is not possible to assign a callback, which would be called after a state change.

Trigger

The trigger is similar to the Wire with a few key differences, such as a clock input used for synchronization of the trigger output to a desired clock domain. In the *In* direction, when the PC asserts a trigger, a single clock period-wide pulse is generated in the Trigger module. This pulse is synchronous with the provided clock. In the *Out* direction, the trigger is sampled on the rising edge of the provided clock. This new value can be read in the PC. Sadly, assigning a callback to this event is also impossible. The author considers this the biggest flaw of the library, as this should be possible to implement through USB interrupt transfers.

Pipe

A pipe is a communication means used for transferring more significant volumes of data. The entire communication is PC-driven, and there are no provisions for some form of handshake or throttling. The data has to be readily available for the pipe to read, most probably in some form of FIFO. The Pipe interface is made so that it can be connected directly to the FIFO provided by Xilinx as part of Vivado.

Block-Throttled Pipe

This is an augmented version of the pipe, which includes a ready signal provided by the FPGA logic. Also, the concept of a block, a user-specified number of words transferred at once, is introduced. The FPGA logic should assert the ready signal only when ready to transmit an entire data block. Opal Kelly recommends using this method only for transferring constant data streams as prolonger de-assertion of the ready signal could cause a USB stall condition.

Register Bridge

As the name implies, the register bridge bridges a register file interface from the PC to the FPGA. A 32 bit address space is provided with 32 bit data word size for a total of 16 GiB of addressable space. In this way, single registers can be made available, or entire address spaces used in the FPGA can be mapped.

Chapter 4
Hardware

It must be stated that the actual design of the electronics and the PCB is the work of Mgr. Michal Dudka, who has developed the hardware (including Figure 4.3 based on the author's requirements and specifications. The author is responsible for the interface specification, the choice of the main connector, and the correct selection of the pins to match the FPGA clocking pins.
Despite that, the author chose to include this chapter as the thesis would only be complete with it included.

To speed up the development, the current hardware version is based on the BRK7310MT breakout board, allowing for easier connecting of the FPGA module to the custom hardware. Each of the TDC channels provides a differential serial interface to transmit data, and to configure the supporting circuitry, an SPI interface is necessary.

The input pulses can have varying voltage levels and have to be passed through a comparator circuit with a suitable, field-configurable reference. For our application, MAX5715 [9] was selected as it provides 4 channels and, like the TDC-GPX2, uses an SPI interface running in the same mode (mode 2) for configuration. A shared SPI bus was used with separate chip select signals. After the input pulses pass through the comparators, they are fed directly into the TDC chip.

To perform any time measurement, a reference clock must be provided to the TDC. This clock will be externally sourced or generated by the FPGA (discussed later). The external clock is not passed to the TDC directly. It is first buffered with some hysteresis to clean possible slow or jittery transitions and then passed to the FPGA, where it can be further divided. Then, either this divided clock or a generated clock is passed back to the TDC for timekeeping.

The TDC, reference DAC, clock buffer, channel, reference clock connectors, and the rest of the custom circuitry are located on a custom PCB attached to one of the four ports on the BRK7310MT board. The choice of the port was based on the length and length difference of the traces (minimizing both), sufficient amount of I/O, better aggregation of I/O into fewer banks, and availability of power. Out of the four possible choices, a port named MC1-A was selected. Despite not containing any power pins and being the least balanced regarding lengths of traces, it was the only connector that had sufficient I/O available in such a manner that specification-compliant clocking of the inputs could be achieved. This was considered the most important.

To allow for the correct sampling of the inputs, the FPGA provides special clock input pins labeled MRCC and SRCC, which are connected in such a way that they allow for proper clocking of their respective I/O banks (in the case of SRCC) or all of the banks (in the case of MRCC). Should the clock input for data sampling be provided through a standard input pin, the clock would have to be passed indirectly through the FPGA fabric. While possible, this would impede the reliability at higher speeds, resulting in a lower maximal clock speed.

The TDC chip and all other custom circuitry are hosted on a PCB pictured in Figure 4.2. The initial version of the board had an issue, which was resolved by a bodge wire. One of the inputs that was thought to be possible to have connected via a single wire had to be made differential.

The entire device is pictured in Figure 4.1. The blue board connecting everything together is the BRK7310 breakout board. On it, the green board is the XEM7310-MT SoM. The connector used to interface did not contain any power pins, and therefore, a separate wire had to be added (red and blue wire connected between the breakout board and the custom PCB).

Figure 4.1 Entire TDC device including the breakout board

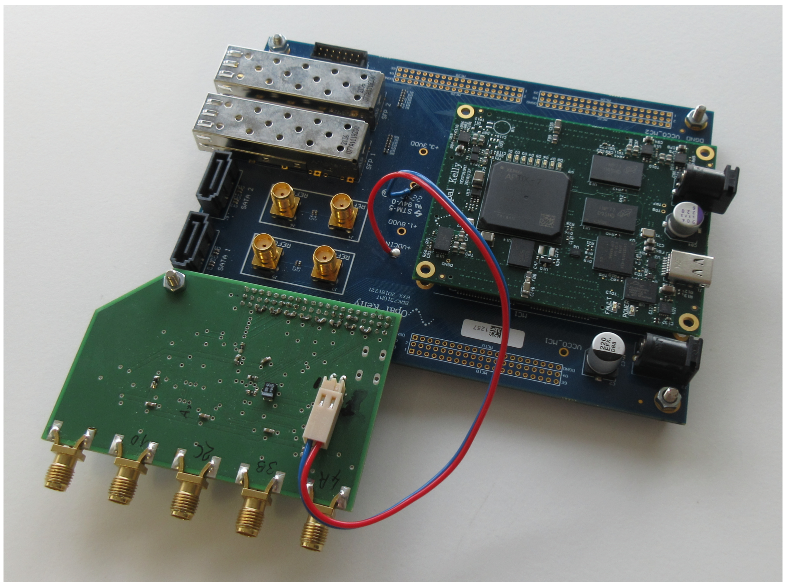


Figure 4.2 Custom PCB with the TDC chip

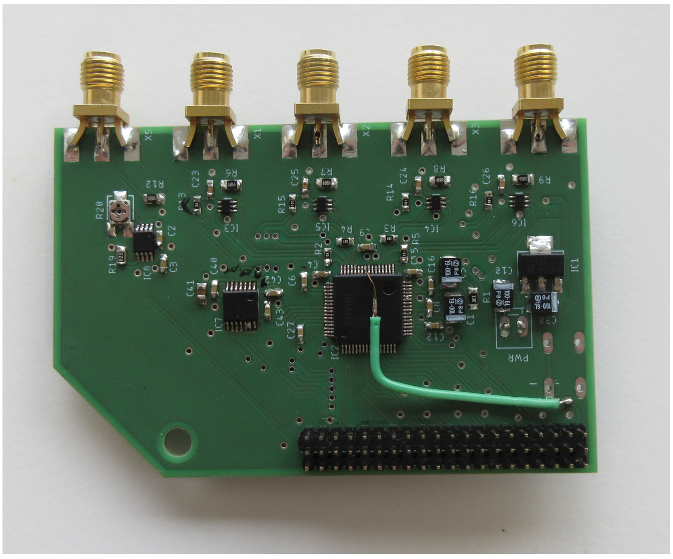
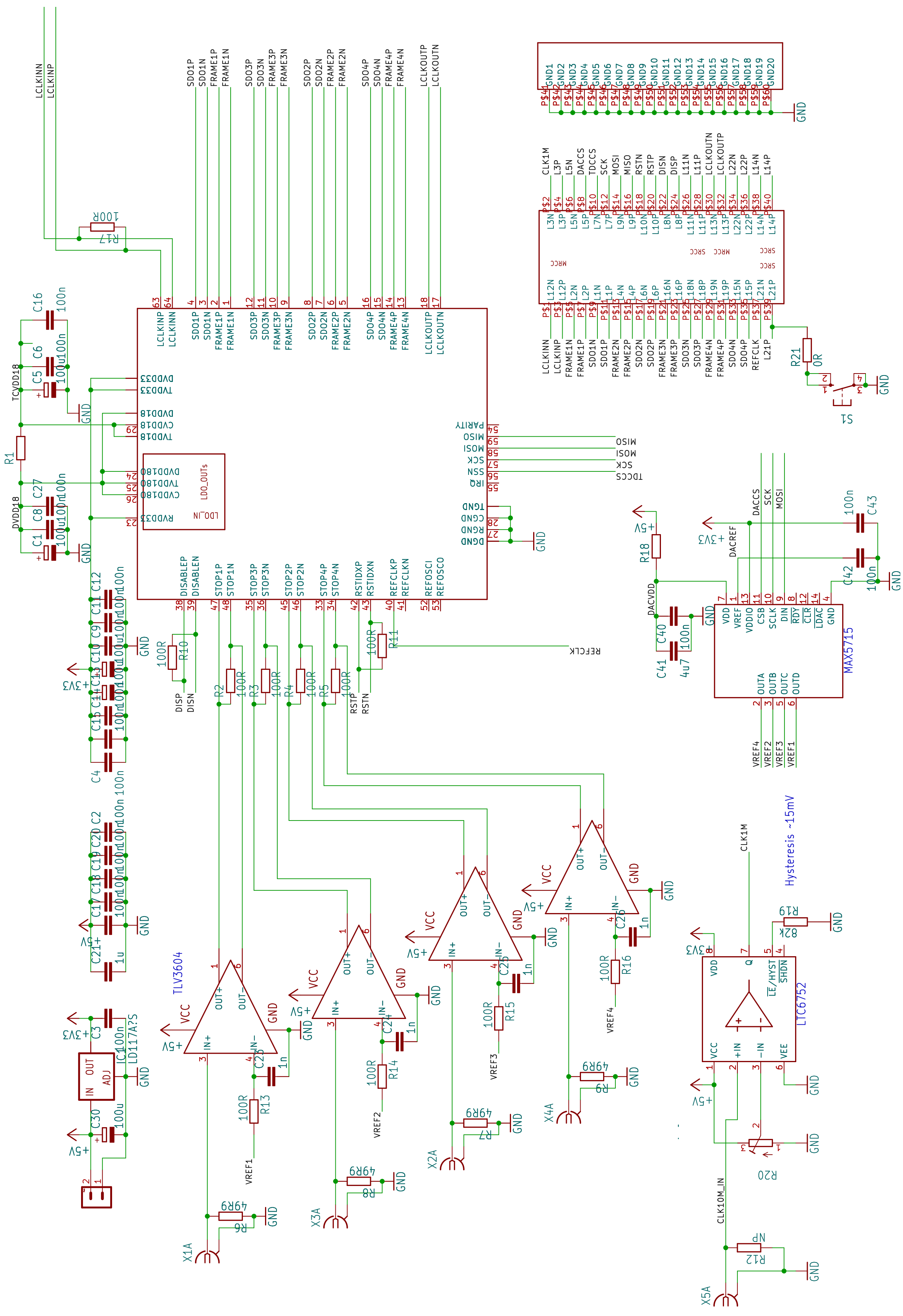


Figure 4.3 Board schematic

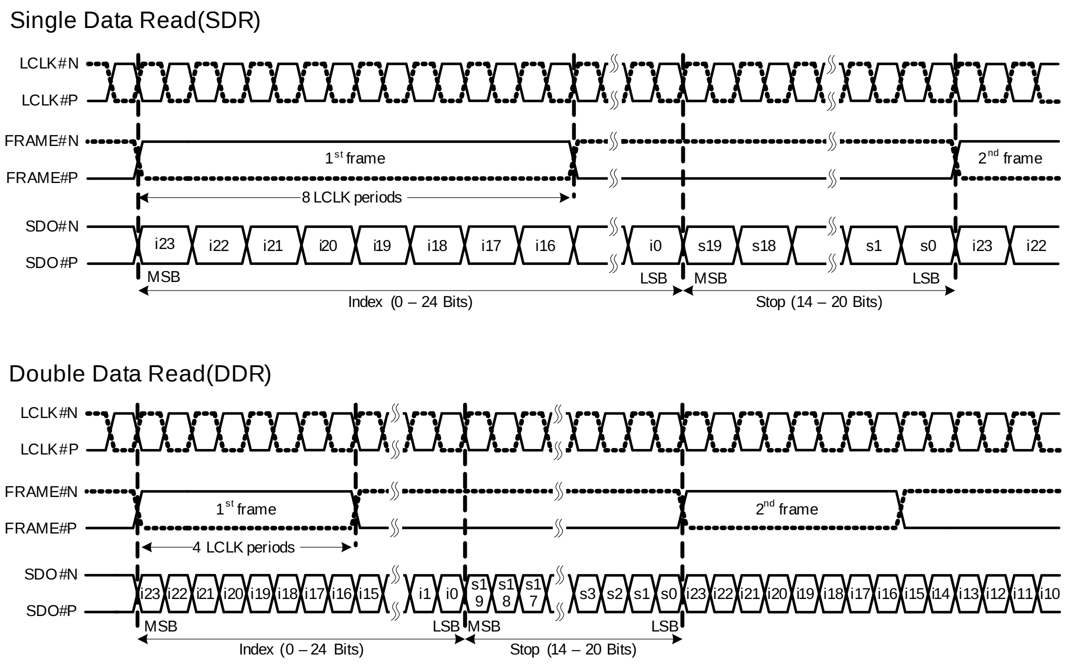


4.1 TDC - FPGA data format

Regarding sending data to the FPGA, each of the channels from the TDC uses an independent communication interface consisting of 2 differential pairs called `FRAME` and `SDO`. The `SDO` pair is responsible for transferring the actual data, while the `FRAME` signal denotes the start of the transaction through assertion to 1 for the first 8 bits of the transaction. The samples are transferred serially, MSB first, and the interfaces are clocked by a common clock provided to the TDC by the FPGA. This clock passes through the device, is output back to the FPGA (albeit somewhat delayed), and is used to sample the incoming data.

Each of the samples is a concatenation of 2 values corresponding to 2 internal TDCs, as described in the theoretical chapter. The bits from the coarse timer called *reference index* in the TDC-GPX2 datasheet, are concatenated to the fine TDC, called *stop bits* and transmitted. The lengths of both of the parts are configurable, and the sum of their lengths directly affects the maximal sample rate, which can be transmitted over the interface. With shorter samples, more samples can be sent in a given amount of time.

Figure 4.4 TDC-GPX2 communication example waveforms[8]



4.2 Price breakdown

Compared to existing solutions, the price is much lower. However, this does not take into account the labour put into development, the unfinished nature of the product, the tools for development others. Only the price of the electronic components and PCB are factored in. The price for the miscellaneous components is a rough estimate only.

Table 4.1 Hardware price breakdown

XEM7310MT-A75	1	618.75 €
BRK7310MT	1	136.45 €
PCB	1	4.55 €
TDC-GPX2	1	48.20 €
MAX5715	1	10.00 €
TLV3604	4	21.68 €
LTC6752	1	5.00 €
Miscellaneous components	1	10.00 €
	Total:	854.63 €

Chapter 5

Communication

Since the provided library by Opal Kelly encompasses all the necessary facilities for both configuration and rapid transfer of data, it is used as the only means of communication. The following sections will describe how the device is configured and how the data is extracted from the device.

5.1 Configuration

The entire configuration of the device is done through a register bridge which mediates communication between high-level API available in the software and a simple address-data-strobe interface in the FPGA. The interface provides a 32 bit address space (which is excessively large for our application), 32 bit data and read/write strobe signals. In the spirit of keeping the design expandable, the address space is partitioned and inside the FPGA, a separate module is responsible for each of the partitions. To select the internal module, the top byte of the address is used.

The idea is to move the majority of configuration logic out of the logic of the FPGA into the software. It was decided to omit some abstraction in order to simplify the FPGA side of things as it is much simpler to implement control logic in software.

Any numeric literals mentioned in the text follow the C programming language convention. 12 is decimal 0x12 is hexadecimal and 0b101 is binary.

5.1.1 General module

This is a module responsible for receiving and storing configuration for the vast majority of the FPGA design. It provides information on the state of the internal buffer, TDC data receiver configuration, and the rest of the configuration with the exception of the external chips.

Table 5.1 General registers

Address	Name	Comment
0x00000000 - 0x00000004	VERSION_SHA	Git commit hash of the source version
0x00000010	SYSTEM_RESET	Reset signals
0x00000020	REFCLK_SOURCE	Reference clock source
0x00000021	REFCLK_DIVISOR	Reference clock divisor
0x00000030	TDCCLK_DIVISOR	TDC LCLK divisor
0x00000040	CHANNEL_RUN	Channel enable/run signals
0x00000041	TDC_TIMESTAMP_SHIFT	Number of dropped bits from TDC data
0x00000042	TDC_STOP_DATA_BITS	Number of stop bits in TDC data
0x00000043	TDC_REF_INDEX_BITS	Number of ref index bits in TDC data
0x00000044	BUFFER_SAMPLES	Number of samples in buffer
0x00000050 - 0x00000053	DROPPED_SAMPLES	Number of dropped samples per channel
0x00000060	STATUS_BITS	Status bits - LEDs

VERSION_SHA

Data type	Access	Range
uint32[5]	R	0x0 - 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

When the FPGA bitstream is compiled, the build environment automatically stores the entire git version hash into this read-only register. This register exists for the sole purpose of simplifying the answer to the question “Which version are you using?” once the device is deployed and some issues have to be inevitably fixed.

SYSTEM_RESET

Data type	Access	Range
bit[32]	W	1

Resets the entire FPGA logic when 1 is written. The TDC chip and reference DAC are not affected and must be reset separately via SPI if needed.

REFCLK_SOURCE

Data type	Access	Range	Reset
uint32	R/W	0, 1	0

It is required to provide the TDC chip with a reference clock with a frequency in the range of 2 to 12.5 MHz [8]. This register allows the user to select between an externally supplied reference clock and an internally generated 10 MHz clock. Value 0 means the internal source is used, and 1 means the external clock is used.

REFCLK_DIVISOR

Data type	Access	Range	Reset
uint32	R/W	0-63	1

Either the external reference clock or the internally generated can be further divided. This register specifies the used integer divisor. Value 0 disables the clock.

TDCCLK_DIVISOR

Data type	Access	Range	Reset
uint32	R/W	0-63	2

This register is used to set the speed at which the communication interface between the TDC chip and the FPGA runs. It specifies the integer divisor used for dividing an initial 200 MHz clock. Value 0 disables the clock.

CHANNEL_RUN

Data type	Access	Range	Reset
bit[32]	R/W	0x0-0xF	0xF

This register allows the user to enable/disable a channel on the FPGA side. This can also be done on the TDC chip side.

TDC_TIMESTAMP_SHIFT

Data type	Access	Range	Reset
uint32	R/W	0-38	0

It may be necessary to drop MSB from the data received from the TDC chip due to the nature of the measurement performed and the reference clock that is currently set. This register specifies exactly how many bits should be dropped. **The TDC configuration via SPI must match.**

TDC_STOP_DATA_BITS

Data type	Access	Range	Reset
uint32	R/W	14-20	14

A register with a value that has to match a similarly named register in the TDC chip configuration referring to the number of bits used for *fine measurement* as described in the theoretical chapter. **The TDC configuration via SPI must match.**

TDC_REF_INDEX_BITS

Data type	Access	Range	Reset
uint32	R/W	0-24	0

A register with a value that has to match a similarly named register in the TDC chip configuration referring to the number of bits used for *coarse measurement* as described in the theoretical chapter. **The TDC configuration via SPI must match.**

BUFFER_SAMPLES

Data type	Access	Range	Reset
uint32	R/W	0x0-0xFFFFFFFF	0x0

A critical register used in the extraction of the data from the register. It contains the number of samples stored in the main buffer, which are guaranteed to be immediately available via the USB interface.

DROPPED_SAMPLES

Data type	Access	Range	Reset
uint32[4]	R/W	0x0-0xFFFFFFFF each	0x0 each

When a channel receives a sample, and the corresponding internal buffer is full, the sample is dropped, and the corresponding `DROPPED_SAMPLES[i]` is incremented. This can be used to detect lost samples due to insufficient bandwidth. The register is cleared by writing 0 to it. Other values are ignored.

STATUS_BITS

Data type	Access	Range	Reset
bit[32]	R	0x00-0x01	0x00

A debugging register mirroring the state of 8 LEDs located on the FPGA module. Currently, only one bit is used, denoting the busy state of the SPI interface.

5.1.2 SPI module

The SPI module is a bridge between the register interface and the other SPI-configured hardware on the board. The logic supports all the multi-byte transactions required by the TDC and the voltage reference DAC. Writing the data is trivial. A write to the register initiates the SPI transaction and captures no data. A read is more complex as the register interface has a design flaw/oversight/simplification. The bridge has no form of a handshake and expects to see data one clock cycle after transmitting the read strobe [10]. This is achievable only when the data is stored inside the FPGA but impossible when it must first be retrieved from an external device. This issue was handled through the addition of an auxiliary register with a flag. To read data from SPI, the master device must provide a clock in sync with which the slave provides data. A write transaction with byte 0x00 can be initiated from the interface to achieve this behaviour. To transfer data from SPI to the PC, a *SPI_RETURN* and *SPI_RETURN_VALID* registers are provided. Upon the reception of a byte, it is shifted in the *SPI_RETURN_VALID* register, which behaves like a FIFO. Also, the register *SPI_RETURN* is incremented by 1. Both registers are cleared when *SPI_RETURN* is read. As *SPI_RETURN* is 4 bytes wide (FIFO is 4 transactions deep), should the *SPI_RETURN_VALID* register reach values higher than 4, an overflow is detected. Should we, for example, issue a read command consisting of one byte written and one byte read, we issue a write command in the form 0xXX00 where XX is the command byte. The result will be found in the second byte of *SPI_RETURN* with *SPI_RETURN_VALID* being set to 2.

The examples shown in Figures 5.1 and 5.2 only illustrate how SPI is used to communicate in this design and not the general usage of the SPI interface. Additionally, the transfers with the TDC chip are up to 3 bytes long, which is not reflected in the waveforms.

Table 5.2 SPI registers

Address	Name	Type	Comment	R/W
0x01000000	SPI_RETURN	byte[4]	SPI Read data	R
0x01000001	SPI_RETURN_VALID	uint8	SPI_RETURN number of bytes	R
0x01010000	REFDAC_RAWCMD	byte[3]	Raw command to DAC as per datasheet	W
0x01020000	TDC_RAWCMD	byte[2]	Raw command to TDC as per datasheet	W

Figure 5.1 SPI write example

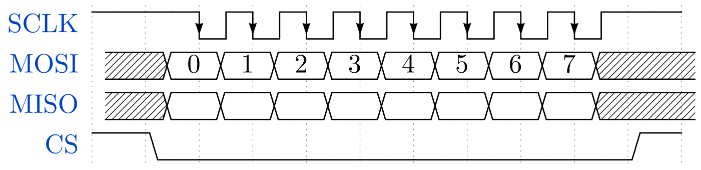
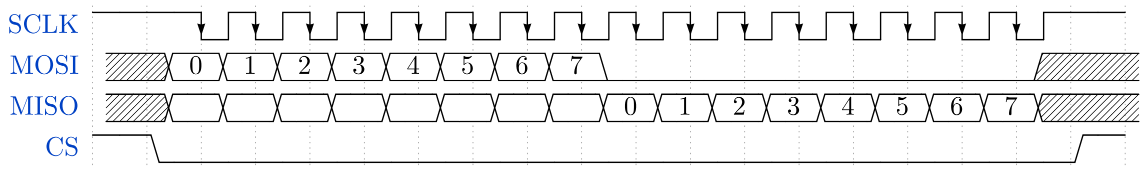


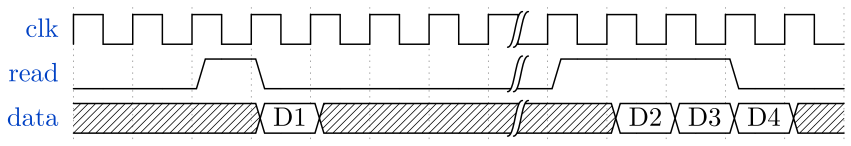
Figure 5.2 SPI read example



5.2 Process of retrieving sample data

The communication library provides a pipe interface, which allows a stream of data to be transferred to or from the FPGA on demand from the PC. The pipe does not include any flow control mechanisms (Figure 5.3), and therefore, it must be manually determined how much data can be transferred at each moment. Bear in mind that the PC initiates all of the transfers, and there are no facilities for sending unrequested data from the FPGA.

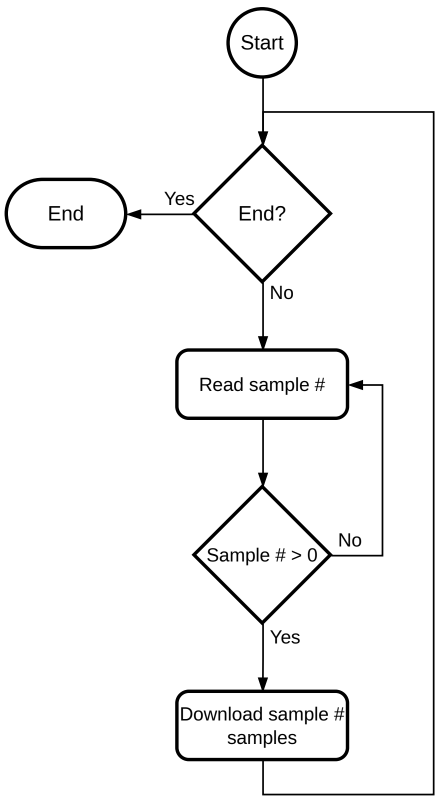
Figure 5.3 Pipe read waveform



The algorithm used is relatively simple but effective. The register interface provides a register containing the number of samples stored in the buffer. A read transaction is always initiated with a size equal to the said number. This ensures that no buffer underflows (reads on empty buffer) occur and that the reading is performed at a maximal safe rate. The data is then further processed and stored on the PC side. The retrieval algorithm is illustrated in flowchart portrayed in Figure 5.4.

This can be further augmented by the inclusion of a delay as a form of rate-limiting in the case that the amount of samples in the buffer is low. The resulting improvement lies in properly blocking the thread at idle times instead of busy waiting.

Figure 5.4 Sample download approach flowchart



5.2.1 Sample format

The pipe interface is 32 bits wide, which is in line with the intended sample size. The sample has to contain the actual sample and the channel number. To capture the channel, 2 bits are necessary, which leaves us with 30 bits for the actual sample. The format is shown in Table 5.3.

Additionally, since the timestamp value is always incrementing, overflows occur and have to be captured somehow. This is reflected in Table 5.4. A special message is therefore sent anytime an overflow occurs.

Table 5.3 Sample format

Bits	Description
0-29	Data
30-31	Channel ID

Table 5.4 Special messages

Value	Significance
0x3FFFFFFF	
0x7FFFFFFF	
0xBFFFFFFF	
0xFFFFFFFF	Time overflow

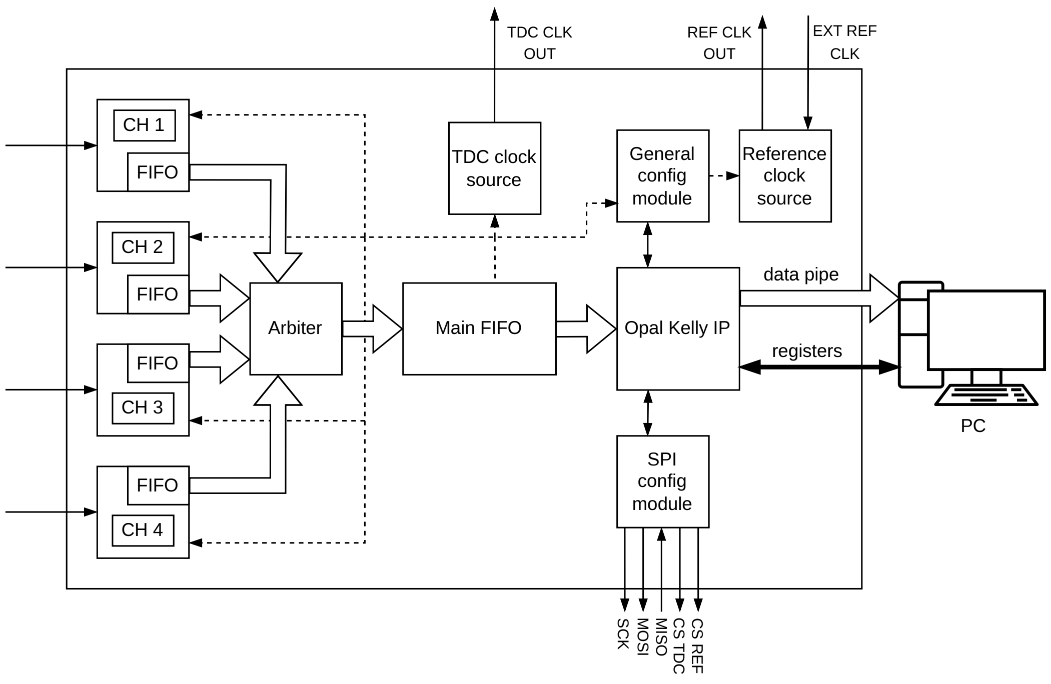
Chapter 6

FPGA design

As mentioned, the device is based on an Xilinx FPGA. The FPGA vendor provides facilities for developing the code, testing it, building it into a suitable bitstream, uploading it, and testing it in the actual hardware. The last feature has proven to be essential in developing the device and allowed us to omit rigorous, time-consuming testing due to time constraints imposed upon the project.

The FPGA design is split into several sections, mainly the communication/configuration section and the sample retrieval and processing part. It is also split up further into two clock domains. One of the domains is synchronous with a clock provided by the USB communication interface, and the other is synchronous with the sample retrieval clock provided by the TDC chip. The sectioning of the design has been done to accommodate further expansion of the feature set of the device. This will become obvious in the further sections.

Figure 6.1 System high level schematic



6.1 Code

The design can be logically split between logic handling the transfer of the samples from the TDC chip to the USB interface and configuration logic dealing with the SPI interface for chip configuration as well as the internal configuration.

6.1.1 Configuration facilities

As said before, the entire configuration is performed via the *okRegisterBridge* register bridge with a simple interface as specified in Table 6.1.

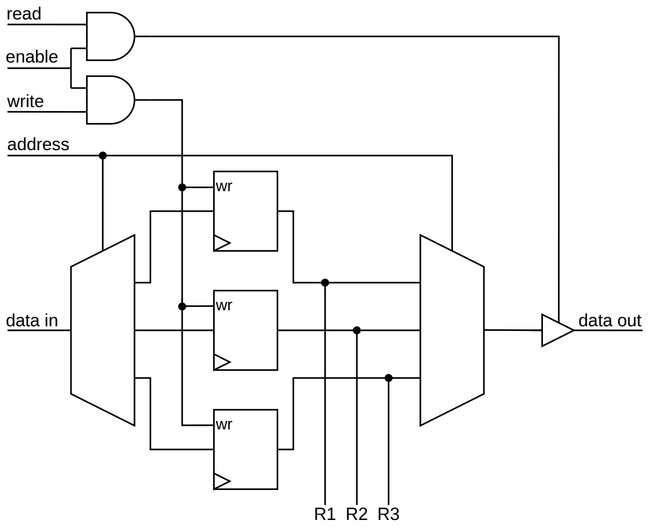
The configuration interface first reaches a switch module, which, based on the most significant byte, generates an enable signal for the respective interface and selects the correct read bus, passing it back to the Opal Kelly interface. The interfaces then branch further into multiple communication modules in an easily extensible manner. For that, an internal interface format is used, utilizing SystemVerilog interfaces as specified in Code listing A.

Unlike the register interface, the internal interface has only 24 bit address as the topmost byte is used to generate the module-specific enable signal. The modules consist of a simple set of 2 case statements running the writing and reading logic, and all the required registers are stored inside the module. This unification and templated design enabled the simple expansion of this system should the logic require an extension in the future. An example module diagram can be seen in Figure 6.2 with the corresponding code in the appendix.

Table 6.1 okRegisterBridge interface

Name	Width	Direction	Description
okHE	113	N/A	OK external interface
okEH	1	N/A	OK external interface
ep_write	1	out	Write strobe
ep_read	1	out	Read strobe
ep_address	32	out	Address
ep_dataout	32	out	Data output
ep_datain	32	in	Data input

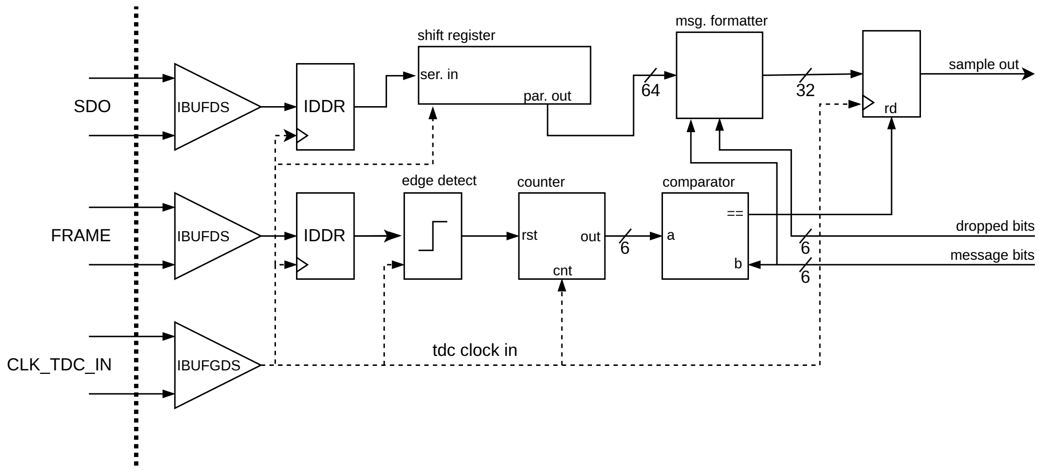
Figure 6.2 Configuration module example with 3 registers



6.1.2 Channel logic

The purpose of the channel module is the sampling, deserialization, and buffering of the data the channel receives. When an event occurs, and the TDC chip captures a timestamp, communication is initiated on the corresponding SDO and FRAME differential pairs. After first passing the pairs through an `IBUFDS` instance (differential input buffer), the signals are sampled using the clock provided by the TDC chip. While this sampling can be done in the FPGA fabric, it is much preferred to utilize hardwired logic found in the I/O section of the FPGA. The `IDDR` primitive can be used. It is a simple DDR buffer found in the I/O section of the FPGA (not in user logic) capable of sampling an input signal on both the rising and falling clock edge. Besides the signal integrity advantage, utilizing this primitive also enables one to choose which clock edge is used for sampling in SDR mode, should the signal be excessively delayed, or to run the TDC-FPGA interface in DDR mode.

Figure 6.3 Simplified TDC channel schematic



The timestamps must be deserialized once the input signals have been correctly sampled. This is done using a shift register and parameters (timestamp width and number of dropped LSB) available from the configuration part of the logic. The logic is controlled with an FSM so that the transactions can be received back-to-back with zero time in between. To prevent data loss in the case of higher load scenarios, a buffer is used inside each of the channels. For this, the Xilinx provided `xdc_fifo_sync` is used with a 256 sample depth (1 KiB).

The read interface of said buffer is then made available to an arbiter circuit, which moves the samples into the main buffer from which the USB interface reads.

The channel also contains provisions for detecting dropped samples due to insufficient sample extraction from the device. The `overflow` output of the FIFO is used, and it is routed to the general module. The FIFO asserts this signal whenever a sample is inserted and cannot be accepted (mainly because the FIFO is full). The general module has 4 pulse counter blocks, one for each channel. Their output is then made available to the application through a register.

6.1.3 Main buffer and arbiter

The logic contains a FIFO buffer to allow for smoother data extraction and prevent sample loss due to insufficient data download from the PC (As all data transfer operations are initiated from the PC). Unlike the in-channel buffers, the `xdc_fifo_async` variant of the Xilinx xdc FIFO is used. This is necessary as this buffer is also used to transfer data between the channel clock domain and the OpalKelly interface clock domain. The size of the buffer is maximal in terms of the available BRAM. For our case, this was 65535 samples (256 KiB).

To fill the buffer, the data from the pipes must be aggregated from 4 independent sources. A dedicated arbiter is constructed for this purpose. The arbiter is responsible for ensuring that no data is lost from the input channels and that the main buffer is filled only when it can ingest data. Solving the second issue is trivial as the buffer provides a `full` signal. The first issue is not so simple as it highly depends on the sampled event occurrence rate. Multiple algorithms can be used to choose which channel should have a sample moved into the main buffer.

The most straightforward approach would be a round-robin approach, where the channels are read sequentially without paying attention to the state of the channel buffers. While this may be sufficient in cases where the channels are receiving a similar amount of data, the moment we want to utilize one channel at a higher speed while keeping the rest unused, the approach is no longer effective. In extreme cases, this could even lead to data loss as the effective bandwidth of the channels is divided by the number of channels.

A better way to select channels to be read could be via a LRU algorithm. It can be expected that the channel left alone for the longest will have the most samples buffered. While this may help somewhat, it will have no impact as the number of channels is very low.

A better-yet approach considers the amount of data in each of the buffers. When a constant stream of data is being transferred from the channels, the number of samples is proportional to the rate at which the channels receive samples. This leads to an approach that balances the fullness of the channels and prefers to extract samples from the fullest. This method has one caveat. The `read_data` found on the FIFO, which denotes the number of samples that can be extracted, is usually delayed by a few clock cycles after a write to the FIFO. This could lead to highly loaded channels having a lower priority assigned than necessary. This will most probably have a negligible effect as the imposed delay is in the range of 1 to 4 clock cycles depending on the configuration [11].

In the end, the simple round-robin approach was selected for its simplicity, and it will be later demonstrated what issues arose from this decision, leading to diminished usability of the device.

6.1.4 Clock boundary handling

Because the design works with 2 clock domains, data and control signals must be transferred using suitable synchronization logic. As mentioned before, the samples are synchronized between the 2 clock domains using an Xilinx asynchronous FIFO, `xdc_fifo_async`. The reset signal is transferred using the `xpm_cdc_pulse` macro. Other signals, such as configuration signals, are not synchronized at all. The signals are not intended to change when the sampling is active and must be set in advance and not changed later. The `read_data_count` signal denoting the number of samples in the FIFO is generated in the clock domain synchronous with the register bridge and does not need synchronization.

6.1.5 Reference and TDC interface clock

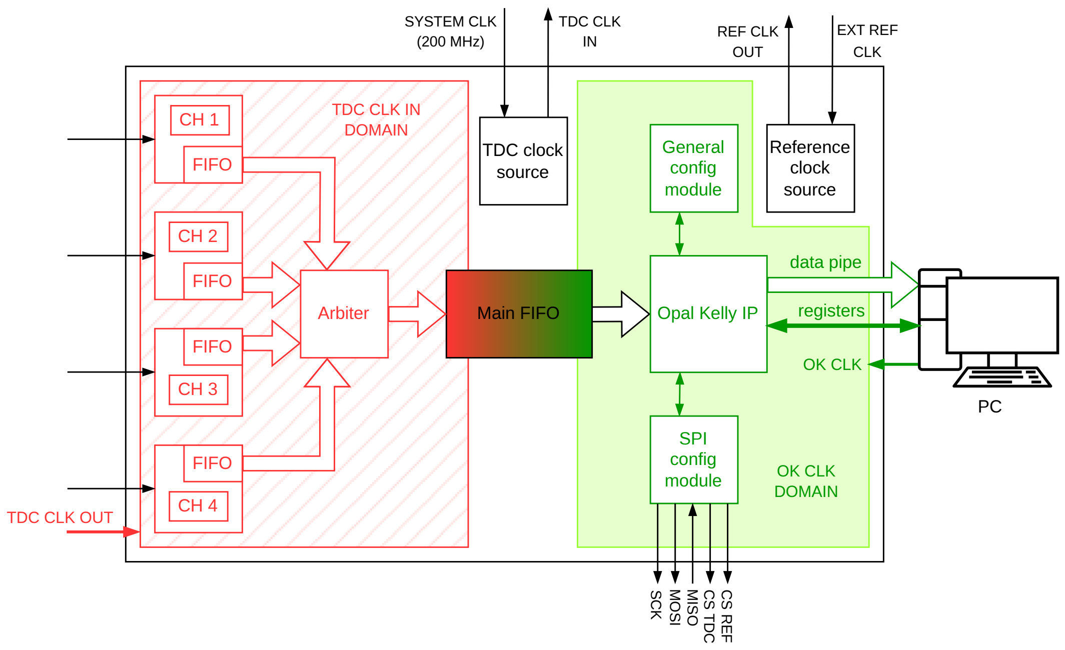
The FPGA code provides facilities for generating and preprocessing the external reference clock and the clock used for running the TDC-FPGA interface.

The interface clock is derived from the 200 MHz clock provided to the FPGA from the module. This clock is then divided, in logic, by a simple counter and passed to the TDC chip.

The reference clock generation is similar. The clock is based either on the same 200 MHz clock, which is divided by a factor of 20 to generate a 10 MHz clock on an externally provided, usually 10 MHz clock. The two clocks pass through a multiplexer controlled by a configuration register, and the resulting clock is then passed to a configurable divider identical to the one used for the interface clock.

The clocking scheme is described in Figure 6.4. The two domains are shown in red and green. The red domain is driven by the clock returned by the TDC chip, and the green domain is driven by a clock received from the Front Panel interface.

Figure 6.4 System clocking schematic



6.1.6 Expandability

While the current design fulfills the goals set in the assignment by providing a sufficient number of channels (4) and bandwidth for the designated application, there is room for improvement. The intention is to develop the design so that further development and improvement is as straightforward as possible. This is taken into account in multiple places.

The first provision for further expandability is within the communication and configuration subsystem, where multiple places for further development are implemented. The architecture was made multi-level to facilitate the addition of whole new configuration modules for additional independent logic. Also, inside of each of the modules, the architecture is trivial to understand and unified among the modules. New instructions are added by appending the new address to a shared package and expanding one case statement. Entire modules are added by, again, appending addresses to the shared package, instantiating the new module and connecting it to a parametrized switch module which switches between the configuration modules themselves.

The theme of parametrization brings us to another area of the logic: the channels themselves. While only 4 channels are currently present, the entire module is fully parametrized. All it takes to add more channels to the logic is to change one parameter and connect the newly expanded interface.

Initially, it was intended to include some form of preprocessing in the FPGA, but it was later not necessary. Nevertheless, should this need arise in the future, all the preprocessing can be included between the arbiter and the main buffer. This module (or multiple chained modules) should use the same interface as the buffer (Xilinx library FIFO).

Not only is the FPGA logic expandable but also the FPGA itself. The current version uses the XC7A75T FPGA, which contains 3780 memory cells [12]. There also exists a pin-compatible board version with the XC7A200T FPGA [13], which has 13140 memory cells [12]. This would enable the use of a larger internal buffer, which would be necessary in higher throughput scenarios. Also, both of the FPGA board versions contain a 1 GiB DDR3 SDRAM chip, which could be used as a buffer at the expense of higher logic use and development complexity.

The main buffer is internally built using the 36Kb BRAM blocks, as our word size is 32 bits. The XC7A75T has 105 of these BRAM blocks. The BRAM uses 38-bit words (32 + ECC), meaning we can fit 1024 words into each of the BRAM blocks. The maximal buffer depth is calculated as follows.

$$max\_buf\_depth = 2^{\lfloor \log_2(1024 * BRAM\_blocks) \rfloor}$$

The depth of the buffer has to be rounded down to a power of 2 due to a limitation of the used FIFO. For XC7A75T with 105 36Kb BRAM blocks, the maximal size is 64 KiS. For XC7A200T, it is 256 KiS as it has 365 of the required blocks[14].

6.1.7 Constraints

The constraints defined in the `.xdc` file are an extension of a constraints file provided by Opal Kelly for the XEM7310-MT module. This template contains all the pin assignments and IO standards for the onboard connections, such as the Front Panel interface, LEDs, the supplied 200 MHz clock, and a reset signal. In addition, the specified pin slew rates are specified as the Front Panel interface is run at 100.8 MHz clock. Lastly, all the clock parameters are specified, including the Front Panel clock and 200 MHz system clock. This includes the differential termination of the 200 MHz clock.

Then, the custom signals were added. This includes the wires for SPI, the reference clock with all the required timing constraints, the clock output and input for transferring data samples from the TDC chip, and the SDO and FRAME differential pairs with all the required configuration.

An issue arose from the fact that the 7-series FPGAs from Xilinx (such as the XC7A75T) do not permit any 3.3 volt standards besides the basic `LVCMOS33` or other, use case specific standards such as PCI33_3 or TMDS_33. To complicate matters further, there is no 3.3 V differential standard available.

This would not be a problem if the signals which need to be run at 3.3 V did not share an IO bank with others. As this is the case, a workaround had to be developed.

It is possible to run the interfaces out of spec by denoting a lower voltage standard in the .xdc file and supplying higher voltage to the I/O bank. The circuitry found there is capable of running on this higher voltage, and since there are no regulators or any other circuits that would lower the voltage, the output then runs on this higher voltage.

The only caveat lies within the timing analysis. The timing analysis, which is responsible for calculating whether the imposed timing constraints are feasible and whether the design will work, assumes the lower voltage. This means the timing analysis wrongly assumes higher performance margins (as running an interface at lower voltage usually leads to higher maximal frequency) and, in edge cases, could result in falsely better timing results than possible.

6.2 Testing and debugging

Initially, for every SystemVerilog module, there was a unit test procedure developed. These tests were then run using the *xsim* simulator built in the Vivado suite and validated using assertions and manual waveform examination.

This form of testing was later abandoned in favour of hardware testing using the Integrated Logic Analyzer due to the growing necessity to test outside-facing logic. ILA is a piece of technology allowing the instantiation of a logic analyser inside the FPGA fabric. This analyser can be connected directly to the signals that should be examined instead of passing them through the I/O. The analyser connects to the FPGA via JTAG and can be controlled through the Vivado GUI. The results are displayed in a manner similar to the waveforms shown during the simulation.

The most significant advantage is the possibility to test the design in actual hardware. This allows the testing to capture problems caused by the connected logic, which would have to be simulated in the case of the pure simulation approach. Development of such models would also require much time, which can be saved in this manner.

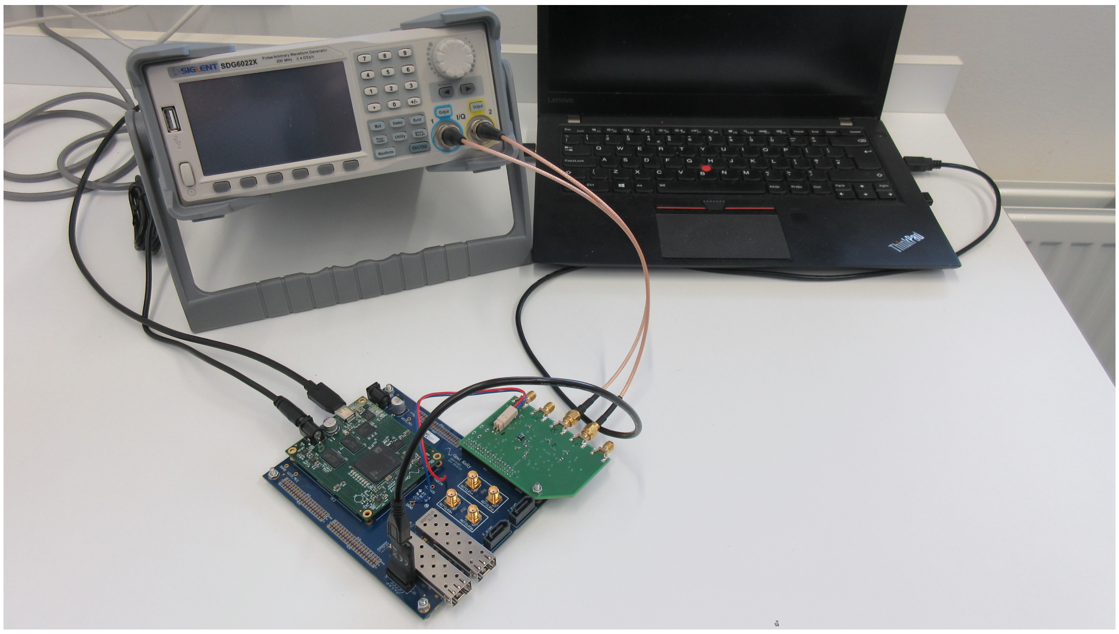
However, the advantage of this approach is also a major disadvantage. To verify the functionality of the design, the presence of the hardware is necessary. Another disadvantage is the necessity to rebuild the bitstream anytime a change is introduced to the code, including the reconnection of the ILA to different signals. As the design grows, this can and will impose a growing time penalty on the development process. Additionally, the ILA instance uses a significant amount of BRAM to store the collected data, which is already taken by the main buffer and the per-channel buffers. To combat this, the main buffer size was limited to 2048 samples (8 KiB).

The first issue of hardware necessity for testing can be somewhat mitigated by setting up a network connected debugged through the use of a Xilinx hardware server. This piece of software then allows a Vivado instance to connect to it as if it was a locally connected JTAG interface.

Despite these problems, this approach was the most suitable in this case due to the size of the project.

In Figure 6.5, the complete device can be seen connected to the Sigilent signal generator and to a PC via JTAG and the Front Panel interface (USB-C connection on the FPGA SoM).

Figure 6.5 Testing setup with signal generator



6.3 Development environment, build configuration and tools

6.3.1 Language and tools

The most important thing related to the actual development is the HDL used. A few options exist, such as VHDL, Verilog, and SystemVerilog. Due to the author's personal preference, only the two Verilog-based languages were considered. While it is feasible and sensible to develop the entire thing using pure Verilog, SystemVerilog became the language of choice for design and eventual testing. The reasoning behind this choice was that utilizing a more modern version would enable the use of numerous quality-of-life improvements introduced (such as *interfaces*), which would lead to more concise and legible code besides being a good learning and practice opportunity. For testing, the Python-based *cocotb* framework was considered. Is was abandoned in the end as the volume of tests done needed to be larger to justify the use of additional tools.

6.3.2 Build

The choice of build tools was straightforward as no practical, usable tools exist for creating the actual FPGA bitstream from user code besides the vendor-provided. While there are alternatives for certain parts of the build process (mainly synthesis), the tooling is not harmonized and compatible. This leaves us with Xilinx Vivado, *the* tool of choice for Xilinx FPGA development.

Vivado offers 2 ways to build a bitstream from SystemVerilog source code, called the *Project Mode* and *Non-Project Mode*[15] and both of them were initially considered.

Project Mode

In project mode, Vivado manages and organizes the source code and all other related configuration. A directory which contains all the aforementioned is created for this purpose. Vivado is also responsible for building the source code into the final bitstream. The user uses the supplied GUI interface to develop the project. This includes writing source code (this can also be done using external tools), testing, specifying design constraints, debugging, and building the bitstream. This mode relies on a `.xpr` file to contain all the project configuration. In is located in the root directory of the project.

The main advantage is user-friendliness, as all the internal details of the source management, build versioning, IP core management, and others are done automatically. On the other hand, this sometimes results in unexpected and hard-to-explain behavior, which is difficult to debug due to the abstract nature of the mode.

Non-Project Mode

It is not straightforward (or even possible) to create a simple `Makefile` to build the bitstream directly. The Vivado command line interface is not created in such a way. Luckily, for power users and anyone who prefers to avoid GUI tools in favor of a simple scripted approach (such as the author), Vivado offers a simpler alternative. The Non-Project Mode lets the user specify a `.tcl` script which contains all the commands used to load the sources and build the bitstream. This can then, if desired, be included in aforementioned `Makefile` to create an illusion of development experience devoid of bloated vendor-specific software (Vivado actually creates a project in memory, but the user is not exposed to it [15]).

The main advantage of this mode is complete control of the build process. The user can specify which build artifacts will be generated and their destination. It is also possible to specify the order in which the sources will be compiled. This is often necessary when using some SystemVerilog features (such as interfaces). While this is also possible in the GUI, finding and adjusting said order is incomparably more difficult.

There are also some disadvantages. The user cannot only configure everything but is required to. There is a plethora of settings and tweaks that have to be set in order to achieve a reliable compilation. This can be a daunting task for a novice.

For our purposes, the Non-Project Mode was selected.

6.3.3 Project structure

The FPGA project structure is as follows. The compilation generates many artifacts and reports along the way. This is useful for debugging and general assessment of timing and utilization. Compilation checkpoints are also generated, which, in theory, could be used to resume a failed compilation while reusing all the successful steps from before. This feature is not used.

The build script is also responsible for updating constants in the source code, which are dependent on variables known only at the compile time (git SHA).

The project structure is as follows:

```
├── opalkelly_hdl ........................................ Opal Kelly HDL library
├── rtl ........................................ design source code directory
│   └── *.v/*.sv_util.rpt ........................................ source files
├── sim ........................................ simulation source code directory
│   └── tp_*.sv_util.rpt ........................................ test procedure files
├── build ........................................ build directory (generated)
│   ├── clock_util.rpt .............................. clock utilization report (generated)
│   ├── post_imp_drc.rpt ...................... post implementation DRC report (generated)
│   ├── post_place.dcp .............................. post place checkpoint (generated)
│   ├── post_place_timing_summary.rpt ................ post place timing report (generated)
│   ├── post_route.dcp .............................. post route checkpoint (generated)
│   ├── post_route_power.rpt ...................... post route power usage report (generated)
│   ├── post_route_timing.rpt ...................... post route timing report (generated)
│   ├── post_route_timing_summary.rpt ............. post route timing summary (generated)
│   ├── post_route_util.rpt ...................... post route utilization report (generated)
│   ├── post_synth.dcp .............................. post synthesis checkpoint (generated)
│   ├── post_synth_power.rpt ...................... post synthesis power report(generated)
│   ├── post_synth_timing_summary.rpt ........... post synthesis timing summary (generated)
│   ├── temporal_ingestor_impl_netlist.v .......... synthesized netlist in Verilog (generated)
│   └── temporal_ingestor.bit ...................................... bitstream (generated)
├── build.sh ............................... build script including dynamic code generation
├── cleanup.sh ........................................................ cleanup script
├── setupenv.sh .................. environment setup script adds vivado command to PATH
├── buildall.tcl ...................................... Non-Project Mode tcl build script
└── constraints.xdc ............................................... project constraints
```

To run the FPGA build, the user shall first edit the `setpenv.sh` script to point towards a valid Vivado installation. Then, to compile the the project, `build.sh` shall be run. The script updates a file inside the source directory, which contains the git SHA of the current version, and then commences the compilation. The build directory is created within which the resulting bitstream is located.

Chapter 7
Software

The supporting software is an inseparable part of a user-friendly device that should be connected to and controlled from a PC. This software should be responsible for the configuration of the device, for the reception of the received samples, preprocessing, and storing them in a file.

Reading the data from the pipe and processing it should also be done in separate threads. This is preferred due to the smaller size (64 KiS) of the buffer in the FPGA, which could cause data when not read out sufficiently frequently, even at moderately low speeds.

7.1 Library versions

Opal Kelly provides a supporting library that provides an interface enabling the use of the primitives, which can be used to connect the software part to the hardware. This library is available in a multitude of programming languages such as C++, Python, and Java, among others, some of which are more suitable for our case than others.

7.1.1 Python

The provided Python library uses the C++ library underneath. The C++ library interface is made available using the SWIG interface compiler, and therefore, there are no concerns regarding the transfer speeds using a scripting language such as Python.

The issue lies with the necessity to use two separate threads, one to handle the pipe transfers and the other to process and save the data. The most common Python implementation, called CPython, has limited multithreading capabilities due to the presence of the GIL [16], the global interpreter lock, which "is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once."[16]

This means that the program can run only 1 thread at a time. While multiprocessing could fix this, it will be much easier to use a different programming language altogether.

7.1.2 Java

The main issue with Java is that the language uses a garbage collector instead of manual memory deallocation. While this provides increased comfort for the programmer, it can cause sporadic delays in processing. This is especially true for programs that allocate and free large quantities of memory, such as in our case.

7.1.3 C++

Despite being the lowest-level programming language of the bunch, Opal Kelly provides the interface library, so C++ does not suffer from the issues found in the other programming languages mentioned above.

C++ natively supports multithreading and is well-equipped with facilities to manage threads and safely move data between them. The garbage collector issue is, too, nonexistent. C++ requires the user to clean up memory manually, and when the RAII technique is used, all worries of leaked memory are mitigated.

Another advantage of using C++ is that the Opal Kelly library is written in C++ and only ported to other languages. While this does not incur any performance penalty or other problems, using the intended interface directly is more comfortable.

Ultimately, these factors are why C++ became the language of choice for the software part of the implementation.

7.2 Implementation

The initial idea was to create a full-fledged GUI application to control the device. With worries that this would be way too large of an undertaking due to a lack of expertise in GUI design, the idea was split into two.

Library

First, a C++ library for controlling the device was created. This library is responsible for configuring the device and extracting the data, serving as a wrapper around the Opal Kelly library with device-specific features added, such as the configuration for the TDC chip and the reference DAC. The library handles the data transfer from the device with all the required multithreaded code. This means that the library would buffer samples internally and provide them upon request via an interface not unlike the interfaces provided by the containers in the standard C++ library.

The current implementation uses a

```
std::deque<t_sampleblock_ptr>
```

where

```
t_sampleblock_ptr
```

is an

```
std::unique_ptr<std::vector<unsigned char>>
```

for transferring the data between the threads. The logic behind choosing the `unique_ptr` for transferring data was the implication of ownership semantics and reduction of data races when accessing a shared byte buffer to a minimum. An additional benefit is the possibility of avoiding copying the freshly downloaded data due to the use of C++ move semantics. The time spent accessing shared resources is, therefore, limited to pushing and popping a single smart pointer for each of the blocks, which are frequently tens of kilobytes large.

Also, waiting for the shared queue to be filled is implemented through the means of a condition variable and a mutex.

A minor improvement could be the replacement of always newly allocated blocks with pre-allocated reusable buffers, which would be alternated between used and empty states and passed between the two threads.

As mentioned before, the majority of the configuration logic is handled by the software, and the FPGA logic only exposes specific registers. A few examples can illustrate this. For example, instead of mapping the configuration address space of the TDC chip and the reference DAC into the register address space or even abstracting the configuration on a higher level, the device forwards the control of the SPI interface and leaves full control to the software. Another example could be the syncing of the sample format sent over the data lines from the TDC chip to the FPGA, the *stop bit count* and *reference index bit count*. These values must match between the TDC chip and the FPGA, and the software is responsible for configuring both. These abstractions are done by the library, and the user/user software developer does not handle this manually.

User software

The current version of the user software supports all the basic functionality to support the upload of a bitstream, configuration of the TDC chip and the reference, downloading the samples, and storing them in a file for a specified time duration. A rudimentary yet user-friendly interface is available per the help message in Code listing 7.1. The user can specify which connected device will be used with a unique Opal Kelly assigned serial number, the bitstream that will be uploaded to the FPGA (or disable programming at all), the destination file where the collected timestamps will be written to, and time for which the collection should happen.

The code also contains provisions for measuring the throughput. When a special bitstream is uploaded, which generates ascending sequences of timestamps, verifying that no samples are lost is also possible. This is not readily available to the user and has to be enabled in the source code and recompiled.

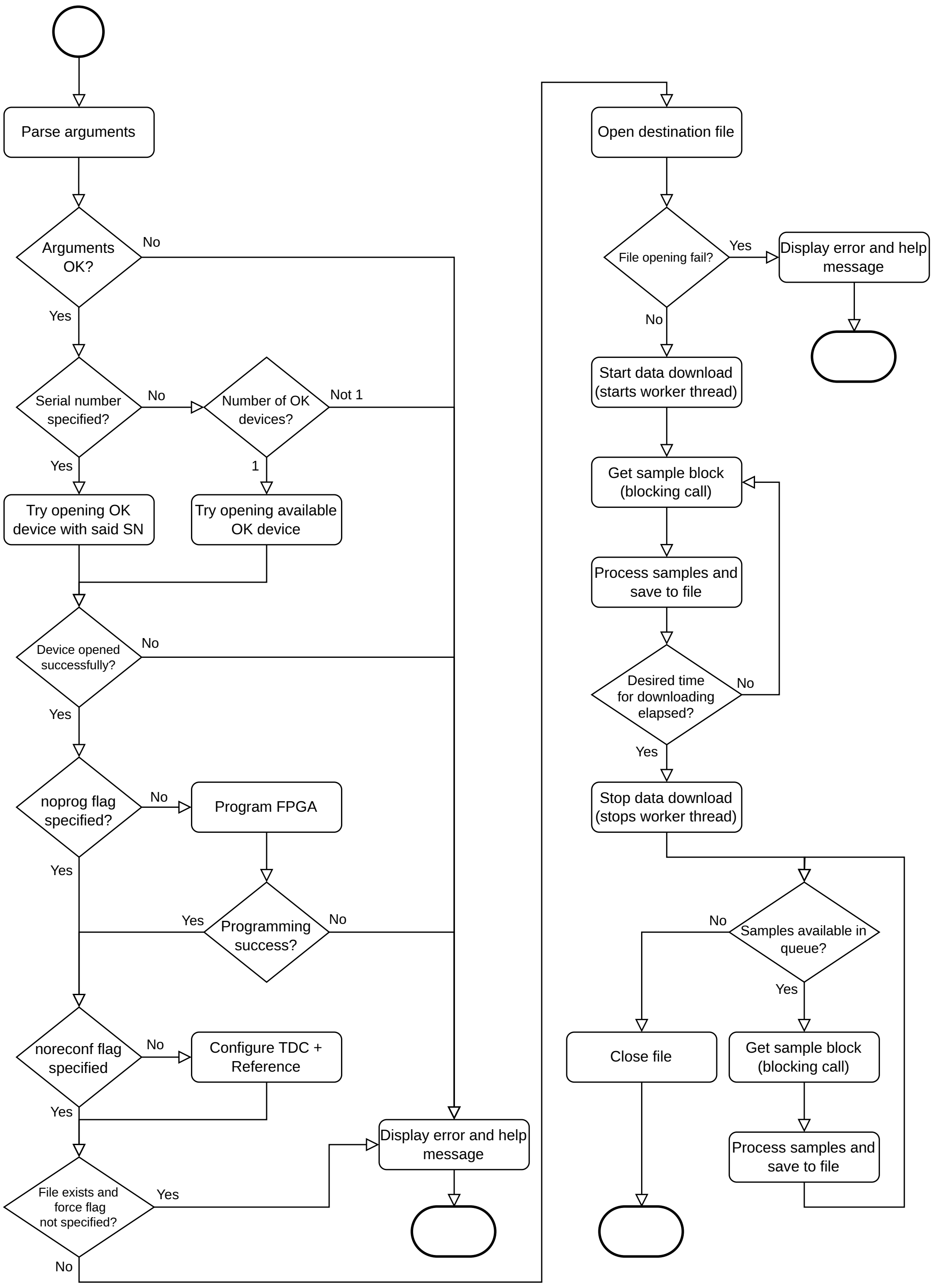
After verifying the inputs and environment, as shown in Figure 7.1, the main thread starts a worker thread responsible for initiating transfers from the FPGA. It applies the algorithm described in the communication chapter and retrieves blocks of samples in a loop. Whether the thread continues execution is controlled by a flag controlled externally from the main thread. When the flag is deasserted, the thread finishes the current transfer and exits. Similar behaviour occurs in the main thread where the block queue is first emptied, and only then does the program exit.

Code listing 7.1 User software help message

```
OPTIONS:

    -o[dest_file],
    --output=[dest_file]                Destination file
    -d, --debug                         Enable debug print
    Bitstream
      --noprog                            Do not upload new bitstream
      -b[bitstream],
      --bitstream=[bitstream]             Bitstream file
    --noreconf                          Do not reconfig TDC and REF
    -s[serial], --serial=[serial]       OK device serial
    -t[time], --time=[time]             Download time in seconds
    -f, --force                         Force file overwriting
```

Figure 7.1 User software flowchart


Parse arguments
Arguments OK?
No
Yes
Serial number specified?
No
Number of OK devices?
Not 1
Yes
1
Try opening OK device with said SN
Try opening available OK device
Device opened successfully?
No
Yes
noprog flag specified?
No
Program FPGA
Yes
Yes
Programming success?
No
noreconf flag specified
No
Configure TDC + Reference
Yes
File exists and force flag not specified?
Yes
No
Display error and help message
Open destination file
File opening fail?
Yes
Display error and help message
No
Start data download (starts worker thread)
Get sample block (blocking call)
Process samples and save to file
Desired time for downloading elapsed?
No
Yes
Stop data download (stops worker thread)
No
Samples available in queue?
Yes
Close file
Get sample block (blocking call)
Process samples and save to file

Chapter 8

Results and performance

Testing was performed in terms of data correctness, lost sample detection, and bandwidth capability. It had to be settled on a data format and hardware configuration that would be used. This configuration is summed up in Table 8.1, which also portrays all the possible configurations for the two sections of the output data. The configuration with 24 reference index bits and 14 stop bits was used. The testing was done in two parts in order to simplify debugging should any problems arise. First, the main buffer and the USB interface were tested. Then, the whole system, including the TDC chip, was tested. Also, a frequency for running the TDC-to-FPGA interface had to be set. The interface is run at 100 MHz at SDR.

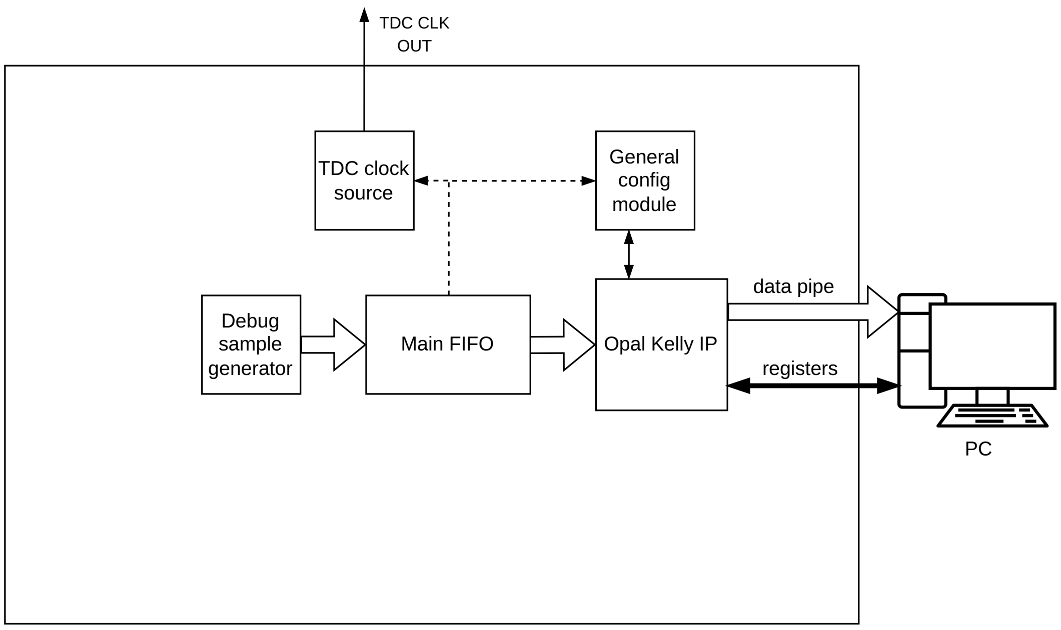
Table 8.1 TDC sample formats

REF_INDEX_WIDTH	STOP_DATA_BITWIDTH
0 bits	**14 bits**
2 bits	16 bits
4 bits	18 bits
6 bits	20 bits
8 bits	
12 bits	
16 bits	
24 bits	

8.1 Main buffer and USB interface testing

A special version of a channel was created to test that no samples are lost between the input to the main buffer and the PC. This version, as portrayed in Figure 8.1, is a stripped version of the variant portrayed in chapter FPGA design Figure 6.1. This debug channel does not generate samples compliant with the format outlined in the communication chapter. Instead, it is a mere integer incremented by 1 between the samples. This sequence can then be checked in the PC and verified that no samples are lost. Through this method, it was possible to achieve sample rates of up to 10 MS/s with no loss. When this throughput was exceeded, sample loss started to be detected. It was impossible to achieve this speed using a single-threaded application on the PC, as saving the data stream to even a sufficiently fast NVMe SSD was causing too large of a delay.

Figure 8.1 Stripped down debug schematic



8.2 Channel throughput testing

To test the whole data path from the channel input to the PC application, it was necessary to come up with a solution capable of driving the inputs. It must be possible to set the output parameters such that pulses of acceptable duty cycle and voltage are output. It must also be possible to control the frequency on the go to test various throughput scenarios. A two-channel Sigilent SDG series function / arbitrary waveform generator was used for this.

The `DROPPED_SAMPLES` register was monitored to test that no samples are dropped.

The results are rather underwhelming and point to a critical flaw in the design of the arbiter. The maximal achieved sample rate was 2.5 MS/s per channel. Moreover, this value was also independent of the amount of samples passing through the other channels. This means that either the arbitration circuitry is not effectively using the dead time from less loaded channels or there is some other problem. The implemented round-robin is unsuitable for such a case with unbalanced input throughput and inherently leads to decreased performance.

The reader could be led to think that the speed is limited by the 100 MHz SDR interface, which, at 38 bits per sample, limits the speed to a suspiciously close value of 2.63 MS/s. This is not the case, though as the circuitry responsible for detecting dropped samples (samples that could not be inserted into the per-channel FIFO) started triggering rapidly at about 2.5 MS/s

The described behaviour was also confirmed with a custom microcontroller-based pulse generation solution, which was able to load all 4 channels. In this manner, the throughput described in the previous section was achieved.

Chapter 9
Future work

Despite multiple flaws found in the design, many lessons were learned, and improvements can be suggested and included in further developments.

9.1 Clock generation and division

Despite being functional, the current method of clock usage needs to be corrected from the point of good engineering practices. In the design, the clock needs to be divided and multiplexed, and this is done in the user logic instead of dedicated resources connected to the provided clock interconnects. While this is feasible for slower clocks (while remaining a poor engineering practice), the generated clock may have worse characteristics, such as higher jitter and less predictable latency. It also places higher demands on the place and route algorithm and pushes the timing constraints further to their limits.

A preferred way to achieve clock division is using a `MMCME2` or `PLLE2` primitives found in the FPGA. If used with the interface clock, it would yield finer control over the resulting clock and higher reliability. Both of the clock synthesis primitives contain facilities for fractional division of the clock. If desired, dynamic reconfiguration can also be done, albeit at the price of additional logic for providing the configuration and dynamic reset for when the clock value is supposed to change as the PLL loses a lock on the generated clock. Additionally, if the PLL is fed using an external clock, the input frequency would have to be fixed, and the phase dependency on the clock would have to be measured if necessary.

For multiplexing, the clocking resources include the `BUFGMUX` primitive, which is a double input clock buffer with a select line capable of switching between the clocks.

9.2 Channel arbiter

A significant oversight was the utilization of the round-robin approach for choosing from which channel a sample will be moved to the main buffer. In this way, the arbiter loops over the channels, attempting to move a sample into the main buffer. In the worst case, 3/4 of the runtime is wasted when only one channel is receiving samples. An improvement could be to skip the empty channels, resulting in no time wasted but still not accounting for different rates at which the channels could receive data.

A way to mitigate that could be prioritization of channels with fuller buffers. This can not be done directly, though, due to the latency of the `rd_data_count`, a signal present on the FIFO denoting the number of words that can be read. Usually, the signal is updated a few (usually 1 or 2) clock cycles after the data has been written to the FIFO. The utilization of the `empty` signal would be necessary to prevent FIFO underruns.

9.3 Generation of overflow tags

To enable theoretically infinite runtimes, the device must be capable of signaling to the application that the counter has overflowed. Ideally, the TDC chip would be responsible for this feature, but that is not true with TDC-GPX2. This means that it will have to be handled in the FPGA through an auxiliary counter, which would be kept in sync, albeit slightly advanced, due to the delay of the reference clock between the FPGA and the TDC chip.

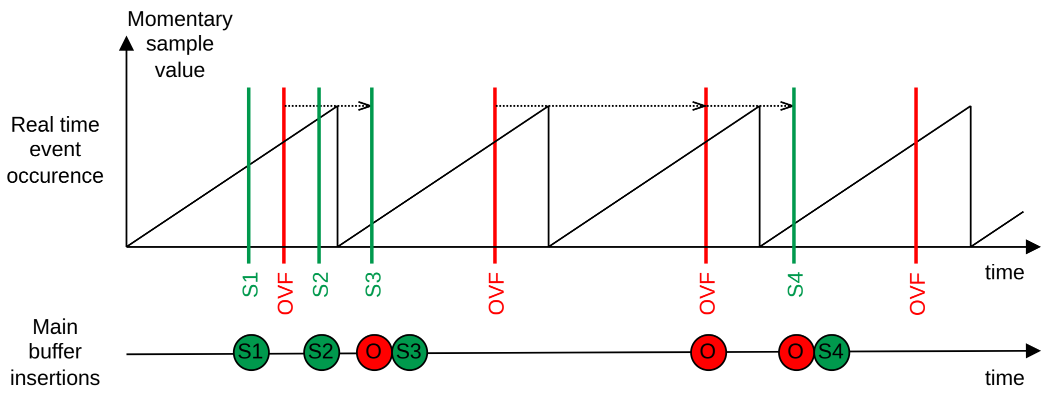
When this counter overflows, the special message described in the sample format section is inserted into the sample stream to be detected by the application.

In this case, the devil lies in the details (or delays). There are 2 sources of additional delay. First, the reference clock is delayed slightly from the FPGA to the TDC chip. Second, when the overflow occurs, the data is still inside the TDC chip and will arrive after another delay. This means the overflow mark would be inserted into the data stream too early, corrupting timestamps that arrive between the mark and the actual overflow (1 reference clock period is added to them)

It should be noted that it is not necessary to insert the overflow in the stream at a precise time but only at a precise location, and therefore, it can be suitably delayed. First, we must detect from the data that the data after the overflow. We will take advantage of the strictly rising values of the timestamps. We know the overflow must be inserted as soon as a change from a high to a low timestamp value is detected. This shift in the insertion time is portrayed in Figure 9.1 by the dotted arrows above the top graph. However, what about situations when there are no arriving timestamps and, therefore, no high-to-low timestamp value transfers? When a second overflow is detected, the previous delayed overflow is inserted while an overflow mark is still stored. Any subsequent marks can then be inserted as soon as an overflow is detected. The previous algorithm resumes as soon as data is detected.

The good thing is that this can be mitigated both at runtime and even during post-processing in the application. During post-processing, the overflow marks are merely shifted forward to the nearest high-to-low time value transition.

Figure 9.1 Overflow delay adjustment explanation



9.4 GUI software

Instead of providing only a command line interface, scientific instruments commonly use a GUI application for control, monitoring, and data processing.

Many suitable libraries exist, and the fact that the underlying control library is written in C++ allows the use of different programming languages (such as C++, Python and others) thanks to compatibility libraries such as SWIG.

To maintain the ease of high-performance processing, it would probably be the easiest to utilize a C++ native framework such as Qt, GTK, CEF, or wxWidgets.

9.5 Higher throughput

As of now, the FPGA is the throughput bottleneck. There are multiple issues:

Small buffer

Currently, the majority of the FPGA BRAM is allocated to the main buffer, with the size being 64 KiS. This means that the maximal block download size is 65535 words (256 KiB). Since initiating the operation has a certain overhead, this limits the bandwidth to around 10 MS/s. To increase the speed further, a larger memory block would have to be implemented, requiring the larger FPGA variant, which could fit a buffer double the size.

Alternatively, the DDR3 memory chip could be utilized as an additional buffer. This would require additional rather complex control logic to transfer the data to and from the memory chip. There is a possibility of a higher sample availability latency, but this could be possibly solved with a buffering scheme built inside the FPGA BRAM accompanying the DDR3 memory.

Low TDC interface clock

Hand in hand with the problematic clocking scheme is the TDC chip interface. This interface can be theoretically run at 200 MHz DDR, resulting in a theoretical maximal bitrate equal to 1.6 Gb/s. This amounts to, assuming the sample size of 38 bits (14 reference index + 28 stop bits) used for the testing to just above 42.1 MS/s. Even higher sample rates can be achieved should the precision be dropped.

The clock speed is limited mostly by the fact that the same clock is used to run the entire receiving logic.

Conclusion

Through the means of using a commercially available time tagging chip and FPGA-based supporting and interfacing logic, a useable Time-to-Digital instrument was created. This instrument is capable of sampling 4 channels at up to 2.5 MS/s. With improvements to the internal logic, an FPGA with a higher amount of BRAM, and a faster TDC chip to the FPGA interface, higher sample rates can be achieved as the USB interface is sufficiently fast. The interface between the TDC chip and FPGA should be capable of just above 40 MS/s at the highest speed.

A controlling and sample download command line application was created to accompany the hardware implementation. The application is written in C++ to accommodate latency requirements easily and allow for a multithreaded approach. It takes advantage of an interface library provided by Opal Kelly, the manufacturer of the FPGA module. The configuration was done through the library using a register interface that provides an address space that can be mapped internally in the FPGA to various functions and parameters.

The development was accelerated using Xilinx Integrated Logic Analyzer, through which it was possible to test the proper operation of the device.

The device is unsurprisingly cheaper to produce than it is to purchase the commercial solutions. This does not factor in the price of labour, which would make up a considerable part of the price of the commercial devices.

Appendix A

Appendix

Code listing A.1 Internal configuration interface

```
interface regbus_if;
    logic[23:0]      address;
    logic[31:0]      data_in;
    logic[31:0]      data_out;
    logic            read;
    logic            write;
    logic            enable;
    logic            busy;

    modport master
    (
        output  address,
        output  data_in,
        input   data_out,
        output  read,
        output  write,
        output  enable,
        input   busy
    );

    modport slave
    (
        input  address,
        input  data_in,
        output data_out,
        input  read,
        input  write,
        input  enable,
        output busy
    );
endinterface : regbus_if
```

Bibliography

1. RAZAVI, Behzad. *Design of Analog CMOS Integrated Circuits.* 2nd ed. New York, NY: McGraw-Hill Professional, 2016.
2. BARON, Robert G. The Vernier Time-Measuring Technique. *Proceedings of the IRE.* 1957, vol. 45, no. 1, pp. 21–30. Available from DOI: `10.1109/JRPROC.1957.278252`.
3. DUDEK, Piotr; SZCZEPAŃSKI, Stanislaw; HATFIELD, J.V. A high-resolution CMOS time-to-digital converter utilizing a Vernier delay line. *Solid-State Circuits, IEEE Journal of.* 2000, vol. 35, pp. 240–247. Available from DOI: `10.1109/4.823449`.
4. QUTOOLS GMBH. *quTAU* [online]. [visited on 2023-12-28]. Available from: `https://qutools.com/qutau`.
5. PICOQUANT. *TCSPC and Time Tagging Electronics* [online]. [visited on 2023-12-28]. Available from: `https://www.picoquant.com/products/category/tcspc-and-time-tagging-modules`.
6. SWABIAN INSTRUMENTS. *Time Tagger Series* [online]. [visited on 2023-12-29]. Available from: `https://www.swabianinstruments.com/time-tagger/`.
7. ID QUANTIQUE. *ID1000 Time Controller Series* [online]. [visited on 2023-12-29]. Available from: `https://www.idquantique.com/quantum-sensing/products/id1000-time-controller/`.
8. SCIOSENSE B.V. *TDC-GPX2 Time-to-Digital Converter* [online]. [visited on 2023-09-02]. Available from: `https://www.sciosense.com/wp-content/uploads/documents/TDC-GPX2_DS000473_3-00.pdf`.
9. ANALOG DEVICES, INC. *MAX5713/MAX5714/MAX5715 Ultra-Small, Quad-Channel, 8-/10-/12-Bit Buffered Output DACs with Internal Reference and SPI Interface* [online]. [visited on 2023-09-02]. Available from: `https://www.analog.com/media/en/technical-documentation/data-sheets/max5713-max5715.pdf`.
10. OPAL KELLY INCORPORATED. *USB 3.0 HDL* [online]. [visited on 2023-11-27]. Available from: `https://docs.opalkelly.com/fpsdk/frontpanel-hdl/frontpanel-hdl-usb-3-0/`.
11. XILINX. *UG953 - Vivado Design Suite User Guide* [online]. [visited on 2023-12-28]. Available from: `https://docs.xilinx.com/r/en-US/ug953-vivado-7series-libraries`.
12. XILINX. *Artix 7 FPGA Product Table* [online]. [visited on 2023-12-28]. Available from: `https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html#productTable`.
13. OPAL KELLY. *XEM7310MT FPGA Development Board with AMD-Xilinx Artix-7* [online]. [visited on 2023-12-28]. Available from: `https://opalkelly.com/products/xem7310mt/`.

14. XILINX. *XMP100 - Cost-Optimized Portfolio Product Selection Guide* [online]. [visited on 2024-01-04]. Available from: `https://docs.xilinx.com/v/u/en-US/cost-optimized-product-selection-guide`.
15. XILINX. *UG892 - Vivado Design Suite User Guide* [online]. [visited on 2023-12-28]. Available from: `https://docs.xilinx.com/r/en-US/ug892-vivado-design-flows-overview`.
16. *GlobalInterpreterLock* [online]. [visited on 2023-12-08]. Available from: `https://wiki.python.org/moin/GlobalInterpreterLock`.

Contents of the attachment

hw		hardware source code directory
	opalkelly_hdl	Front Panel HDL library directory
	rtl	FPGA source code
	sim	simulation source code
	build.sh	build script
	buildall.tcl	Vivado tcl. script for compilation
	cleanup.sh	cleanup script
	constraints.xdc	project constraints
	README.md	brief description and guide on FPGA code
	setupenv.sh	environment setup script
sw		application source code directory
	libs	Front Panel C++ library
	*.h, *.hxx, *.cpp	application source code
	CMakeLists.txt	CMakeLists file for building the project
thesis		thesis source directory
	drawings	draw.io drawings
	graphics	photos and figures used in thesis
	text	source text of the thesis
	assignment-include.pdf	thesis assignment
	bib-database.bib	bibliography
	ctufit-thesis.cls	FIT CTU thesis template
	ctufit-thesis.tex	main source tex file
out		
	temporal_ingestor.bit	compiled bitstream
	tidown	compiled application binary for x86_64 Linux
	thesis.pdf	thesis pdf