Warsaw University of Technology

FACULTY OF
ELECTRONICS AND INFORMATION TECHNOLOGY

Institute of of Electronic Systems

Bachelor’s diploma thesis

in the field of study Electronics
and specialisation Electronics and Computer Engineering

Precise PID controller for quantum applications

Jakub Matyas
student record book number 277464

thesis supervisor
Krzysztof Poźniak, Ph.D., D.Sc.

WARSAW 2020

Precise PID controller for quantum applications

Abstract. Realization of the world's first Bose–Einstein Condensate in 1995 has opened a door for conducting research in the field of quantum physics on the atomic or even sub–atomic scales. Since then, a number of scientific laboratories around the world have started performing quantum physics experiments. Running this kind of experiment, however, requires advanced hardware and software to control its various aspects such as driving laser sources used to trap atoms or ions. In this thesis, a design of a PID (proportional–integral–derivative) controller for such applications is described. It aims to be compatible with the ARTIQ (Advanced Real–Time Infrastructure for Quantum physics) control system and Sinara hardware, both of which are open–source projects. The developed PID controller provides an input and output voltage range of ±10 V. Its measured bandwidth and latency are no worse than 1 kHz and 1 ms, respectively.

Keywords: PID controller, ARTIQ, Sinara, frequency lock loop, real–time control system

Precyzyjny kontroler PID do zastosowań kwantowych

Streszczenie. Zaobserwowanie w 1995 r. po raz pierwszy kondensatu Bosego–Einsteina umożliwiło przeprowadzanie badan z dziedziny fizyki kwantowej w skali subatomowej. Od tego momentu, wiele laboratoriów naukowych na całym świecie rozpoczęło realizacje eksperymentów kwantowych. Jednakże przeprowadzanie tego typu badan wymaga użycia zaawansowanego sprzętu i oprogramowania, które umożliwiałyby sprawowanie kontroli nad rożnymi aspektami eksperymentów. Za przykład może posłużyć kontrolowanie laserów, które są używane do oświetlania chmury atomów i jonów w celu złapania ich w pułapki magnetyczno–optyczne. W niniejszej pracy opisany został proces realizacji regulatora PID (proporcjonalno–całkująco–różniczkującego, ang. *proportional–integral–derivative*) stworzonego miedzy innymi do wspomnianych wyżej zastosowań. Został zaprojektowany z założeniem kompatybilności z systemem ARTIQ (Advanced Real-Time Infrastructure for Quantum physics, pol. Zaawansowana Infrastruktura Czasu Rzeczywistego dla Fizyki Kwantowej) i sprzętem z rodziny Sinara. Zrealizowany regulator PID zapewnia poprawna prace i regulacje przy zakresie napiec wejściowych i wyjściowych ±10 V. Zmierzone pasmo i latencja są nie gorsze niż, odpowiednio, 1 kHz i 1 ms. Pętla synchronizacji częstotliwościowej

Słowa kluczowe: regulator PID, ARTIQ, Sinara, pętla synchronizacji częstotliwościowej, system sterujący czasu rzeczywistego

załącznik nr 10 do zarządzenia
nr 46 /2016 Rektora PW

Politechnika Warszawska
Warsaw University of Technology

……….......................
miejscowość i data
place and date

……………………………..
imię i nazwisko studenta
name and surname of the student

……………………………..
numer albumu
student record book number

……………………….………
kierunek studiów
field of study

OŚWIADCZENIE

DECLARATION

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.
Under the penalty of perjury, I hereby certify that I wrote my diploma thesis on my own, under the guidance of the thesis supervisor.

Jednocześnie oświadczam, że:
I also declare that:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- *this diploma thesis does not constitute infringement of copyright following the act of 4 February 1994 on copyright and related rights (Journal of Acts of 2006 no. 90, item 631 with further amendments) or personal rights protected under the civil law,*
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- *the diploma thesis does not contain data or information acquired in an illegal way,*
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- *the diploma thesis has never been the basis of any other official proceedings leading to the award of diplomas or professional degrees,*
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- *all information included in the diploma thesis, derived from printed and electronic sources, has been documented with relevant references in the literature section,*
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.
- *I am aware of the regulations at Warsaw University of Technology on management of copyright and related rights, industrial property rights and commercialisation.*

załącznik nr 10 do zarządzenia
nr /2016 Rektora PW

Politechnika Warszawska
Warsaw University of Technology

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.
I certify that the content of the printed version of the diploma thesis, the content of the electronic version of the diploma thesis (on a CD) and the content of the diploma thesis in the Archive of Diploma Theses (APD module) of the USOS system are identical.

....
czytelny podpis studenta
legible signature of the student

Contents

1. Introduction

The introduction of the quantum theory in the early 20th century and its extensive confirmation within the next few decades have been a major step towards understanding the principles of the quantum world and, what is probably even more exciting, the Universe itself. When a gaseous Bose–Einstein Condensate (BEC) was realized in 1995 for the first time [1], a door to study particle's world not only theoretically, but also experimentally, was opened as wide as never before. After over 70 years of research, Einstein's prediction regarding condensation of atoms has been finally confirmed. The team led by Eric Cornell and Carl Wiemann at the JILA laboratory in Boulder, Colorado, having achieved world first gaseous condensate in a gas of rubidium-87, laid the foundation for the field of cold atoms and a new branch of atomic physics. Since then, over 50 research groups across the globe have made BECs [2] and until 2009 a BEC has been achieved for 12 atomic species [3].

Bose–Einstein Condensate may be in principle described as a state of matter in which a large number of identical particles occupy a single [3]. This means that when a BEC occurs, multiple atoms behave as a single entity which allows scientists to study the complicated world of quantum physics by observing and conducting research on a 'superatom', instead of extremely small single atoms [4]. Being able to explore the behaviour of systems at atomic and subatomic length scales is essential to make significant discoveries and advancements in modern science and technology. Even though the filed of cold atom research is still in its early years of development, studies conducted on BEC have already made an impact on today's industry and science. In particular, Bose–Einstein Condensates are used in quantum experiments aiming to achieve the next generation of improved atomic clocks with better accuracy, based on optical transitions instead of microwaves ones [5]. Furthermore, due to BEC being a coherent wave, an atomic laser may be produced soon and used, e.g. in the process of nanolithography or applied to any application that requires well–controlled beams of atoms [6]. Atomic interferometry based on BEC has also proved itself a versatile tool for measuring inertial forces or physical constants [5]. Even though these applications are already extremely useful, Bose–Einstein Condensate may be applied in the field that has gained the exceptional popularity throughout the last few years, and which is believed to be a breakthrough in the field of computational power: quantum information. On one hand, following Richard Feynman's suggestion that neither the world nor quantum mechanical effect should be simulated by a classical computer because they are not classical [7], a quantum simulation is developed [8]. On the other hand, BEC is used as a platform for processing qubits with the aim to build a quantum computer [5].

It is clear that even though the first Bose–Einstein Condensate was realized 25 years ago, there is a lot yet to be discovered in the field of quantum mechanics Thanks to achievements of previous generations of scientists, the world is on the verge of the 'Second Quantum Revolution'. The timing is perfect, as science is facing the challenge of the end of Moore's law, while the world is in desperate need for constant advancement in terms of processing power [9].

1.1. Magneto-optical trap

At the foundation of achieving Bose–Einstein Condensate lays ion or neutral atom cooling and trapping. Although trapping may be performed using for example Ioffe--Pritchard trap or Paul trap (the latter of which works for charged particles [10]), usually the magneto–optical trap (MOT) is used, because of its simplicity of construction and its depth [11]. The concept seems to be straightforward, as William D. Phillips and Harold J. Metcalf once stated [12]:

> 'Atoms are slowed and cooled by radiation pressure from laser light and then trapped in a bottle whose "walls" are magnetic fields. Cooled atoms are ideal for exploring basic questions of physics'

However, the idea of temperature in the sentence above might be misleading and should be taken into careful consideration. Thermodynamics connects temperature to a state's parameter of a closed system in thermal equilibrium and its potential to exchange heat, which is not applicable to laser cooling, because atoms constantly absorb and scatter light. Additionally, light cannot be treated as heat. In this case, temperature is recognized as an atom sample's average kinetic energy, which is reduced by laser light [13]. Therefore, in principle, the first step to achieve cold atoms is to slow down atomic beams. It can be done with the use of momentum transfer that happens when an atom absorbs a photon. Each act of absorption reduces atom's velocity by $\frac{\hbar k}{m}$ (single photon's velocity) [14]. Due to the Doppler effect, the photon's frequency is up-shifted when atoms are propagating towards the laser source. Therefore, to ensure that photons are absorbed only by those atoms, the laser frequency is tuned slightly below the atomic resonant frequency. In order to achieve cooling across all dimensions, it is necessary to illuminate gas of atoms with three pairs of circularly polarized orthogonal laser beams of the proper frequency. Furthermore, to successfully trap atoms, application of a magnetic field is necessary. Laser beams are responsible not only for slowing down atoms, but also, thanks to radiation pressure that converges on the centre of the gas chamber, for creating a force that traps atoms inside. Applying a magnetic field of quadruple distribution acts as a control on this force [11]. Having achieved enough densities after cooling and trapping, certain atoms undergo a transition — one sought after by physicists for decades Bose–Einstein Condensation.

Cooling down and trapping atoms is a process that needs applying a particular frequency of laser beams that is slightly detuned from the atomic resonant frequency. If a laser beam of longer or shorter wavelength was used, fewer atoms would be trapped and achieving BEC would become even more problematic. Therefore, a precise reference laser source is used, with which other lasers are synchronised. To ensure that each laser beam that illuminates a gas of atoms is of the desired wavelength, a frequency locking mechanism is needed.

1.2. Frequency locking

When clouds of atoms are to be cooled down and trapped, a need for a control system that adjusts laser source frequency to the set value arises. Fortunately, even though cooling lasers need to be set to precise values, for many applications phase coherence is not required [15], and simple frequency offset lock is sufficient. Since frequency offset locking is based on a feedback loop, commonly used control systems are based on proportional—integral–derivative (PID) controller. One of the common controlling schemes is to modulate laser light directly with electric current. On the upside, it is simple and robust. On the downside, its non–linearity and heating limits the practical bandwidth [16], while laser output wavelength drifts with modulating current. However, when higher bandwidth and stability is needed, acousto-optic modulation (AOM) is commonly used. A simplified frequency offset locking scheme, used for example in [17], is shown in figure 1.1.

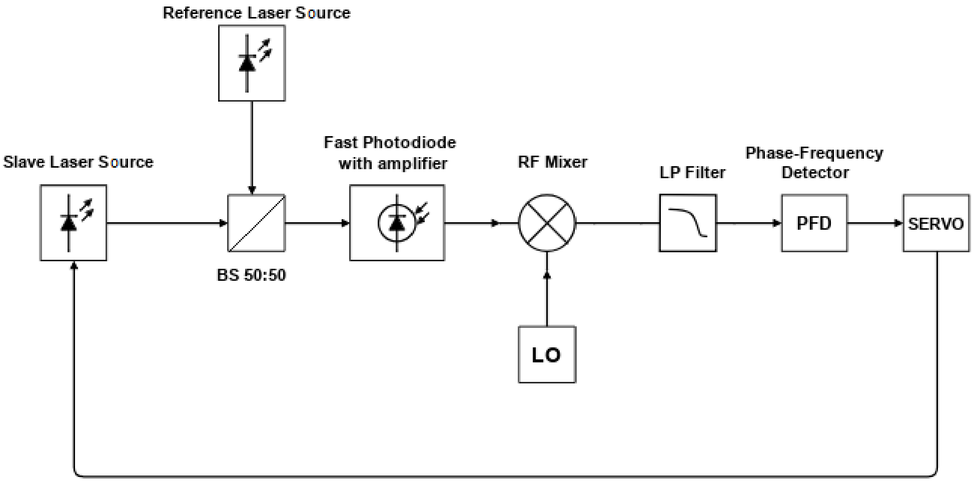


Figure 1.1: Simplified frequency offset locking scheme

Since optical frequencies are usually too high to measure them directly, scaling down is required. It can be done by combining beams from a reference laser source and the controlled one. Achieved optical beat note is then mixed with stable frequency from radio frequency (RF) local oscillator (LO), and their difference is fed into phase-frequency detector (PFD). Based on PFD output, servo, which is a workhorse of the control system, drives the controlled laser appropriately.

1.2.1. Servo

Thanks to its flexibility, ease of implementation and robustness, the PID controller is among the most popular regulators used in electronics. A generic PID controller's application, with its particular parts specified, is shown in figure 1.2.

The PID regulator's mathematical model is well–developed and described in numerous papers. Despite its many advantages, one has to bear in mind that a feedback loop controlling scheme has drawbacks as well. Probably the most noticeable one is its susceptibility to becoming unstable when not designed properly [18]. This property requires a designer to carefully study plant's impulse response and adjust the controller gains (proportional, integral and derivative) in order to avoid *positive feedback*, which may result in uncontrolled behaviour or even the system being damaged. The controller's output signal can be described with the following equation [19]:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}, \tag{1.1}$$

where u is a control variable (CV), e is the error signal and equals to $r - y$, with r being the *setpoint* (SP) — value to which y is desired to converge — and y being the *process variable* (PV) measured from the *plant's* response to the control signal.

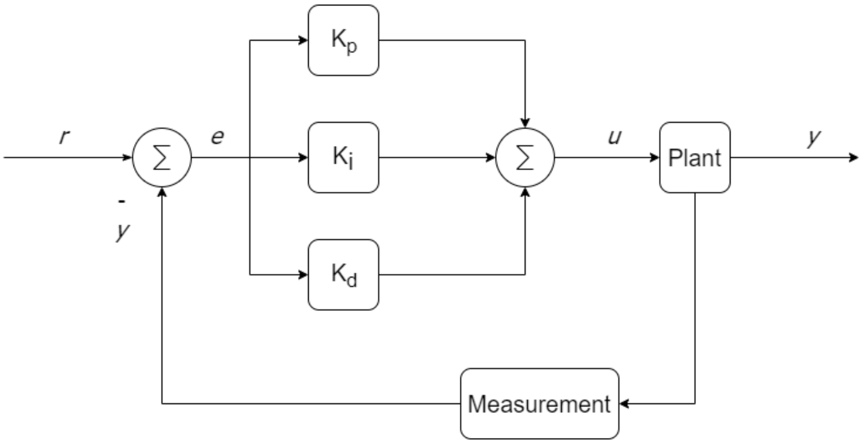


Figure 1.2: A generic closed feedback control loop

The control signal consists of three terms: proportional, integral and derivative. As Hesheng Wang shows in [20], each of them is responsible for a different closed–feedback–loop–system behaviour:

- proportional gain (K_p) makes controller's output proportionally big to *error* signal, however, does not eliminate steady–state error and easily makes the controller prone to becoming unstable;
- integral action (K_i) allows to eliminate steady–state error, but is likely to cause

overshooting — it accumulates past values of *error* signal and based on them acts proportionally to K_i;
- derivative action (K_d) counteracts overshooting and improves the system's stability.

According to Murray and Aström [19], most PID controllers' implementation lacks a derivative part. Therefore, they point out, term *PI controller* is more appropriate in that case. Murray and Aström, however, argue that term *PID controller* may be used as well, as a more general one for this class of regulators. The same convention is followed in this thesis.

1.2.1.1. Implementation of PID controller in FPGA

With coming of the digital era, cheap integrated circuits and high–speed controllers have become available on an unprecedented scale. It was only a matter of time when the industry started using digital control systems (for example field–programmable gate array (FPGA) based). What makes digital control so appealing is mostly its efficiency and ease of implementation.

System analysis and design may be a daunting task when done in the continuous–time domain only, especially when it includes some sort of digital control. To ease that task, system's behaviour is often represented in various domains — each of them aims to provide information about the system's nature or characteristics and is used for different applications and purposes.

While every system exists in the time domain, it is often convenient to consider its response in the complex frequency domain (*S–plane*) [21]. The mathematical tool that allows to transit a signal from continuous–time to continuous–frequency domain is the *Laplace transform*, which is defined as

$$\mathscr{L}\{f(t)\} = F(s) = \int_0^\infty f(t)e^{-st}dt, \tag{1.2}$$

where s is a complex frequency variable.
When it comes to discrete–time signals, the ones used in digital systems, the *Z–transform* is of great importance — it is a counterpart of the *Laplace transform* for continuous–time signal that transits a signal to *Z–plane*, instead of *S–plane*. The *Z–transform* of sequence $x[n]$ is defined as

$$\mathscr{Z}\{x[n]\} = X(z) = \sum_{n=0}^{\infty} x[n]z^{-n}, \tag{1.3}$$

where z is a complex continuous variable [22].

Both transforms are presented in unilateral versions, which means that they become non–zero at either $t = 0$ or $n = 0$, and this convention is commonly used in digital signal processing literature — since the system does not recognize values of a signal before the time $t = 0$.

Every control system action is based on the modification of the signal's wave shape, and therefore, can be considered and described as a digital filter. A system response can be either infinite, which happens when the filter's output is computed by using current and previous input samples and previous output sample, or finite, when its response is dependent only on input samples. This means that the finite impulse response (FIR) filter is a special case of the infinite impulse response (IIR) filter [23]. Filters, in general, are often described with the following difference equation:

$$y[n] = \sum_{i=0}^{N} b_i x[n-i] - \sum_{i=1}^{N} a_i y[n-i]. \tag{1.4}$$

As a result of performing the *Z–transform* on equation 1.4, the filter's transfer function in the discrete zero–pole domain, which is defined as a ratio of the *Z–transform* of the output signal to the *Z–transform* of the input signal, is obtained:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + ... + b_N z^{-N}}{a_0 + a_1 z^{-1} + a_2 z^{-2} + ... + a_N z^{-N}}. \tag{1.5}$$

As mentioned in the previous section, the PID controller is a closed feedback loop system; therefore its output depends linearly on both input and output values. This property allows one to treat a PID controller as an infinite impulse response filter and its transfer function may be described in continuous–frequency domain as [19]:

$$H(s) = K_p + K_i \frac{1}{s} + K_d s. \tag{1.6}$$

To improve timing and processing performance, many FPGA manufacturers in their design include modules, that are dedicated specifically to performing calculations on binary vectors in digital signal processing (DSP). As an example of a such DSP block, Xilinx DSP48E2 may be used [24]. Implementation of a digital filter in FPGA is a task DSP modules have been designed for, and are included in FPGA integrated circuits. Thus a common way of implementing digital PID regulators in FPGAs involves usage of those DSP blocks. However, PID controller coefficients used in continuous–time domain equations have to be converted to the domain in which the digital control circuit exists for the control system to do its job. A general way to calculate the digital biquad filter coefficients is transforming the controller's transfer function from *S–plane* to *Z–plane* representation, which can be achieved using e.g. bilinear transformation, and comparing the result with equation 1.5. In order to perform the bilinear transformation, one has to substitute s in equation 1.6 with $\frac{2}{T} \cdot \frac{1-z^{-1}}{1+z^{-1}}$ [21], where T is a sampling period. Having transformed equation 1.6 after the substitution, the PID controller's transfer function in a canonical form is obtained:

$$H(z) = \frac{(K_p + \frac{K_i T}{2} + \frac{2K_d}{T}) + (K_i T - K_d T) z^{-1} + (-K_p + \frac{K_i T}{2} + \frac{2K_d}{T}) z^{-2}}{1 - z^{-2}}. \tag{1.7}$$

To find PI controller's transfer function, all coefficients from derivative term have to be removed:

$$H(z) = \frac{(K_p + \frac{K_i T}{2}) + (-K_p + \frac{K_i T}{2})z^{-1}}{1 - z^{-1}}. \tag{1.8}$$

Comparing equations 1.7 and 1.8 with IIR filter transfer function (equation 1.5) allows the extraction of coefficients that make the IIR filter play the role of either a PID or PI controller. Calculated filter coefficients are presented in table 1.

Table 1: Dependency of IIR filter's coefficients on PID conroller's coefficients

IIR coefficient	PID	PI
b_0	$K_p + \frac{K_i T}{2} + \frac{2K_d}{T}$	$K_p + \frac{K_i T}{2}$
b_1	$K_i T - K_d T$	$-K_p + \frac{K_i T}{2}$
b_2	$-K_p + \frac{K_i T}{2} + \frac{2K_d}{T}$	0
a_0	1	1
a_1	0	-1
a_2	-1	0

1.3. Existing control systems

The vast variety of tasks and the increasing number of applications in which Bose–Einstein Condensate is used requires enormous sets of often incompatible hardware (often from different electronic eras). Since almost no quantum information experiment is the same, connecting, controlling and integrating various equipment becomes challenging. Moreover, quantum gas experiments usually consist of not only creating BEC but also controlling other external equipment and basing the system's feedback on the outcome of certain algorithms. The new wave of quantum experiments in microgravity or even in space [25]–[27], poses a challenging demand for miniaturization of control systems.

Issues mentioned above are not the only ones that a modern control system for quantum information experiments has to overcome. In order to obtain good control of the experiment's flow, the system should be as close to real–time as possible.

A few full-stack solutions exist and are under development, like IBM Q, Google or D–Wave. However, their main concern is to build a quantum computer and run quantum algorithms aiming to solve real–world problems, and are not to run custom physical quantum experiments.

There are few off the shelf solutions for controlling quantum experiments. As stated in [28], systems available until recently suffered either from poor timing control in exchange for ease of use, or from difficulties in programming in favour of tight timing thanks to implementation on programmable chips. Vlad Negnevitsky in [29] states that the majority of research groups performing atom or ion trapping construct their own custom–built control system out of a combination of commercially available components.

There are, however, a few solutions used in quantum information experiments, i.e. National Instruments PXI Systems and LabVIEW [30], NQontrol based on ADwin digital control platform [31], Zurich Instruments Quantum Computing Control System [32] or ARTIQ software with Sinara hardware [33]. Each is described briefly below.

1.3.1. PXI Systems and LabVIEW

National Instruments offer a system for high performance measurement and processing applications with a wide range of modular input and output (I/O) instruments (more than 600) [34]. It provides two different synchronization schemes: time–based and signal–based, and the NI–TClk delivers modular synchronization of up to a few hundred ps [35]. In the platform that incorporates the Peripheral Component Interconnect Express (PCIe) bus into the PXI system, the achievable latency is of hundreds nanoseconds order of magnitude [36]. The system is not flawless, though. The fastest modules are meant to be used only sequentially and performing concurrent tasks is not allowed. Surprisingly, the main disadvantage of the system is probably the software. Although LabVIEW is a well–known programming environment, being proprietary software makes it impossible to review even old source codes without purchasing a paid license.

1.3.2. NQontrol

The problem with proprietary software does not occur when NQontrol is used. It is a control system developed specifically to provide loop control of hardware, that is obtainable with the use of a software package written in Python. It is capable of handling up to 8 control loops running simultaneously at the sample rate of 200 kHz. The achievable control bandwidth, however, is limited to several kHz due to phase delay [31]. NQontrol is limited only to control loops and cannot be used as an off the shelf solution to control various quantum information experiments because of its inability to perform tasks other than just the control loop itself.

1.3.3. Quantum Computing Control System

Zurich Instruments offers precise, ready to use equipment for performing quantum physics experiments. The Quantum Computing Control System (QCCS) consists of four components, three of which are shown in figure 1.3 and listed below:

- HDAWG Arbitrary Waveform Generator, with bandwidth up to the 750 MHz [37];
- UHFQA Quantum Analyzer;

- PQSC Programmable Quantum System Controller — allows to synchronize the whole setup [38].

Figure 1.3: QCCS modules; image taken from [32]

QCCS's fourth component is the LabOne control software. One of its key features is that it is fully compatible with the other three devices. Furthermore, it supports many of popular programming languages used by physicists, including i.a. Python, C or MATLAB [39]. On the downside, QCCS is a fully proprietary system. Therefore scalability and modifying possibilities that could provide better system's adjustment to laboratory equipment and specific quantum experiment are limited.

1.3.4. ARTIQ and Sinara

ARTIQ

ARTIQ (Advanced Real–Time Infrastructure for Quantum physics) is a software created with the aim to provide real–time control over the quantum physics experiments. Although it was firstly developed in collaboration with National Institute of Standards and Technology (NIST), it is supported by a growing number of researchers and scientific institutions across the globe [33] thanks to being an open–source project. ARTIQ is based on Python and tries to join the expressivity of an interpreted language with timing performance of custom FPGA design. Thanks to its time–critical part being compiled (called *kernel*) and executed on FPGA (called *core device*), it is able to achieve nanosecond resolution and sub-microsecond latency [40]. A non–time–critical task may be easily interfaced with Python remote procedure call (RPC) from a host computer. ARTIQ uses hybrid architecture with a central processing unit (CPU) implemented in FPGA logic [29]. Due to its goal of fulfilling modern physics laboratories' demands for a versatile quantum information control system, ARTIQ provides a graphical user interface (GUI) and supports i.a. controlling digital inputs and outputs (DIO), RF generation hardware, digital–to–analog (DAC) and analog–to–digital (ADC) converters [41].

In order to control hardware for quantum physics experiments with a high–level programming language, low–level drivers are needed. In ARTIQ, these are created using Migen — a Python–based tool that aims to ease very large-scale integration (VLSI) system design and introduces synchronous and combinatorial orientated programming. It allows to design gateware with Python and takes advantage of various notions available in high-level programming languages, such as object–orientated programming etc. [42].

Sinara

Sinara is a product family that has been designed for ARTIQ–based quantum experiments. Thanks to being constructed under the CERN Open Hardware License (CERN OHL), scalabilty can be achieved and laboratories may adapt the hardware to their needs, as long as they publicly announce their changes and designs.

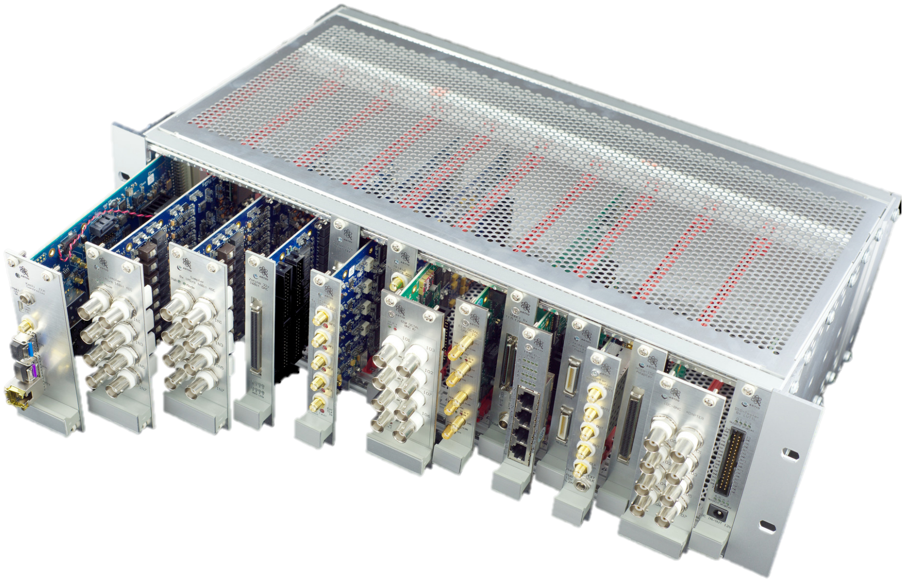


Figure 1.4: Example of Sinara family devices' setup in a crate; image taken from [43]

The family consists of a core device (at first it was KC705 board, but now usually either Kasli board with Xilinx Artix-7 FPGA [44] or Metlino board with Xilinx Kintex UltraScale KU040 FPGA is used), that controls slave devices using ARTIQ's distributed real-time input/output (DRTIO) protocol, and a number of extension modules of choice [43]. A large amount of hardware design was done at the Warsaw University of Technology at the Institute of Electronic Systems. A list of available extension modules (i.a. DAC board, ADC board or direct digital synthesizer (DDS) board) can be found on the Sinara's main wiki page: [43]. The motivation behind the Sinara project was to produce open–source devices that would allow physics research groups from across the globe to reproduce and reuse hardware needed to perform quantum information experiments. Furthermore, it is meant to reduce the amount of duplicated designs and work in different physics labs, because, as mentioned in [29], many research groups conducting quantum information experiments develop their own control system optimized for their specific use case.

1.3.5. Summary

A brief summary of existing control systems presented above is shown in table 2. Even though NI control system provides a great number of modules with wide range of application and one of the best timing resolution and latency in its class, the ARTIQ software and the Sinara hardware family are chosen by the group that supports this thesis project. The necessity to use the LabVIEW software and the PXI being the proprietary hardware were too big disadvantages to choose this system. Latency and resolution offered by the ARTIQ and the Sinara are of the similar order of magnitude as those in NI system, therefore the timing cost of using an open–source system in this case is almost negligible.

Table 2: Summary of existing control systems features

	NI PXI and LabVIEW	NQontrol	QCCS	ARTIQ and Sinara
Open–source	no	yes	no	yes
Multi–purpose	yes	no	yes	yes
Modular	yes	N/A	yes	yes
Amount of modules	over 600	1	3 + LabOne software	18 and increasing
Resolution	10–100 picoseconds for some modules	N/A	N/A	nanosecond
Latency	hundreds of nanoseconds	N/A	< 100 nanoseconds	sub- -microsecond

2. Thesis genesis, goal and technical assumptions

2.1. Genesis

This thesis project has been developed in collaboration with Optical Metrology Group (QOM) of Humboldt University of Berlin (HUB), where several pieces of research regarding quantum physics (i.a. creating rubidium Bose–Einstein Condensate) and atoms' behaviour in the state of microgravity are currently being conducted (for example: [45], [46]). Their needs to reduce weight, dimensions and to be able to control quantum experiments with one simple interface were met by ARTIQ and Sinara hardware. Even though modular nature of the Sinara project allowed to design and run quantum physics experiments of various kind with a wide range of laboratory equipment, there was no easily interfaced, ready–to–use servo to modulate connected laser sources directly (with electric current), which was imposed by laboratory equipment.

A simplified diagram of the process the QOM needed to control and connections between its particular components is shown in figure 2.1.

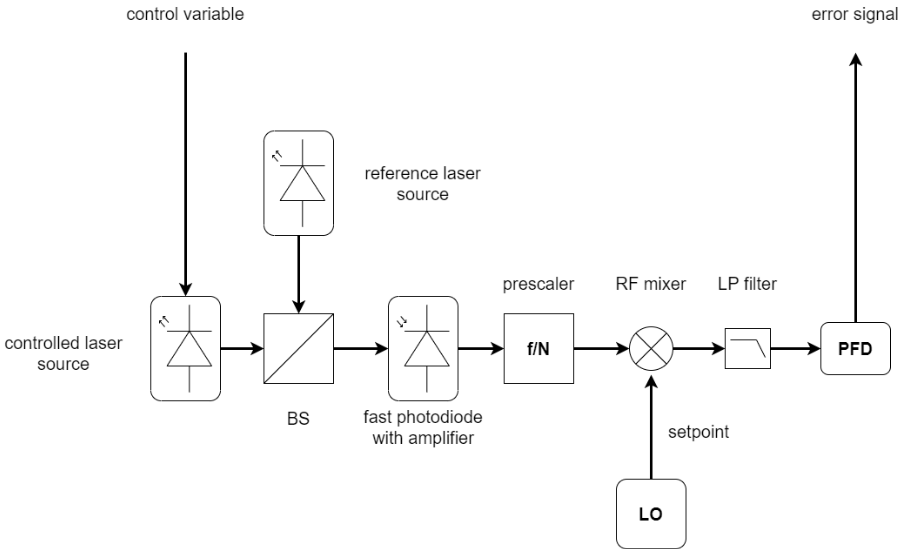


Figure 2.1: Block schematics of the controlled process

Optical frequency signal emitted by controlled laser source was mixed with optical frequency signal from a stabilized on an atomic rubidium transition reference laser source in order to obtain the difference of the two (both of them were of the hundreds of THz order of magnitude). The frequency difference (GHz order of magnitude) — a beat note — was then detected by a fast photodiode and immediately converted to a voltage signal. The resulting frequency was scaled down and mixed with the radio frequency (RF) from a local oscillator — a process setpoint. During this project the Urukul module (which is guaranteed to be able to output frequency correctly up to 400 MHz [47]) played this role. As a result, further frequency scaling down and an error signal were achieved, and after the filtration with low–pass filter signal reached phase–frequency detector. PFD's output voltage was dependent on the lock type the system operated in — when the frequency lock was in use, the signal oscillated between -1 and 1 volt, whereas when the loop was phase–coherent it took the form of a ramp signal. From the controller's point of view, the PFD's output signal can be treated as an error signal.

2.2. Goal and assumptions

The need described above defined the goal of this thesis project, which was to design a PID controller that would be useful in QOM's quantum experiments and would be accessible through a PC connected to the Sinara hardware. The phase coherence had not been required from the control loop, therefore designed regulator was needed only to be able to lock to the desired reference frequency.

The fundamental rule that had to be abided by was for the controller to be run on the Sinara core and be built with the ARTIQ software and its auxiliary tools (for example Migen). What is more, implementation of an anti–windup protection in the *integral* part of the controller in order to allow it to react instantly on the change of the input signal, even when the controller output signal remained at the rail for a longer period of time, was compulsory. Implementation of the derivative part was not required. The laser source should be modulated directly (this can be achieved by applying varying voltage to the laser driver module) and the controller output should be able to set the control variable voltage within a range of at least ±1 V. The input voltages to the controller should be supported in a range of at least ±1 V as well. Furthermore, the regulator's bandwidth, often defined as a cut–off frequency where the closed–loop gain reaches the point of 3 dB difference from the reference value [48], was required to be of at least hundreds of Hz order of magnitude. Last but not least, the controller average latency should be not worse than 1 ms (with accuracy of the half its sampling period).

3. Conception

3.1. Hardware setup

Hardware used in this thesis project to attain goals presented in section 2.2 consisted of:

- the Kasli board, which is the FPGA–based controller;
- the Sampler board, which includes 8–channel, 16–bit ADC[1];
- the Zotino board, that includes 32–channel, 16–bit DAC[2];
- PC that provides user with the controlling interface to the whole Sinara hardware via Kasli's Ethernet or USB–JTAG (Universal Serial Bus – Joint Test Access Group) connection.

The Kasli board was the only controller from the Sinara family that was available in the laboratory at the time. Therefore, its usage was necessary to meet the requirement of using the Sinara hardware and ARTIQ with its auxiliary tools. Although the whole ARTIQ environment allows to port programs and solutions to new boards, doing so was outside of this thesis scope.

Sampler is equipped with 8 integrated circuits that perform analog–to–digital conversion with 16–bit resolution [49]. Moreover, thanks to the programmable–gain instrumentation amplifiers that are mounted on the board, its input voltage ranges from ±10 mV up to ±10 V [49]. As the Sinara project page states [49], the board's sample rate reaches up to 1.5 MHz when the device is in a *fast mode*.

To meet the requirement set for the laser source to be modulated directly, the Zotino board was chosen. It is a digital–to–analog converting module that allows setting its output to the voltages in range of ±10 V [50] and provides a designer with the analogue bandwidth of 75 kHz and the update rate of 1 MSPS (Mega–Samples Per Second).

To access and communicate with the designed control system and the Kasli board, a PC was connected to the Kasli controller via either Ethernet (to interface the hardware with ARTIQ software) or USB–JTAG cable (to access the FPGA directly.)

The Sinara modules' connections to themselves and to the controlled process are shown in figure 3.1. The PFD's output voltage (the error signal) was sampled by the Sampler board and forwarded to the Kasli controller. That is where the PID algorithm was implemented and all the control over both connected modules was done. Based on the error signal's values and coefficients implemented by the user, the Kasli controller performed calculations and sent their results to the DAC on the Zotino board. As a result, the control variable that drove the laser source was set to the appropriate voltage value.

[1] See Appendix 1.1 for details on the Sampler board and its schematics.

[2] See Appendix 1.2 for details on the Zotino board and its schematics.

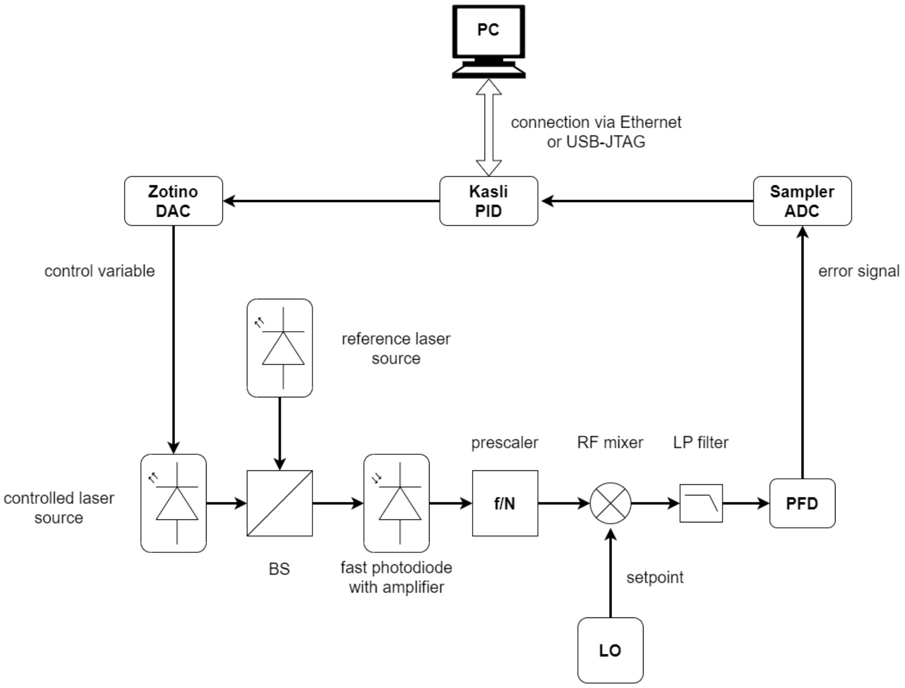


Figure 3.1: Block schematic of the system used in the thesis project and their connections

3.2. Stages of the implementation

In the controller's construction four main parts may be distinguished:

- initialization of both Sampler and Zotino;
- communicating with the Zotino board and setting desired values on its output;
- communicating with Sampler and accessing sampled values;
- actual control and integration of the other three parts.

3.3. Two approaches

Two different approaches of servo implementation were tested during the work on this thesis project: implementing PID controller with ARTIQ (software implementation with the time–critical part being compiled and run on the FPGA) and with Migen (bare–metal implementation, which means that the whole program had to be compiled and then flashed into the FPGA's memory).

3.3.1. ARTIQ implementation

Sinara hardware is created with the goal to be compatible with ARTIQ control system; therefore drivers that allow controlling the number of boards, including Zotino and Sampler, had been already well–developed and incorporated into ARTIQ at the time of the implementation. Thanks to the above and to the fact that ARTIQ is a subset of a high–level programming language — Python — employment of each stage and designing the whole script was easy to obtain, took little time and required almost no prior knowledge of the protocols used by integrated circuits mounted on module boards. There are, however, drawbacks to this solution — ARTIQ does not support concurrency, only parallelism (this means, that two different events can occur at the same time, e.g. two independent DIO can be set to take place at the same time, but nothing else can be processed until the parallel block's longest action has already ended). Therefore, implementing a closed feedback loop controller with ARTIQ, prevents a user from further usage of the control system, and thus from using all the Sinara family boards, alongside the regulator. What is more, implementation in the software does not allow to fully exploit hardware's possibilities and significantly limits the controller's available bandwidth, especially when it comes to driving more DAC channels than one at the time.

3.3.2. Bare–metal implementation

The standard approach to bare–metal implementation in an FPGA would be to choose either Verilog or VHDL as a hardware description language. Although it is possible to incorporate Verilog code into the ARTIQ control system or to implement it directly in Kasli FPGA's logic, Migen was chosen for this task to meet the requirements described in section 2 and to keep better compatibility with the whole control environment. It is supposed to be easier and more powerful than both of hardware description languages mentioned above.

Due to the fact, that this approach lacks direct access from the ARTIQ level and is implemented in FPGA logic, it overcomes the available bandwidth limit imposed by the software controlling Sinara modules, but in result, no control over the regulator's behaviour is possible with ARTIQ. Having introduced synchronous and combinatorial blocks in Migen, it's creators made it support both sequential and parallel programming, following the scheme present in modern FPGA programming paradigms. Available bandwidth, therefore, is mainly dependent on the hardware constraints and on the implementation itself. However, the bare–metal approach's biggest flaw is that it has to be flashed into the FPGA memory (either volatile or non–volatile), which means that it cannot be used in parallel to the ARTIQ software. As a result the user cannot exploit full potential of ARTIQ and Sinara to conduct experiments.

Moreover, contrary to the ARTIQ implementation, the bare–metal approach requires the designer to carefully scrutinize data sheets of integrated circuits mounted on boards, in order to get familiar with protocols they use to communicate with the master device — the software drivers are accessible only from the ARTIQ level.

4. Controller development

4.1. ARTIQ implemetation

ARTIQ implementation was performed in two ways: one with the immediate conversion of sampled values from machine units to volts, and one with only adjustment of sampled values to the binary representation required by DAC integrated circuit instead. The structure of both implementations is almost identical and the differences are limited only to use of different functions to obtain or to produce value either in volts or machine units.

Time–critical parts of the code are marked with a *@kernel* flag. In order for the compiled function to return a value that may be used outside the compiled part of the code, some kind of container (i.e. list or variable) for it has to be provided as a function's argument. In other case, the value may be returned with the Python *return* statement.

4.1.1. Initialization

Initialization of both Zotino and Sampler boards is done by invoking a function *initialize_devs*. It clears all *real–time input/output* first–in, first–out queues (FIFOs) and moves the time cursor to the current value of hardware clock with a margin. Moreover, it sets each channel's gain of programmable–gain instrumentation amplifier (PGIA) to the given value — input signal's maximum amplitude that the module is able to sample correctly depends on that — and initializes the Zotino board. Each operation has to be followed by a delay by the appropriate amount of time — it sets the core device's time cursor after the operation is already done.

4.1.2. Communication with Zotino

Communication with the Zotino board is done by calling a function named *write_-output*, which takes as arguments channel 'number' and 'value': 'number' parameter sets the number of DAC channel the data is written into, and the 'value' is the DAC's output value in volts. What is more, this function drives the *ldac* (load data) line low, in order for the DAC to know that the data sent is valid, and prevents the output value from being out of Zotino's operating range.

4.1.3. Communication with Sampler

In order to acquire sampled values of the error signal, a function *sample* has to be invoked. Since sampling is a time–critical operation, a Python list object — *values* — is passed to the function as an argument. *Values* is a list of variables in which sampled data from ADC channels is stored. In case of the implementation with conversion to volts, variables inside the list are floating–point numbers, whereas in the other case, integers. The length of the argument passed to *sample* is imposed by the functions controlling ADC implementation — to deliver high transmission speeds of sampled data, Quad SPI mode to communicate with ADC is used. Therefore, only an even number of samples may be acquired each time sampling takes place. The channel values are read sequentially.

4.1.4. Integration and control

According to the ARTIQ style of designing experiments, each part of the program is implemented as a method of the class that inherits from the *EnvExperiment* module. In case of this thesis project, the experiment controlling class is named *PID_controller*.

In order to run an ARTIQ experiment, two essential methods should be implemented by the user: *run*, which may interact with the hardware and is the heart of the control flow of the experiment, and *build*, which should be used to request for devices. What is more, there is a *prepare* method overloaded, which is executed before the start of *run* function and which asks the user to enter coefficients for the PID controller.

In addition, there are three independent methods implemented, that are responsible for proportional multiplication, integral and derivative action, and are named *proportional_multiply*, *integral_part* and *derivative_part* respectively.

4.1.4.1. Proportional part

Method responsible for the proportional part, takes *K_p* coefficient and sampled value of signal — *error* — as arguments, performs multiplication of them and returns the result.

4.1.4.2. Integral part

The integral part requires three arguments: *error*, the same value as in the proportional part, *integral_in*, which is the integrator output from the previous iteration, and coefficient *K_i*. What is more, the anti–windup protection is implemented, which prevents the integrator from adding next values, in case the sum of previous samples has exceeded the operating values of DAC. Therefore, PID controller is able to react immediately when the error signal changes and does not have to wait until the sum falls again into the range of the DAC operating voltages. The method returns two variables: the updated value of *integral_in* as the sum of all samples of *error* and *integrated_out*, which is the integrator's reaction on the current *error* value.

4.1.4.3. Derivative part

The method responsible for the PID controller's derivative action requires, similarly to the integral part, three arguments: *error*, which is the same value — corresponding to the error signal — as in the both parts described above, *last_error*, which equals to the previous value of *error*, and the coefficient — *K_d*. This method returns two variables: *last_error* and *derivative_out*, which is the method's reaction on the incoming signal.

4.1.4.4. Run method

The actual integration of all PID controller's components and control of the loop takes place inside this method. In an infinite loop, ADC channels are sampled using the method *sample*, each controller part's contribution to the output value is calculated either in parallel. Eventually, all contributing values are summed up and written to the DAC with the *write_output* method. Listing 4.1 shows the example of the infinite loop where the sampling, calculating and writing to Zotino is done.

Listing 4.1: ARTIQ main loop

```
while True:
    self.sample(values)
    delay(400*us)
    with parallel:
        prop_out = self.proportional_multiply (values[6], self.Kp)
        integral_in, integrated_out = self.integral_part (values[6], integral_in, self.Ki)
        last_error, derivative_out = self.derivative_part (values[6], last_error, self.Kd)
    result = prop_out + integrated_out + derivative_out
    self.write_output(self.DAC_channel, result)
```

4.2. Bare–metal implementation

PID controller (actually the PI controller, but, as mentioned in section 1, term PID in this paper is used, as a more general one to describe this class of controllers) had already been implemented with Migen, but its usage was limited to only acousto–optic modulation with the Urukul board used to control external devices. Since it made use of the ADC driver, it performed control with the module that had been implemented in FPGA's dedicated circuit for digital signal processing and was proven to be working correctly, a decision to base implementation described in this thesis on already existing solution has been made. Therefore what was needed to be done was the implementation of the DAC driver, careful and thorough examination of the existing controller's implementation and adjustment of the module that is managing, controlling and integrating all parts.

4.2.1. Architecture

Overview of the controller's architecture implemented with Migen is presented in figure 4.1. Each of the modules inherits from the Migen's *Module* class, which defines special attributes that are used to describe classes' logic, in particular, synchronous and combinatorial statements. The top class is *Servo*, which integrates all components necessary for the implementation of a control loop, and is in charge of the control over each device.

Migen, similarly to VHDL or Verilog, interprets signals as single–ended ones, whereas the Sinara boards require from the Kasli controller usage of the low–voltage differential signalling (LVDS) standard; therefore conversion of signals that were to be used outside of the FPGA was needed. Due to the fact, that Migen creators had already developed classes for that purpose, the conversion could be obtained with little coding, and thus placed in the same file a component had been described in. However, in order to keep the code as transparent as possible, a decision to separate the module's description and signals' conversion into independent files was made. Hence, separate '*Pads*' classes for modules that need to communicate with components outside the Kasli board were created. Also, they serve the purpose of assigning data lines used by modules to the Eurocard Extension Module (EEM) connectors' pins.

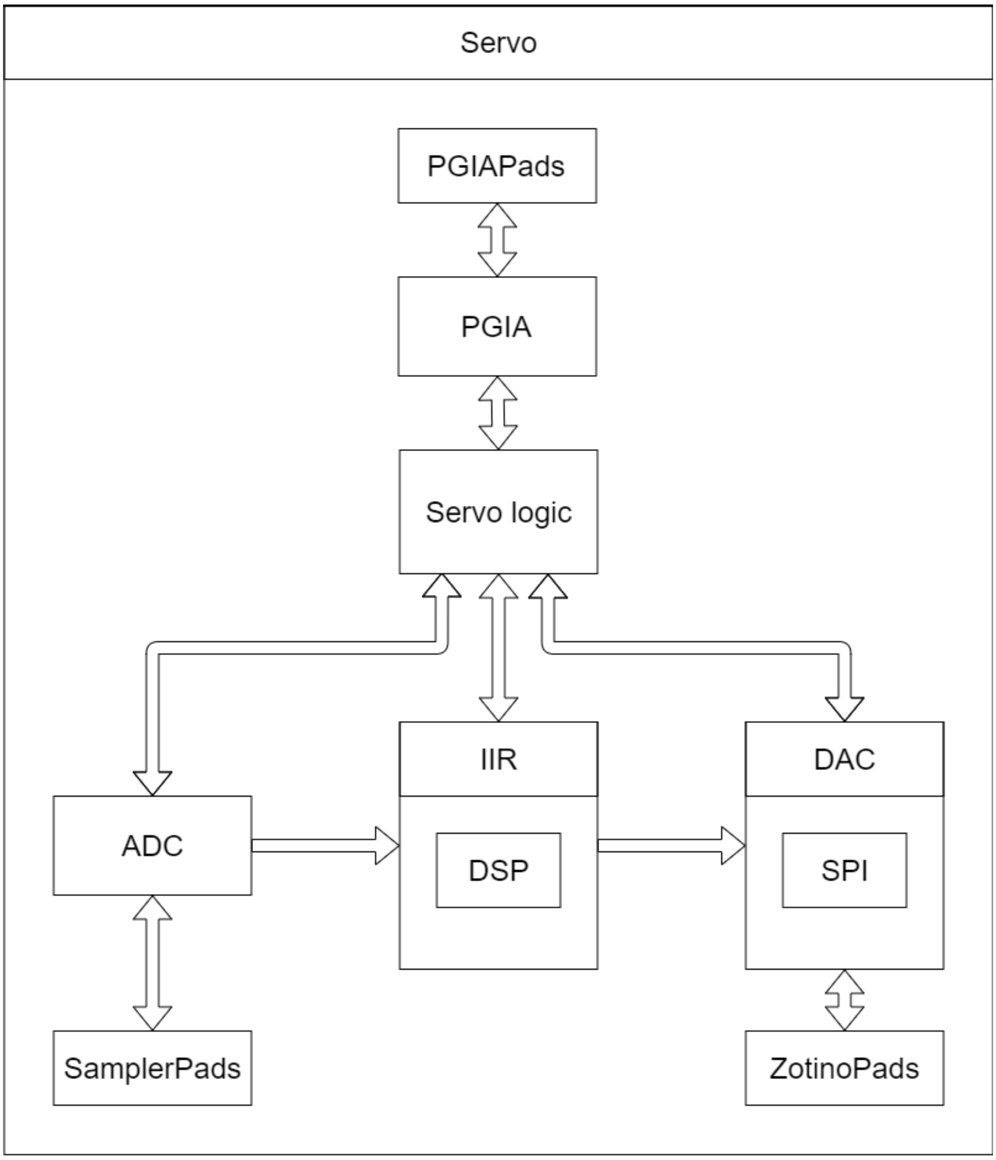


Figure 4.1: Block diagram of Servo and its submodules

The bare–metal implementation of PID controller follows the construction scheme presented in section 3.2. In this solution, contrary to the ARTIQ implementation, particular functionalities were divided into separate modules, instead of functions. This way, each module is responsible for one functionality or operation of one class of integrated circuits only:

- PGIA allows to set each of the Sampler's programmable–gain instrumentation amplifiers' gains to the desired value;

- ADC implements control over sampling process;
- DAC with SPI are used to manage data framing and communication with the Zotino board;
- in IIR module the infinite impulse response filter, that plays the role of the PI controller, is implemented;
- in each of *Pads* modules, assignment of data lines and signals' conversion is done;
- Servo module's internal logic controls other four modules.

4.2.2. Setting PGIAs' gains

Application of PGIA for each ADC channel, allows the Sampler to operate on a wide range of input voltages (from ±10 mV up to ±10 V) without loss to the measurement's accuracy. Their initialization is performed by setting amplifiers' pins corresponding to the desired gain in accordance with table 5. in [51]. On the Sinara's Sampler board PGIAs are controlled by shift registers, whose output pins are set by the controller over Serial Peripheral (SPI) lines. PGIA module's block diagram is presented in figure 4.2.

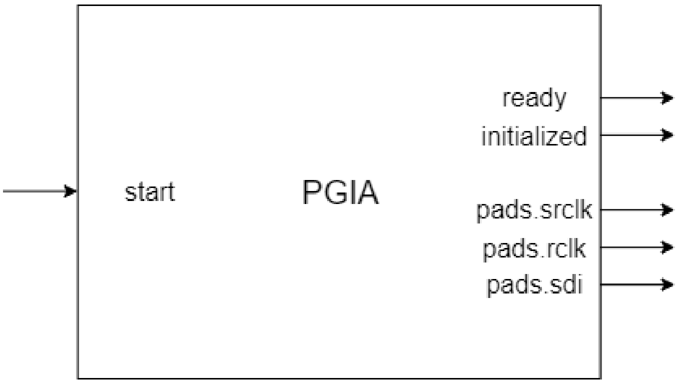


Figure 4.2: PGIA module's block diagram with input and output ports

Although shift register's timing requirements would be satisfied, if Kasli's internal clock had been used to feed data into the register, a decision was made to slow output clock by the factor of two, to ensure that data is recognized by the receiver correctly.

PGIA module communicates with the controller module inside the FPGA using three pins: *start*, *ready* and *initialized*. Lines labelled as *pads* are connected to the lines used by serial peripheral protocol and serve the communication with integrated circuits mounted on Sampler board. Control over data transmission is done using the finite state machine (FSM) that has five states: *IDLE*, *SETUP*, *HOLD*, *RCLK* and *END*. When the module is in the *IDLE* state, it lets the master controller know that it is ready to transmit data by driving the *ready* pin high. To begin a data transmission, the *start* pin has to be driven high by the master device — a sequence that sets gains is latched and the state is changed to *SETUP*. After beginning its operation, FSM alternates its state between *SETUP* and *HOLD*, as long as there are still bits left to be sent; during the *HOLD* state both *pads.srclk* (shift register clock) and *pads.rclk* (storage register clock) are driven high. If all 16 bits have been already transmitted, FSM switches to the *RCLK* state, which only purpose is to provide one storage register clock cycle more, in order for the data in the shift register to propagate to their desired positions. The FSM's last state is *END*, during which *initialized* pin is driven high.

Listing 4.2 shows how the control over the module's bit shifting with Migen is performed. When the FSM is in the *IDLE* state, none of the 16 bits has been transmitted and the *bits_left* signal keeps that value. Each time the FSM leaves the *HOLD* state, number of bits left is decreased and the content of the *sr_data* signal is shifted by one position.

Listing 4.2: Example of control over the bit shifting

```
If(fsm.ongoing("IDLE"),
    bits_left.eq(params.data_width)
),
If(fsm.before_leaving("HOLD"),
    bits_left.eq(bits_left - 1),
    sr_data.eq(Cat(sr_data[1:], 0))
),
```

Gain setting sequence is passed to the PGIA module as a constructor's argument and equals to the concatenation of eight 2–bits–wide vectors — each value sets the gain of one of the PGIAs. Those vectors can be of the following values:

- "00" — results in PGIA's gain of 1;
- "01" — results in PGIA's gain of 10;
- "10" — results in PGIA's gain of 100;
- "11" — results in PGIA's gain of 1000.

One has to bear in mind that with the increasing of the PGIA's gain, Sampler's input voltage range decreases proportionally.

4.2.3. Infinite Impulse Response filter

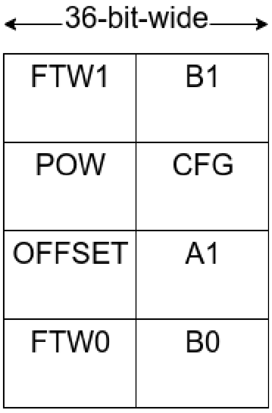
IIR module is the heart of the control system — it describes, as stated in its name, the infinite impulse response filter. The majority of work regarding the implementation of this module has been already done at the time of doing this thesis project. However, a few modifications were needed for the filter to perform its task, when applied to this project.

The module implementation's thorough examination has shown that IIR uses two random–access memory (RAM) blocks. One is used for storing filter's coefficients and the second one, for current and previous input and output values. However, the existing solution lacked the tool, that would allow to write into the module's memory IIR filter's coefficients directly from the controller module in Migen. Values that are kept inside the coefficient memory are the following:

- FTW0 and FTW1 — two parts of the frequency tuning word;
- POW — phase offset word;
- OFFSET — value added to the sampled word in order to reduce Sampler's offset error;

- CFG — information about target ADC channel that is connected to the particular DAC channel;
- B1, B0 and A1 — IIR filter's coefficients.

The coefficient's memory layout for one Servo's channel is shown in figure 4.3 — the pattern is repeated for every DAC channel used by the PI controller. Since two different values occupy the same memory address, it was essential for the access to one value not to overwrite the second one. Therefore, masks, addresses and information, whether the accessed coefficient is located in the higher or lower half of the address field, are calculated inside the Servo module, and then passed as arguments to the IIR's constructor. Coefficient's writing process begins with accessing memory's particular address and checking the value that is already stored there. Then, the word's half width is substituted with the coefficient's value and written back to the corresponding address's memory field. The scheme is repeated until there are not any coefficients left to be written into the memory. Each coefficient needs at least the duration of the three clock cycles to complete the task. When all the words are already inside the memory, IIR module signalizes it to the Servo, by driving pin *done_writing* high. The whole operation is performed in parallel to initialization of the PGIAs and the Zotino board, and is necessary to be completed before servo begins controlling operation.



FTW1	B1
POW	CFG
OFFSET	A1
FTW0	B0

Figure 4.3: Module's coefficient memory layout

The module was created with the purpose of driving Urukul integrated circuit (IC) that accepts amplitude scale factor (ASF) which is 14 bits wide. Since the DAC IC mounted on the Zotino board is a 16–bit one, what was needed to be done here, was to ensure that output data delivered to the DAC module was 2 bits wider than ASF. The filter's output value changes accordingly to the output vector's widening, since its construction allows the data to be even 25 bits wide. Widening of the vector was achieved by enabling the filter to output signed values and by changing the number of bits, by which data was shifted on its way out. The structure of the IIR filter, after changes described above being applied, is presented in figure 4.4. Clipping, to either maximum or minimum value the two's complement representation, plays the role of anti–windup protection and, what is more, prevents the output signal from overflowing.

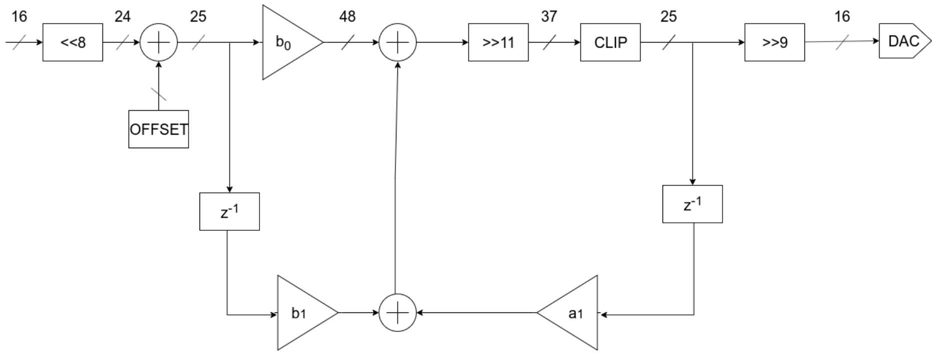


Figure 4.4: Structure of the IIR filter described inside the IIR module

The filter's output signal for each channel (*profile*) is 64 bits wide and consists of ASF, POW, FTW1 and FTW0 values— from the most significant to the least significant bits.

4.2.4. Control over digital to analog converter

The control over the digital to analog conversion takes place on two levels of abstraction. To the higher layer belongs the DAC module, which extracts information about the target voltage, frames it with additional data and sends it to the lower–level controller — SPI module. It, on the other hand, provides the PID regulator with transmission over data lines adjusted to the Zotino's IC requirements.

4.2.4.1. SPI module

SPI module is responsible for low–level communication with an integrated circuit that performs digital to analog conversion (IC used on Zotino board is AD5372BCPZ [52] from Analog Devices). Although the protocol used by the IC is described as a Serial Peripheral Interface (hence the module's name SPI) protocol, there are a few additional signal lines used. However, it is the controller of the higher level — DAC module — that makes use of those extra lines and from the SPI module's perspective, it may be considered as a standard SPI protocol. SPI module block schematic is shown in figure 4.5.

The module's input ports consist of *spi_start* and *dataSPI* bus; width of the latter is parameterized and equal to the *data_width* field of *params* tuple, that is passed as an argument to the module's constructor.

The module's output ports fall into two categories: the ones that are used to communicate with other components inside the FPGA, and the ones used to drive data lines connected to external devices. To the former category belongs *spi_ready*, which indicates that the module is ready to accept new data and send them to the IC, whereas to the latter belong ports labelled as *pads*. During ongoing data transmission, serial clock signal — which is required to be slower than 50 MHz [52] — is sent to slave device over *pads.sclk* line. When data transmission is over, *pads.sclk* line is kept low. Data received over *dataSPI*

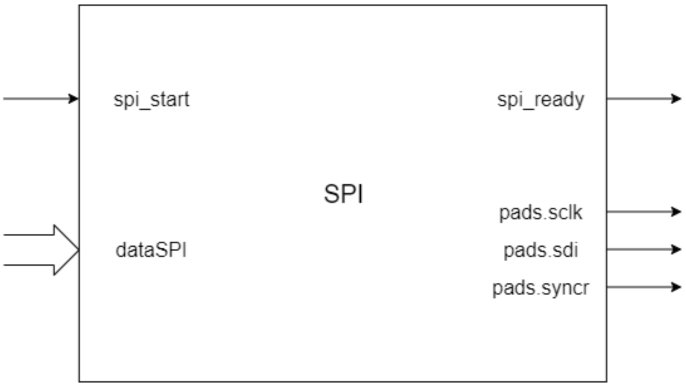


Figure 4.5: SPI component's block diagram with input and output ports

line is latched and stored as a vector, and when start event is issued, vector's content is serialized and sent over *pads.sdi* line, most significant bit first. *Pads.syncr* line is used to frame the data sequence — when driven low, *pads.sdi* and values are valid and should be checked by the DAC on *pads.sclk's* every falling edge. After every 24th bit sent, *pads.syncr* has to be driven high for at least 20 ns, for the transmission to be accepted. Otherwise, data would be recognized as corrupted and transmission would not be accomplished.

The SPI's control over the transmission is implemented using a finite state machine that has three states: *IDLE*, *SETUP* and *HOLD*.

When in the *IDLE* state, the SPI module drives both *pads.syncr* and *spi_ready* pins constantly high. Whenever *spi_start* pin is issued, data from *dataSPI* is latched into the shift register and the FSM's state is changed to *SETUP*. Moreover, the counter responsible for the *clock_enable* signal is launched. Its highest value is parameterized and equal to the *params.clk_width* field. Each FSM state is entered on the rising edge of the FPGA's clock, but only if *clock_enable* condition is satisfied. It allows manipulating state's duration and the frequency of the clock sent to the IC. In addition, during the *IDLE* state, *bits* signal, which is another counter that provides with information of how many bits are still to be sent, keeps the value of *data_width-1*. During the *SETUP* state, *pads.syncr* is driven low and the module enables the output clock. When the *clock_enable* signal is active, FSM changes state to *HOLD*. Each time FSM enters the *HOLD* state, the number of bits inside the shift register is checked, and if there are no bits left to be transmitted, the module switches back to the *IDLE* state. Otherwise, the SPI module shifts data inside the shift register by one position, decreases the *bits* counter by one and then, having launched *clock_enable* counter, it switches back to *SETUP* state. Disabling the *pads.sclk* output takes place in that state as well. This means that data on *pads.sdi* line is currently valid, because IC collects data on each falling edge of the clock line. Because the DAC requires at least 10 ns of a delay between the 24th bit's falling edge and the *pads.syncr* rising edge for data not to be corrupted, going back to *IDLE* state is possible only when *clock_enable* is asserted.

When using the SPI module, one has to bear in mind that *pads.syncr* line has to be driven high for at least 20 ns. Therefore, the *spi_start* cannot be constantly tied high, because the FSM would leave the *IDLE* state too soon and the IC would not recognize data correctly. The control over that timing is not done inside this module — it is the DAC module that takes care of it.

Listing 4.3 shows how to define the FSM in Migen and how the signals in the SPI module's *IDLE* state are driven. Each finite state machine has to be assigned as one of the object's submodules. In the first combinatorial statement the *clock_enable* signal is added to the FSM, so this condition is checked in each state automatically.

Listing 4.3: Example of definition and one state of FSM

```
self.submodules.fsm = fsm = CEInserter()(FSM("IDLE"))
self.comb += fsm.ce.eq(clk_cnt_done)
self.comb += pads.sdi.eq(sr_data[-1])
fsm.act("IDLE",
    self.spi_ready.eq(1),
    pads.syncr.eq(1),
    If(self.spi_start,
        NextState("SETUP"),
        data_load.eq(1),
        cnt_load.eq(1)
    )
)
```

4.2.4.2. DAC module

DAC module performs three, equally significant, tasks: it derives information about the voltage to be set from *profile* signal, wraps it in additional information regarding target channels, and controls the SPI component. What is more, it coverts derived information about the output voltage from the two's complementary to the binary representation. If it was not for this action, voltages on the Zotino's output pins would not be set correctly. DAC's block diagram with input and output ports is shown in the figure 4.6.

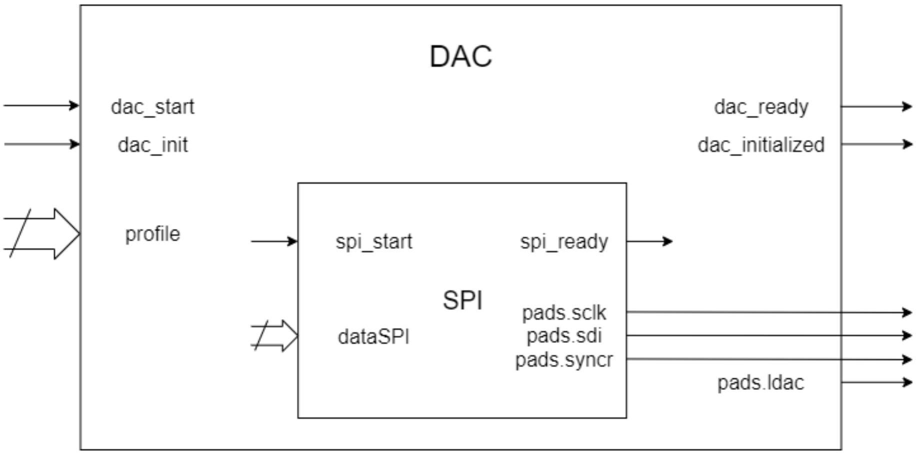


Figure 4.6: Block diagram of DAC module

Dac_start and *dac_init* ports are used by the master controller — in this case Servo module — either to begin the DAC's operation and data transmission to the Zotino board or to perform initialization of the DAC's IC. As soon as the controller drives the *dac_init* signal high and the device has not yet been initialized, the SPI module sends a data sequence that calibrates the DAC's OFS0 register and sets *dac_initialized* high to let the controller know that initialization is not needed any more. DAC begins its operation when *dac_start* pin is driven high, and after transmission to all used channels has ended, it drives *dac_ready* pin high. DAC component adds to the SPI output ports *pads.ldac* signal — it has to be driven high only during initialization and is permanently tied low afterwards.

Similarly to the SPI module, DAC driver is implemented using FSM. What is more, it only has three states as well: *IDLE*, *INIT* and *DATA*. When in *IDLE* state, *dac_ready* pin is constantly asserted and module's next state depends on *dac_start* and *dac_init* pins. To ensure that SPI module's *pads.syncr* pin is driven high long enough for the DAC to recognize framing, timing control is implemented using a counter. It counts down the clock cycles needed for a valid transmission — the number of bits to send times *params.clk_width* times 2, plus 4 clock cycles. The additional four cycles are required for the SPI to remain in IDLE state and drive *pads.syncr* line. When the *dac_start* is set, the counter is launched and the FSM's sate is changed to *DATA*. During ongoing *IDLE* state, data from the *profile* bus is latched in a shift register, information about target channels is added to it, and the word counter, that indicates how many words remain inside the shift register, is set to the number of used channels. When in *DATA*, every time the number of the clock cycles needed for one SPI transmission has elapsed, the number of words in the shift register is checked. If there are still words to be sent, the counter is launched and the words inside the shift register are shifted by the width of a single word. If all the data has been sent and the shift register is empty, FSM switches back to the *IDLE* state. Every time cycle counter is launched, the DAC module sets *spi_start* pin on the next clock's rising edge for the duration of one clock cycle.

Shifting data received from the IIR module, however, is needed. The ADC IC delivers data in two's complement 16–bit–wide representation. Even though the DSP block used by FPGA handles two's complementary operations smoothly, DAC IC requires data to be in the binary representation. Thus, adding half of the range achievable on signal's width was done — thanks to overflow occurring when adding to the signed values, a shifted values are obtained.

4.2.5. Pin assignment and signals conversion

Each *pads* module's constructor takes two values as arguments: *eem* and *platform*. The former one indicates the name of the board used by the module and the number of EEM connector the board is connected to; the latter one marks the target platform on which the program is to be executed. Passing them as arguments allows for the design to be portable across different hardware setups and controller boards. Each of the platforms supported by Migen has a *request* method, that takes the name of the requested I/O tuple as an argument and allows to gain access to its peripheral signals and pins used by the FPGA. To illustrate how the board's connectors and pins assignment are structured in Migen, Zotino's connector tuple is shown in listing 4.4

Listing 4.4: Example of Zotino's I/O tuple structure

```
("zotino{}_spi_p".format(eem), 0,
    Subsignal("clk", Pins(_eem_pin(eem, 0, "p"))),
    Subsignal("mosi", Pins(_eem_pin(eem, 1, "p"))),
    Subsignal("miso", Pins(_eem_pin(eem, 2, "p"))),
    Subsignal("cs_n", Pins(_eem_pin(eem, 3, "p"))),
    IOStandard(iostandard),
),
("zotino{}_spi_n".format(eem), 0,
    Subsignal("clk", Pins(_eem_pin(eem, 0, "n"))),
    Subsignal("mosi", Pins(_eem_pin(eem, 1, "n"))),
    Subsignal("miso", Pins(_eem_pin(eem, 2, "n"))),
    Subsignal("cs_n", Pins(_eem_pin(eem, 3, "n"))),
    IOStandard(iostandard),
),
```

I/O tuple consists of the peripheral's name, identity number, set of Subsignals and IOStandard. Identity number allows to distinguish different instances of the same peripheral — for example, different light–emitting diodes (LEDs). Thanks to the Subsignal helper class, resources that use FPGA's pins may be arranged in an easy to read way. What is more Subsignal's constructor structure is identical to I/O tuple, with the identity number omitted. It allows to gain access to the FPGA's specific pin to by using signals name. Finally, IOStandard indicates the logic standard used by FPGA to drive those pins. A method of requesting access to peripheral's pins is presented in listing 4.5.

Listing 4.5: Example of the *request* method usage

```
spip = platform.request("{}_spi_p".format(eem))
spin = platform.request("{}_spi_n".format(eem))
ldacn = platform.request("{}_ldac_n".format(eem))
busy = platform.request("{}_busy".format(eem))
clrn = platform.request("{}_clr_n".format(eem))
```

As shown in listing 4.5, there are two I/O tuple instances of the close resemblance. This is because each of the tuples corresponds to the polarity of the FPGA pins they drive. Thus, there are four SPI lines connected to negative–polarity pins and four lines connected to positive–polarity pins.

After pins are requested and accessed, a conversion from single–ended signals to differential signalling is done. As mentioned in section 4.2.1, Migen creators developed classes that aim to perform that task easily: *DifferentialInput*, *DifferentialOutput* and *DDROutput*. Each of their constructor's argument list consists of single–ended signal and two pins that are to be driven by the signal. Access to these pins by their name has to be gained prior to the conversion with the *request* method.

Exemplary usage of conversion classes is shown in listing 4.6.

Listing 4.6: Conversion of single–ended signals to differential signalling

```
self.specials += [
        DifferentialOutput(self.ldac, ldacn.p, ldacn.n),
        DifferentialOutput(self.sdi, spip.mosi, spin.mosi),
        DifferentialOutput(self.sclk, spip.clk,  spin.clk),
        DifferentialOutput(self.syncr, spip.cs_n, spin.cs_n),
        DifferentialOutput(self.clr,  clrn.p,  clrn.n),

        DifferentialInput(busy.p, busy.n, self.busy),
]
```

4.2.6. Servo module

The module that integrates all of the parts described above and is in charge of launching its every submodule is the Servo module. It has two pins to communicate with a potential controller of higher level: *start*, which, when driven high, begins Servo single iteration — reading the ADC channels' values, processing them and then setting DAC outputs accordingly, and *done*, which signals that Servo's iteration has been completed.

The overall control of the data flow and four submodules, that communicate with external Sinara boards, and connection of ADC, IIR and DAC channels to each other is done by Servo's logic. What was needed to be taken into consideration when having planned assignments, was the fact, that the order of the channels had been reversed on the Sampler board. To restrain latency introduced by the Servo's components, sampling of the ADC channels begins a precise amount of time before the processing and sending data to DAC finishes. It allows the ADC to convey samples just in time the IIR and DAC modules are ready for the next transmission.

The data flow and behaviour of components are defined inside Servo's synchronous and combinatorial statements: events that trigger components are combinatorial, whereas asserting components' readiness and the current state is done synchronously. Therefore, components' states are checked every time a rising edge of the Kasli's internal clock happens and action that results from those states occurs on the next rising slope. To illustrate the difference between combinatorial and synchronous assignments, a simple assignment of two variables was done — their timing dependencies are presented in figure 4.7 (both signals' values are assigned when the condition signal is asserted).

The components' states are stored inside the three–bits–wide Signal named *active* — each bit corresponds to the current status of each device: third bit to DAC module, second to the IIR module and the first bit to the ADC module. Whenever a component finishes its task, Servo changes *active* bit in accordance with the action performed, provided the

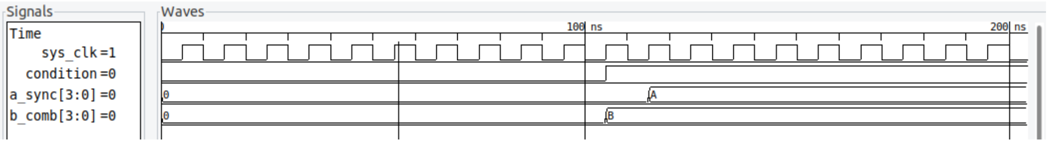


Figure 4.7: Example of synchronous and combinatorial assignments

Servo iteration has been issued. Each device is dependent on particular conditions that have to be satisfied in order for it to operate:

- ADC begins sampling and data transfer to the Kasli, only if both Sampler's PGIAs and DAC are already initialized, the Servo iteration has been issued, filter coefficients have been already written to the IIR's memory and the amount of time for the IIR and DAC modules to finish their current operation is sufficient;
- IIR module begins operation only if ADC is ready (it either has already sampled signals or it did not start any action at all) and when the ADC start event has been issued;
- DAC module starts transmission to the Zotino board only if IIR start event has been issued and the IIR module either is currently in shifting phase or has already finished operation.

Because initialization of DAC and PGIAs as well as writing the filter coefficients into the IIR module's memory is mandatory before the control loop starts, it is the first thing the Servo module does when launched.

In addition to the logic described above, a function *coeff_to_mu*, that allow to convert PID coefficients from the form used in equation 1.1 to the digital IIR filter coefficients from equation 1.5, was implemented. The number of channels used by the PID controller determines how many times the *coeff_to_mu* function is invoked. After invoking the function, its results are passed as arguments to the IIR module's constructor, where it is written into the coefficient memory.

To ensure that all values and signals are settled after the FPGA's initialization, the beginning of the Servo's operation has to be delayed. Therefore, a counter that counts down the clock cycles from the specified in the Servo module amount. When it reaches zero, the *start_done* line is asserted which is necessary for any other action, such as initializing PGIAs or writing coefficients to the IIR module's memory.

5. Tests and measurements

After the design process was finished, tests and measurements were conducted to find out whether the implemented PID controller works properly. To ensure that each of the controller's terms — proportional, integral (and in case of ARTIQ implementation, derivative as well) — functions as planned, they were tested separately, at first. If their assessment result was positive, they were combined to form the whole PI controller, which was then examined.

Since PID controller tuning is outside of this thesis scope, each of the coefficients, that were used during tests of individual controller's terms, was chosen arbitrarily — just to illustrate that the regulator works as designed and demonstrate its features. However, to conduct tests of the controller as a whole, a method proposed by Karl Johan Åström in [53] to calculate the controller's coefficients was followed. It is a modified version of an open–loop Ziegler-–Nichols tuning method [54] with a few adjustments for use with digital implementations, in which delay introduced by sampling period has to be taken into account. Even though the Ziegler–Nichols method provides only moderately good tuning [53], it allows to calculate regulator's coefficients easily by taking only a few measurements. The PID coefficients used during the controller's testing can be found in table 3.

Table 3: PID controller's coefficients used during device testing

	ARTIQ		Bare–metal	Test type
	With conversion	Without conversion		
K_P	2	2	2	P–TERM
K_I	0	0	0	
K_D	0	0	—	
K_P	0	0	0	I–TERM
K_I	2	2	2000	
K_D	0	0	—	
K_P	0	0	0	D–TERM
K_I	0	0	0	
K_D	5	5	—	
K_P	0.0048	0.0072	0.1007	PI
K_I	1.8415	4.3617	839.19	
K_D	—	—	—	

5.1. Testing procedure

In order to conduct each term's unit test, the devices used during this thesis project — the Kasli controller, the Zotino and the Sampler boards — and measuring equipment were set up as shown in figures 5.1 and 5.2. It is essential to emphasize that each test and measurement was performed with the control loop being open and the oscilloscope being connected as a plant. The stimulus signal from the function generator played the role of the process setpoint. However, since the loop was open and the error could not decrease due to the controller's action, the error signal was equal to the setpoint. Therefore, these two terms are used interchangeably from now on.

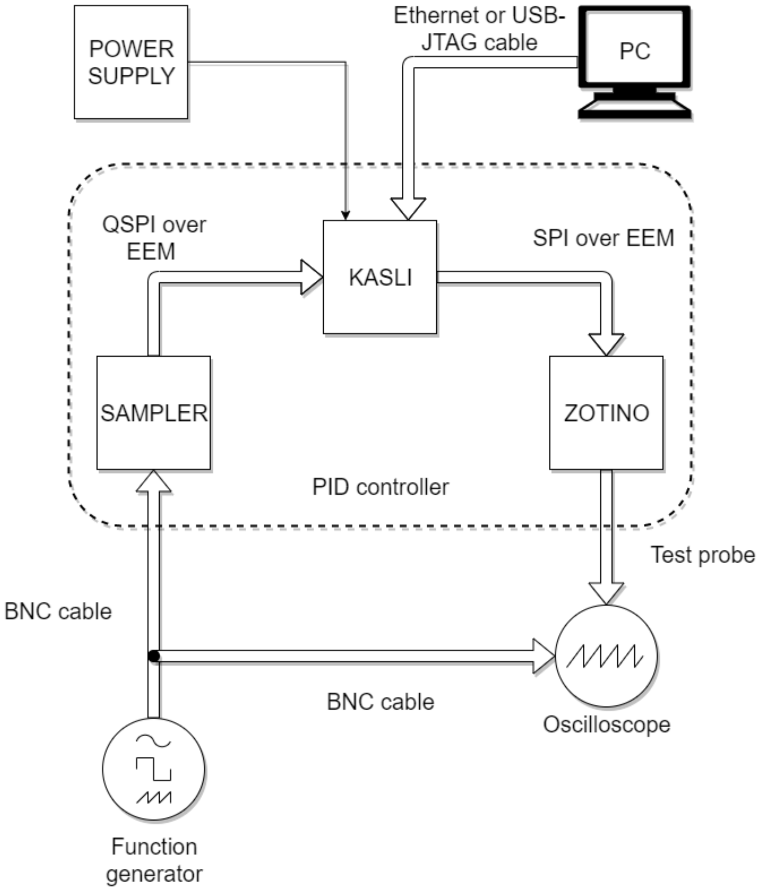


Figure 5.1: Block schematic of the test setup

To ensure that the setpoint was within the regulator's bandwidth and to prevent the controller from saturating, sample rate for each controller's implementation was estimated. As a result, the signal fed to the Sampler board was set to the frequency of 100 Hz and the amplitude (peak–to–peak) of 2 volts. The function generator, connected to the tested hardware with a BNC cable, was set to produce sinusoidal wave (tests of the proportional term), square wave (tests of the integral and the derivative actions) and a step change (to calculate controller's coefficients using the method suggested by Karl Johan Åström).

The oscilloscope and the test probe were used to observe the regulator's response to the stimulus signal and to take measurements.

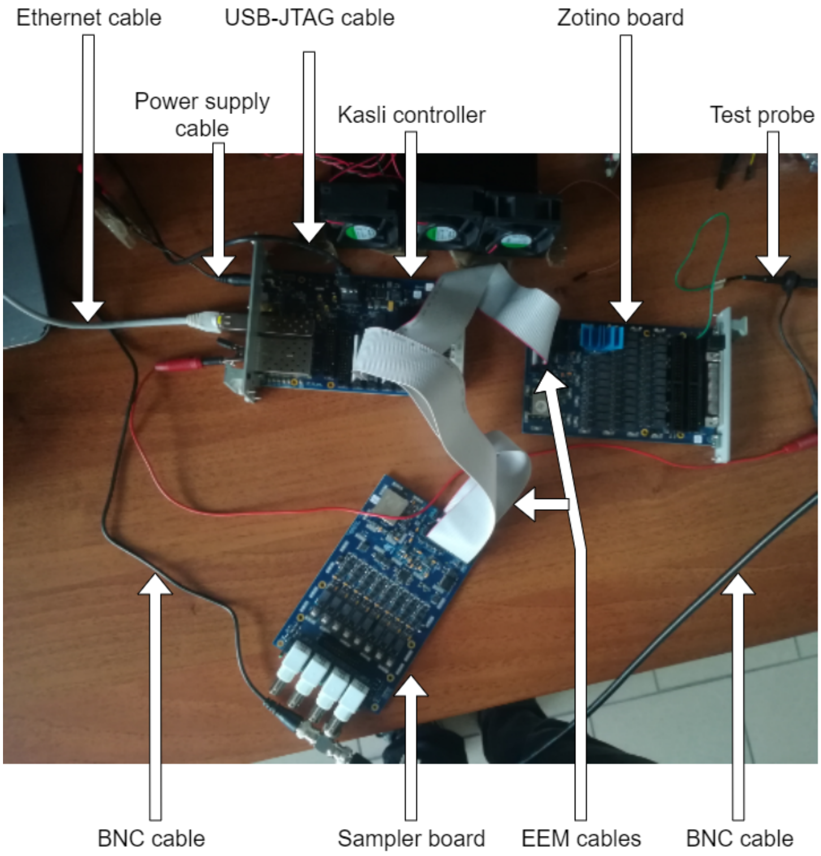


Figure 5.2: Photograph of the test setup

Having established that all the terms worked as designed, the examination proceeded to the phase of testing the PI controller as a whole. The test setup was identical to the one presented in figures 5.1 and 5.2. However, during this phase, the regulator responses to setpoint's rapid changes were studied, instead of its waveform's shape.

To examine the controller's response to a sudden change of the setpoint, a voltage step change was applied to the Sampler's input. In the similar way the controller's response to the voltage spike on its input was studied. Because controller tuning has a great impact on the system's response, to avoid its influence the latency measurements were performed with the integral term set to 0. Moreover, control system's bandwidth heavily depends on the controller coefficients as well. Thus measuring the implemented regulator's frequency response with precise instruments such as a vector network analyzer would not be a sensible thing to do. Instead, the controller's bandwidth was estimated from its multiple responses to the step change.

In each of the figures presented below, the first trace shows the setpoint waveform, whereas the second trace presents the control variable at the output of the Zotino board.

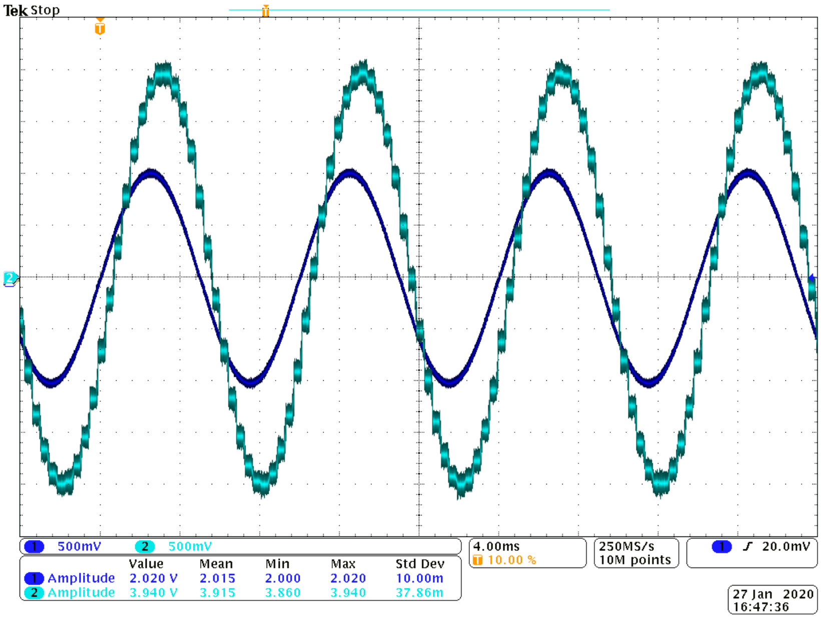
5.2. Tests of the ARTIQ–based implementations

ARTIQ–based implementations sample rates were estimated using the ARTIQ core's *real–time input/output* (RTIO) counter which counts the FPGA's clock cycles. It allowed retrieving absolute timestamps of the events associated with the beginning and the end of the control cycle and computing their difference, which was the sample rate. Having applied the sampling theorem [55], the system's bandwidth was obtained and estimated.

5.2.1. Implementation with immediate conversion to volts

Proportional term

As shown in figure 5.3, the control variable's amplitude is almost twice as big as the error signal's. This indicates that the proportional term works properly, as the ratio of output and input signals stays in accordance with the gain coefficient specified in table 3. What is worth noticing, is the fact that even though the wave's frequency is relatively low — only 100 Hz — the phase shift introduced by the controller to the controlled signal is not negligible and visible with the bare eye..



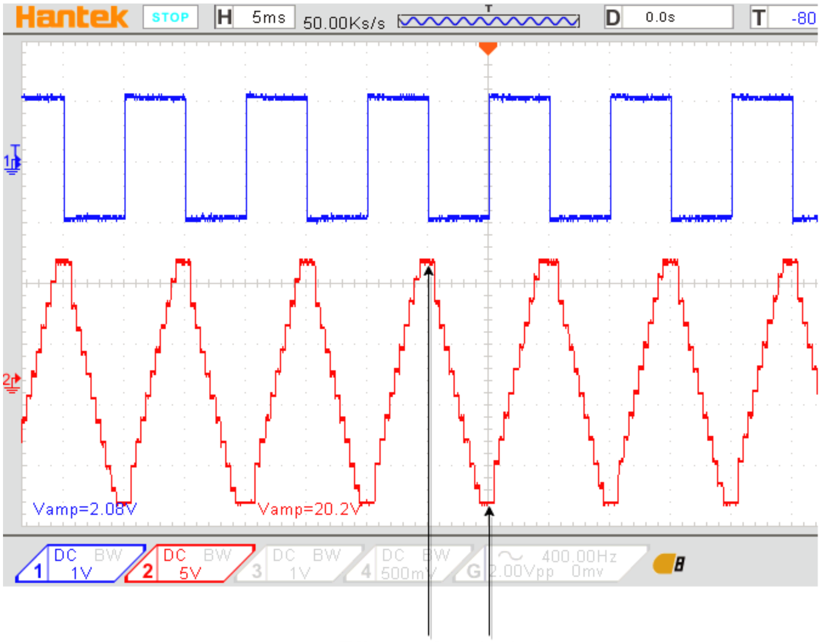
	Value	Mean	Min	Max	Std Dev
1 Amplitude	2.020 V	2.015	2.000	2.020	10.00m
2 Amplitude	3.940 V	3.915	3.860	3.940	37.86m

Figure 5.3: Setpoint and control variable — test of the proportional term

Integral term

Since integral part of the PID regulator eliminates steady–state error by adding error signal's values from the past to the current sample, it may cause the CV to drift with time due to the additional offset introduced by either the Sampler or the Kasli controller. Therefore, to see the full CV's waveform on the oscilloscope screen, the coupling was set to AC (alternating current) when the regulator's integral term was term was undergoing testing.

In figure 5.4 the CV waveform is shown. It takes the form of a ramp signal, which is the expected result of adding previous sampled values to each other. The flat parts of the output waveform are the effect of anti–windup protection. If it had not been implemented, the controller would not stop integrating and would try to set its output voltage to the value that is beyond the hardware limits. Only two examples of this effect are indicated with arrows in figure 5.4 to keep the its clarity.

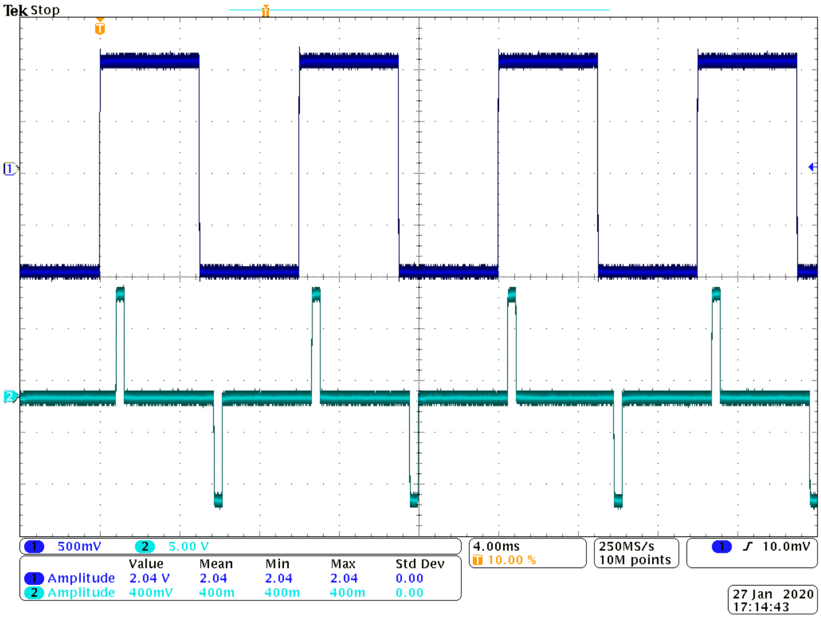


Effects of anti-windup protection

Figure 5.4: Setpoint and control variable — test of the integral term

Derivative term

The controller's derivative action is shown in figure 5.5. Similarly to its mathematical model, it is a measure of how fast the function values change in a response to the function's arguments. Thus it forms a spike every time the setpoint is rapidly changed and stays at the steady–state position in case no SP's change occurred within the last sampling period. Similarly to figure 5.3, the influence of the sampling period can be observed — the voltage spikes lag a certain amount of time behind the setpoint's change. However, it does not prevent the controller's derivative term from setting control variable properly.



1 500mV 2 5.00 V

	Value	Mean	Min	Max	Std Dev
1 Amplitude	2.04 V	2.04	2.04	2.04	0.00
2 Amplitude	400mV	400m	400m	400m	0.00

4.00ms
T 10.00 %

250MS/s
10M points

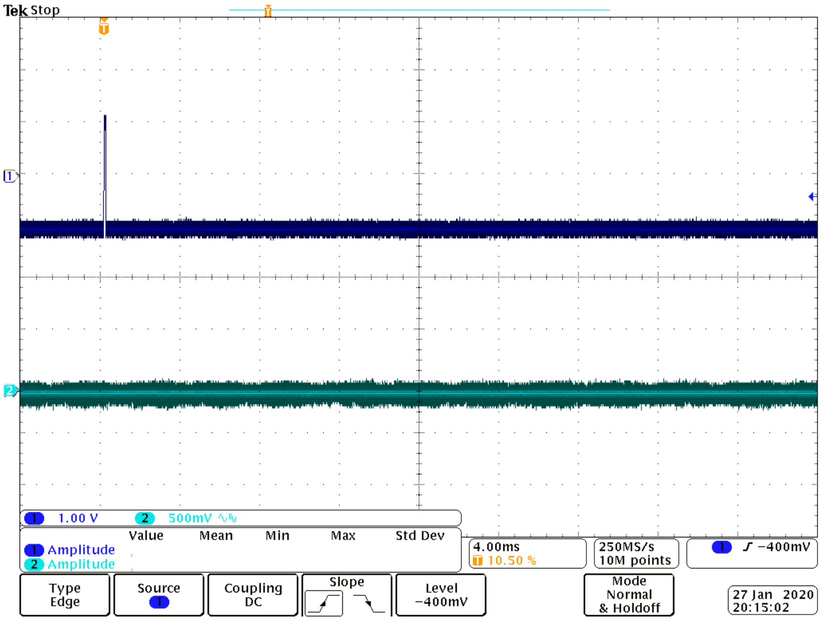
1 ƒ 10.0mV

27 Jan 2020
17:14:43

Figure 5.5: Setpoint and control variable — test of the derivative term

PI controller

Figures 5.6, 5.7 and 5.8 present the PI controller responses to the impulse (100 μs duration) and the step stimulation and measurements of the average latency, respectively. What the testing revealed was that the regulator did not recognized the impulse and did not respond to it at all. It is most likely the result of the controller's bandwidth being too small.



1 1.00 V 2 500mV

	Value	Mean	Min	Max	Std Dev
1 Amplitude					
2 Amplitude					

4.00ms
T 10.50 %

250MS/s
10M points

1 ƒ −400mV

Type Edge | Source 1 | Coupling DC | Slope | Level −400mV | Mode Normal & Holdoff

27 Jan 2020
20:15:02

Figure 5.6: The controller impulse response

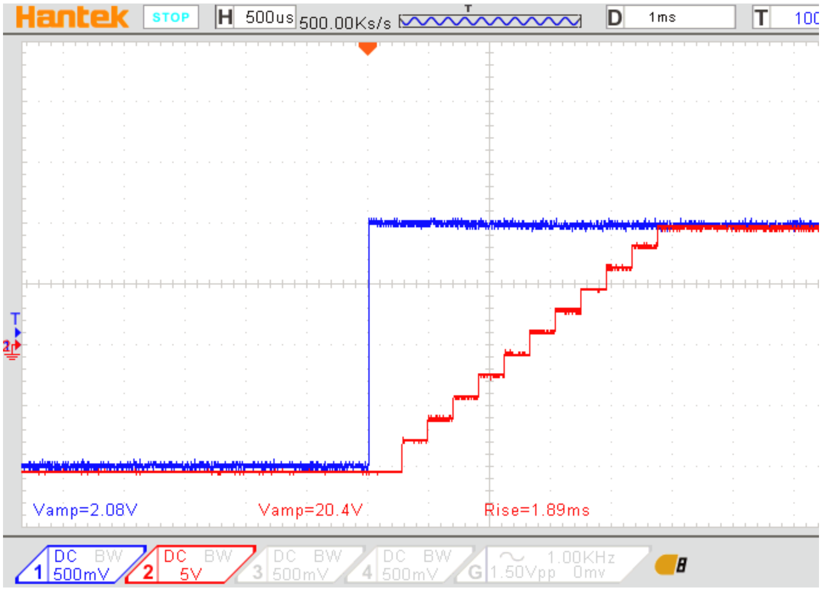


Figure 5.7: The controller step response

However, as shown in figure 5.7, a step change caused the PI controller's reaction. Having detected the voltage step, the regulator started integration. After it reached the point of maximum voltage output, further integration was stopped thanks to the anti–windup protection.

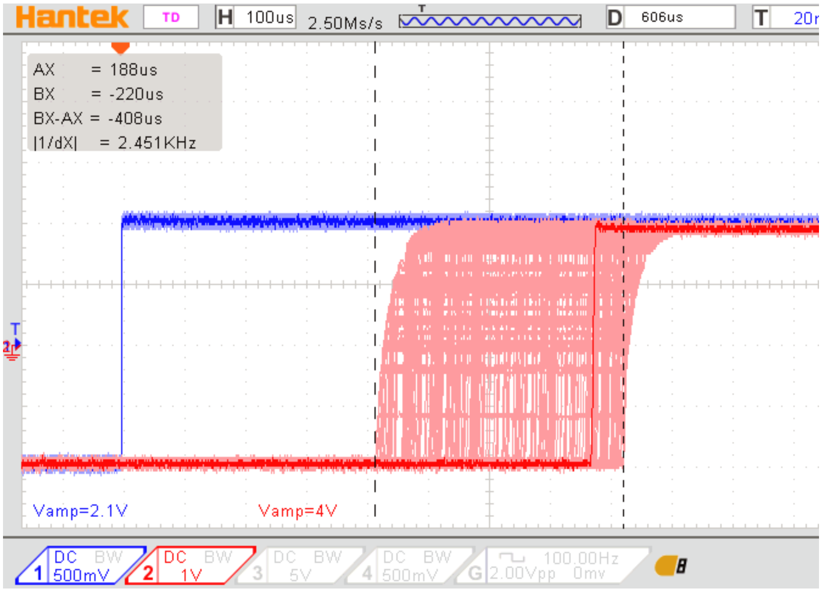

```
AX     = 188us
BX     = -220us
BX-AX = -408us
|1/dX|   = 2.451KHz
```

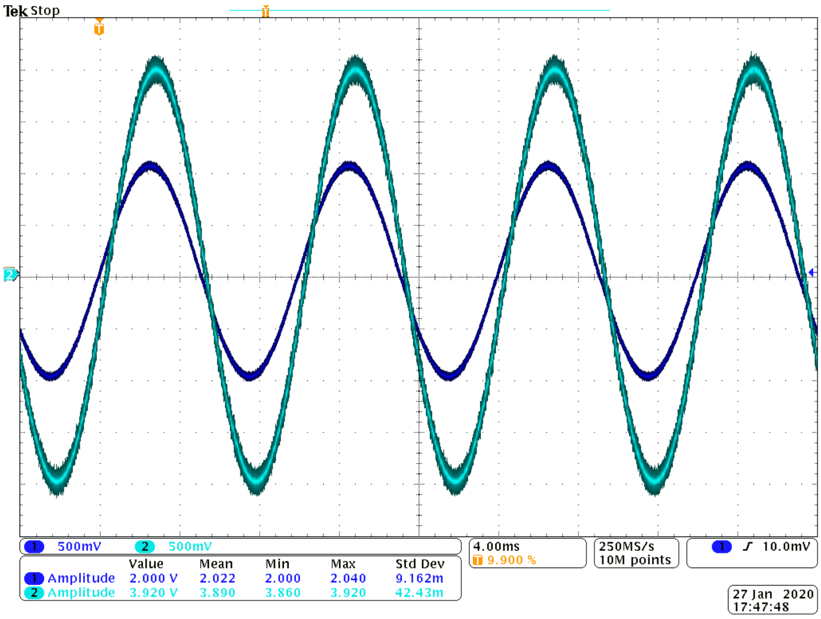
Figure 5.8: The controller average latency

The controller overall latency consists of not only the delay introduced by sampling, processing data and communication with the DAC, but also of the delay caused by the different moments of sampling the error signal. To include the latter's influence, the latency was measured from the moment when the setpoint changed to the average point of time when the regulator responded. What is more, the deviation of the signal from this point of time is equal to the controller's sampling period. Having obtained this parameter allowed calculating the regulator's bandwidth. As a result, the average latency of approximately 600 μs was measured and the bandwidth of around 1.2 kHz was calculated.

5.2.2. Implementation without conversion to volts

Giving up the conversion from machine units to volts allowed to spare some time when it comes to the controllers sampling period. However, the results of each term's measurements are similar to those obtained during testing the implementation with the immediate conversion to volts, and therefore are not extensively discussed. As presented in figures 5.9–5.11 every part of the controller performs its tasks properly. What is worth mentioning is that thanks to the improved bandwidth, the controller's dead time — the amount of time that has to elapse before the controller responds to the setpoint's change — decreased. Moreover, phase shift decreased as well. It is easily seen in figure 5.9, though.

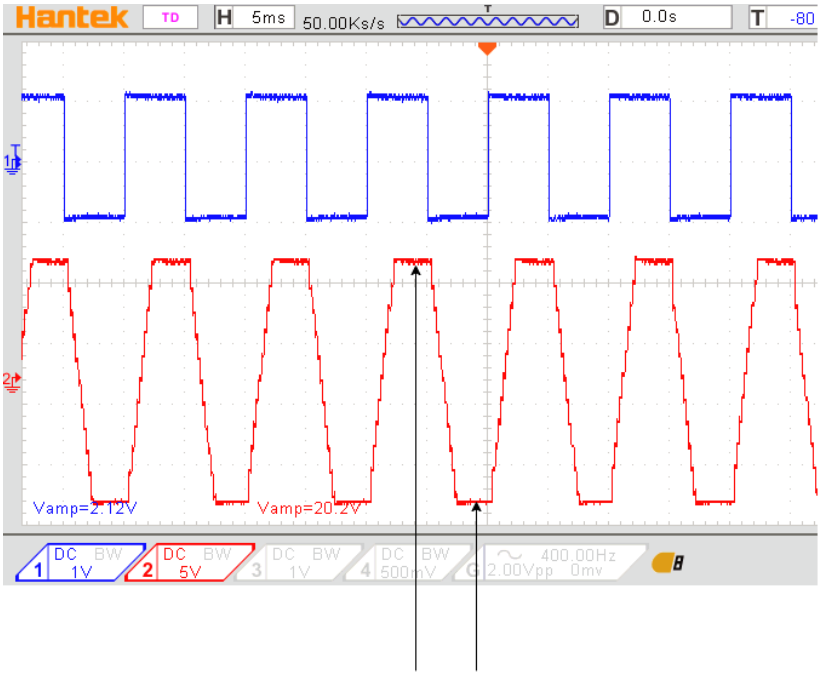
The measurements of the controller's step response resulted in calculating the controller's bandwidth: approximately 2.4 kHz, and its average latency: around 340 μs. Increased bandwidth and decreased latency can be observed in figure 5.11 as well — the spikes caused by the derivative term are thinner and closer to the setpoint changes.



	Value	Mean	Min	Max	Std Dev
1 Amplitude	2.000 V	2.022	2.000	2.040	9.162m
2 Amplitude	3.920 V	3.890	3.860	3.920	42.43m

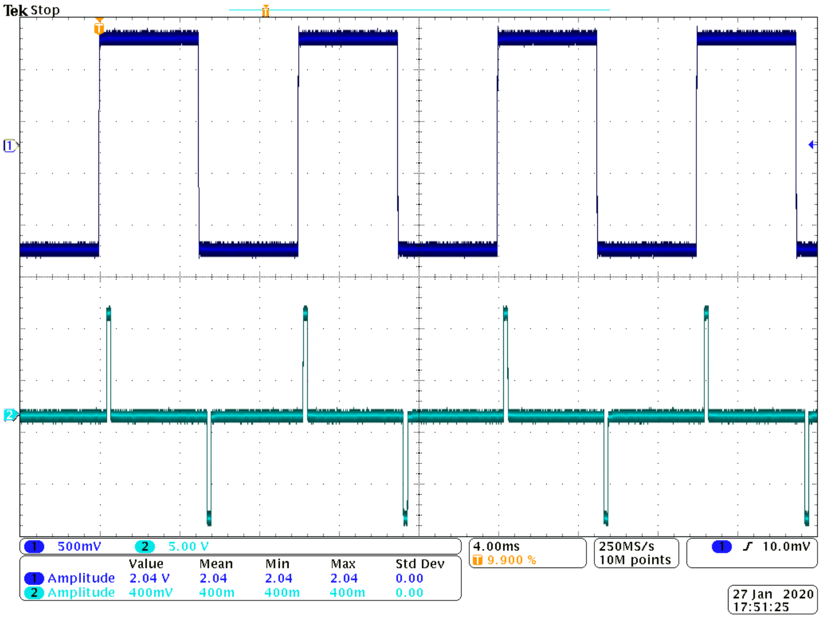
Figure 5.9: Setpoint and control variable — test of the proportional term

Although the regulator's bandwidth increased, the change that was not big enough for the PI controller to respond to the generated impulse signal and its impulse respone matches the one presented in figure 5.6. The PI controller, similarly to the one from the previous subsection, reacted on the step change of the input signal. Here as well, the anti–windup protection worked and the regulator reacted almost instantly in spite of being at rail for some time. It is illustrated by the waveform in figure 5.12.



Effects of anti-windup protection

Figure 5.10: Setpoint and control variable — test of the integral term



	Value	Mean	Min	Max	Std Dev
1 Amplitude	2.04 V	2.04	2.04	2.04	0.00
2 Amplitude	400mV	400m	400m	400m	0.00

Figure 5.11: Setpoint and control variable — test of the derivative term

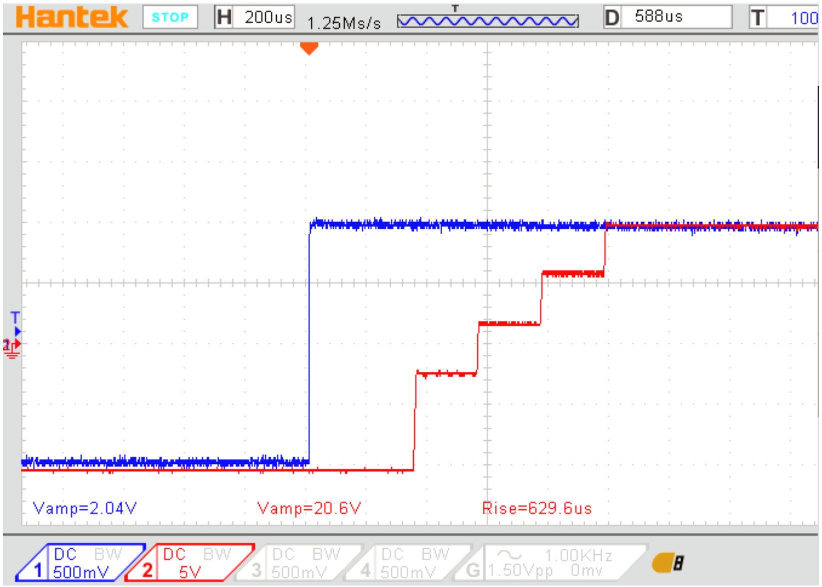
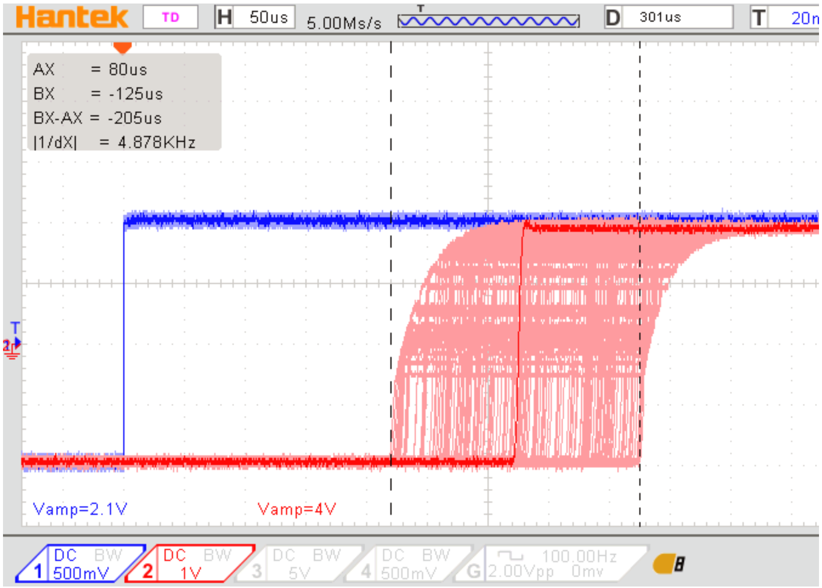


Figure 5.12: The controller step response



```
AX     = 80us
BX     = -125us
BX-AX = -205us
|1/dX|   = 4.878KHz
```

Figure 5.13: The controller average latency

5.3. Tests of the bare–metal implementation

Estimating the bare–metal implementation's bandwidth required other approach than the previous two. Control over the PI regulator from the ARTIQ level was not possible and integration with the ARTIQ core's RTIO had not been implemented either. Therefore, a time of one servo cycle, which was calculated during the *Servo* module implementation (see section 4.2.6) was used.

To compile and run the implementation, *main* and *eem2* files were created. These files served the purpose of the *Servo* module declaration and gaining access to the EEM connectors associated with Zotino and Sampler. Listing 5.1 shows how the FPGA's clock buffer was accessed and assigned to the module's clock signal. It presents how to build the design as well. The code used for the bare–metal implementation testing is placed in Appendix 2.1 and Appendix 2.2.

Listing 5.1: Example of clock signal assignment and build command

```
clk125 = plat.request("clk125_gtp")
m.specials += [
    Instance("IBUFDS_GTE2", i_I = clk125.p, i_IB = clk125.n, o_O = clk_signal),
    Instance("BUFG", i_I = clk_signal, o_O = servo.cd_sys.clk)
]
plat.build(servo, run=True, build_dir = "building/pid/{}ch/pgia{:0>4x}/Kp_{}_Ki_{}".format(
        channels_no, pgia_init_val, Kps[0], Kis[0]), build_name = "top")
```

Proportional term

The improvement of controller's action can be realized immediately — the waveform generated by the proportional term is shown in figure 5.14. In this case, the controller did not introduced any significant phase shift, while being fed with the error signal of the frequency of 100 Hz. What is more, the amplification of the input signal by the factor of 2 was preserved.

Integral term

The integral action's test result was positive, as well. As shown in figure 5.15, its response to the square–shaped error signal was, however, much smoother than the responses of the previous two implementations. It was achieved thanks to the controller's much higher sample rate.

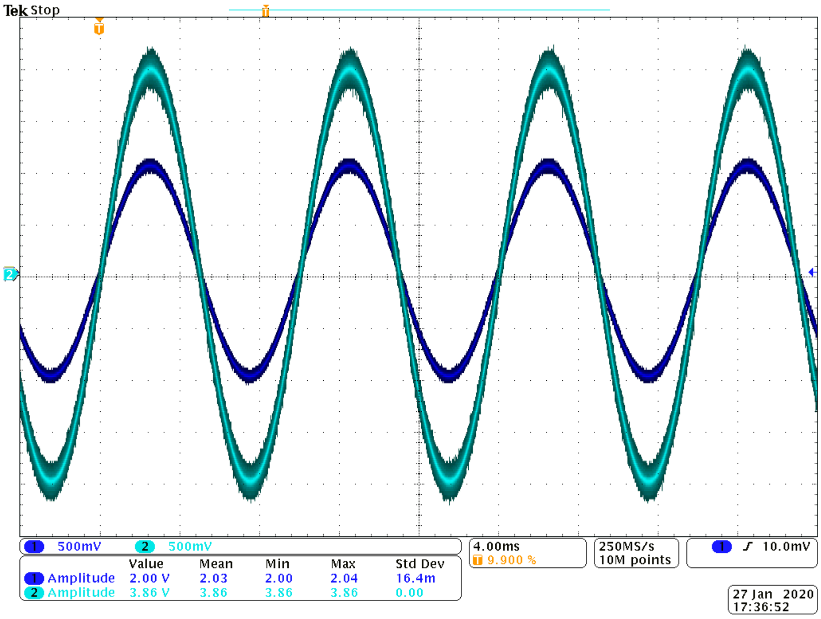

Tek Stop
1 500mV
2 500mV
Value Mean Min Max Std Dev
1 Amplitude 2.00 V 2.03 2.00 2.04 16.4m
2 Amplitude 3.86 V 3.86 3.86 3.86 0.00
4.00ms
9.900 %
250MS/s
10M points
1 10.0mV
27 Jan 2020
17:36:52

Figure 5.14: Setpoint and control variable — test of the proportional term

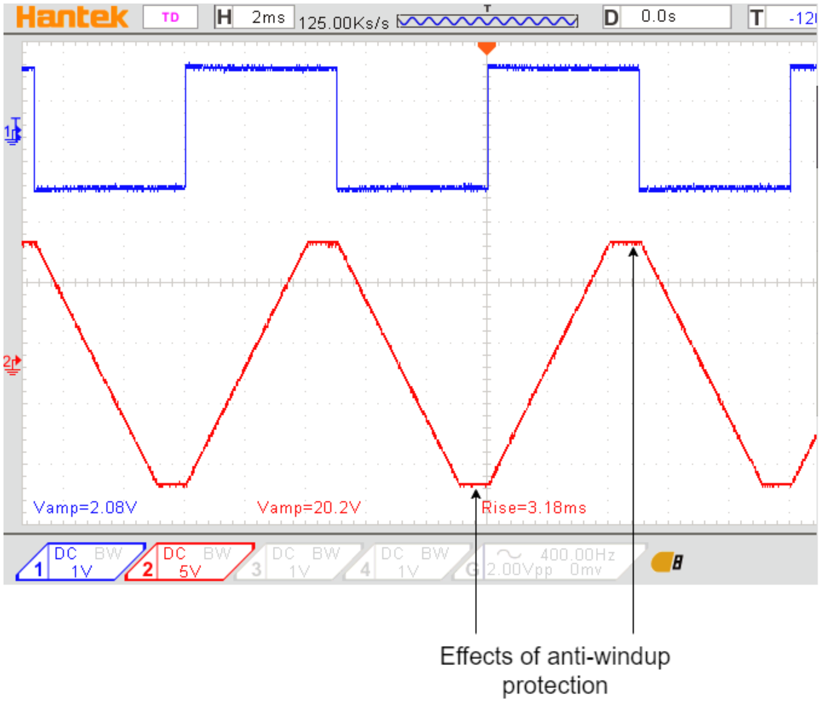


Figure 5.15: Setpoint and control variable — test of the integral term

PI controller

Contrary to the both ARTIQ–based implementations, the bare–metal one successfully detects the short impulse and generates a response based on that input. This, however, may be of ambivalent significance. On one hand, it demonstrates that the bare–metal implementation has the higher bandwidth than the ARTIQ–based ones. On the other hand it may be sensitive to undesirable external disturbances. Application of filters on the input of the PID controller should solve this problem when it occurs. The controller's response to the impulse is presented in figure 5.16, while its response to the step change in figure 5.17. What is of the extreme importance is the fact, that the controller reacts to the error signal's change almost immediately.

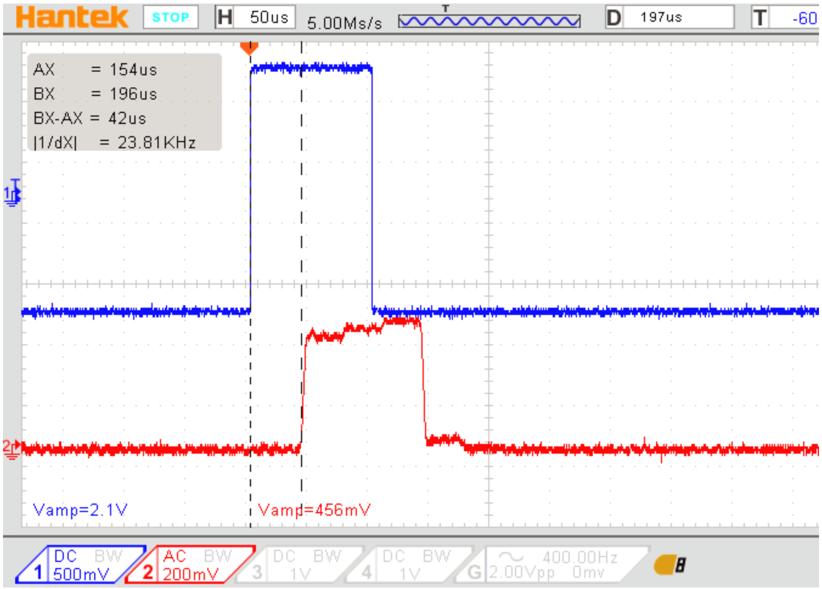


Figure 5.16: The controller impulse response

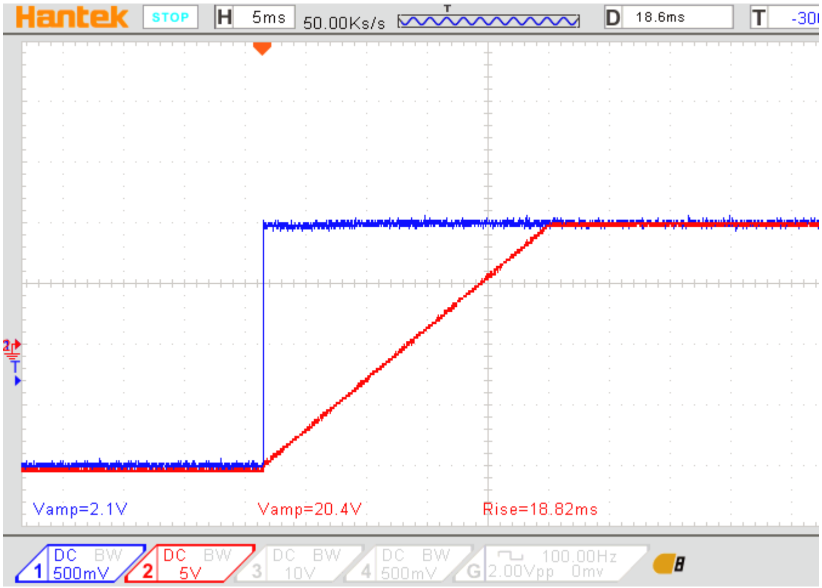


Figure 5.17: The controller step response

As a result of the measurements shown in figure 5.18, the controller's bandwidth of approximately 15.6 kHz and its average latency of around 31 μs were calculated. It is more than tenfold improvement in terms of the latency in comparison with the best of two ARTIQ implementations, and about fivefold in terms of the available bandwidth.

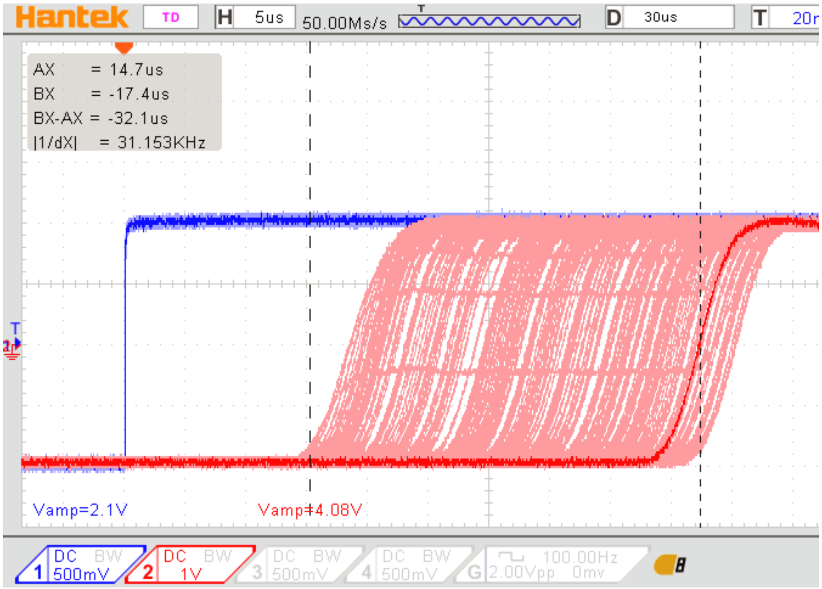


Figure 5.18: The controller average latency

5.4. Tests and measurements summary

Tests conducted on different PID controller implementations have shown that they are able to provide at least the proportional–intergral type of control. The most important parameters of each regulator, such as its bandwidth or latency, were measured and are presented in table 4. What was realized is that the best control should be provided by the PI regulator implemented as a bare–metal controller — its measured results were significantly better than the other two's. Since all the implementations are using the same hardware and the same connections, this difference is a result of the ARTIQ software working above the Kasli controller — it does not allow to fully exploit advantages of ADC and DAC integrated circuits at the moment.

Table 4: Comparison of implementations' parameters

	ARTIQ		Bare–metal
	With conversion	Without conversion	
Bandwidth	≈1.2 kHz	≈2.4 kHz	≈15.6 kHz
Sampling period	≈400 μ s	≈200 μ s	≈32 μ s
Latency	≈600 μs	≈340 μs	≈31 μs

6. Conclusion and thesis summary

During this thesis project a new PID controller was developed. The first two stages involved a pure software implementation of the regulator. This way of implementing the proportional–integral–derivative controller's functionality, in both cases resulted in regulator's small bandwidth. The second approach involved developing a PID controller, as well, but rather directly in FPGA's logic than in software. Its outcome should be seen as a successful end of a project, since the controller's bandwidth is of 15 kHz order of magnitude. This implementation, however, could have been even more promising, if some other DAC integrated circuit had been installed on the Zotino board. As stated in IC data sheet [52], it needs more than 30 μs to settle its output before new data may be processed, which makes it the main limiting factor to the regulator's bandwidth.

What has to be taken into account is, however, the fact that the achieved bandwidth does not guarantee the closed–loop control to operate at such high sample rates. The phase shift, which was clearly visible with the bare eye in both ARTIQ–based implementations, would probalby limit the system's bandwidth.

From the three implementations, the last one — the bare–metal implementation — has been shown to offer the best performance for a control loop. What is more, its only disadvantage in comparison to the previous to, is the lack of the derivative part, which, however, had not been required by the technical assumptions and is an acceptable trade–off for the higher bandwidth and much smaller average latency.

All of the requirements set in section 2 has been fulfilled: the PID controller is run on the Sinara hardware and has been developed using either ARTIQ or Migen; anti–windup protection has been implemented and proved to perform its task properly in every implementation. The process is driven directly with voltage using the Zotino board, which allows setting the control variable in the range of ± 10 V. The error signal is sampled by the Sampler board, which is capable of accepting data in the voltage range of ± 10 V as well. Achieved bandwidth and latency are better than required — even in the worst case scenario.

The devices used during this thesis projet can be put inside the rack for user's convenience, as shown in figure 6.1.

6.1. Further work

Although the controller is proved to be working and can be used to control laser sources already, the most effective implementation's application can cause some troubles for many inexperienced users because its usage requires the knowledge on how to flash binaries into the FPGA's memory. What is more, the usage of the implemented PID controller prevents the user from performing anything else on Sinara core. Therefore, the next step of development should be to incorporate the bare–metal implementation into the ARTIQ.

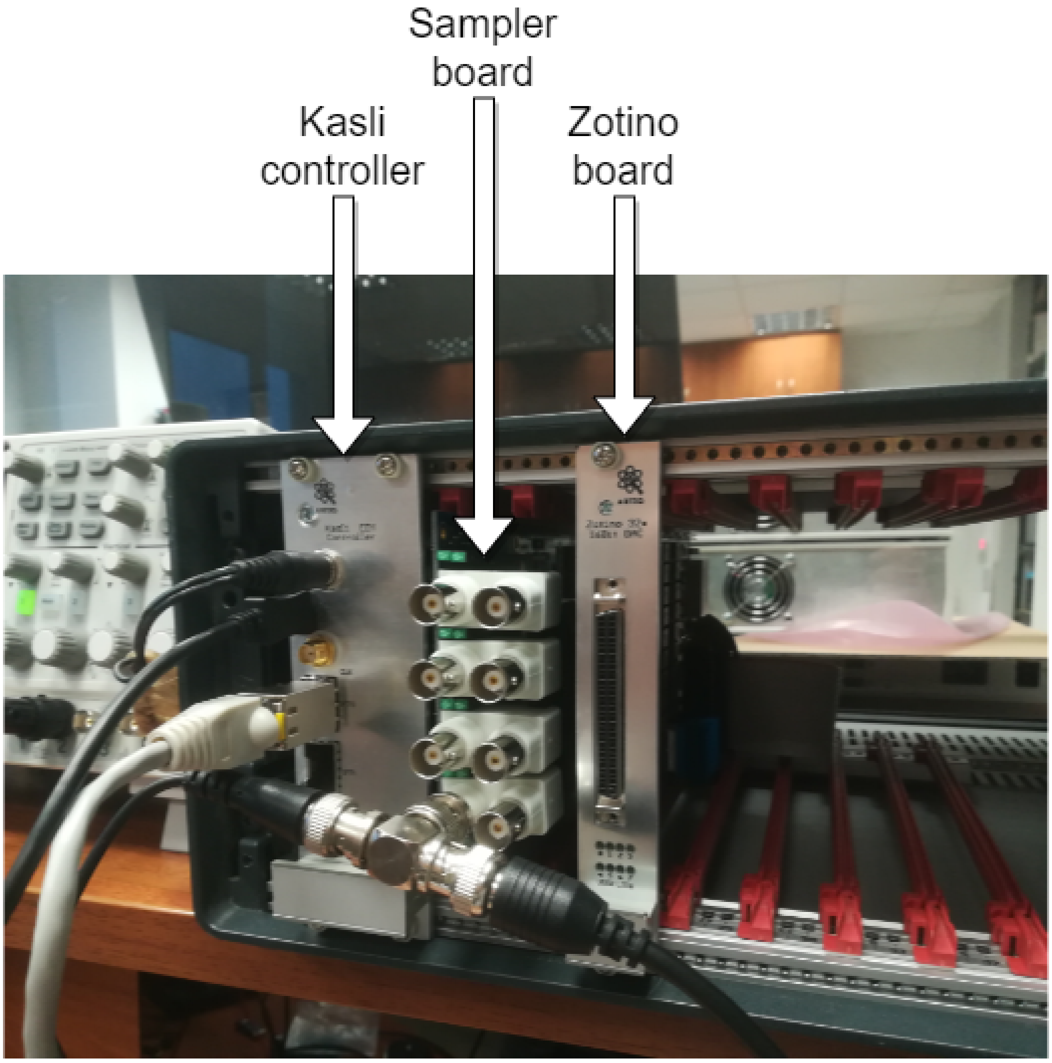


Figure 6.1: Photograph of the hardware setup used in thesis project

It would allow using all of the hardware available, with the PID control loop running in background. The development could go even further and the bare–metal implementation could be adjusted to perform control loop on eight channels independently with little effort.

In figure 6.2 the hardware setup used at Humboldt University of Berlin for performing quantum physics experiments is presented. The connections are made with a laser source to drive it with the control loop and with the PFD to sample the error signal. With the dashed line the Sinara hardware family is marked.

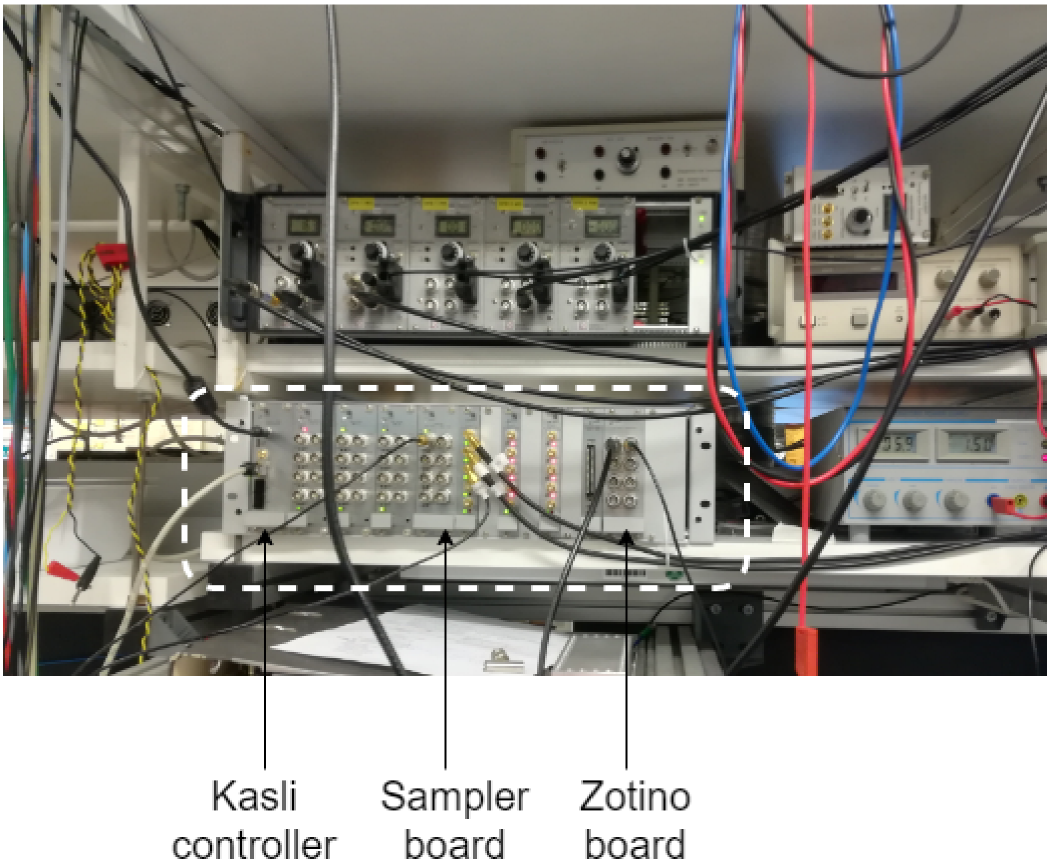


Figure 6.2: Photograph of the hardware setup at HUB

References

[1] M. H. Anderson, J. R. Ensher, M. R. Matthews, C. E. Wieman, and E. A. Cornell, “Observation of Bose-Einstein Condensation in a Dilute Atomic Vapor”, *Science*, vol. 269, no. 5221, pp. 198–201, 1995.

[2] E. Hinds, K. Burnett, C. Foot, and E. Riis, “Bose–Einstein condensates”, (visited on 01.01.2020): `https://www.iop.org/publications/iop/archive/file_52185.pdf`, 2005.

[3] I. Yukalov V., “Cold Bosons in Optical Lattices”, *Laser Physics*, vol. 19, no. 2, pp. 1–100, 2009.

[4] NIST, “Bose-Einstein Condensate: A New Form of Matter”, (visited on 01.01.2020): `https://www.nist.gov/news-events/news/2001/10/bose-einstein-condensate-new-form-matter`, 2001.

[5] L. Fallani and A. Kastberg, “Cold atoms: A field enabled by light”, *Europhysics Letters*, vol. 110, no. 5, 2015.

[6] S. N. Staff, “First Atomic Laser”, (visited on 01.01.2020): `https://www.sciencemag.org/news/1997/01/first-atomic-laser`, 1997.

[7] M. Demmer, R. Fonseca, and F. Koushanfar, “Richard Feynman: Simulating Physics with Computer”, (visited on 01.01.2020): `http://bit.do/fisica_feynman`.

[8] T. Bravo, C. Sabín, and I. Fuentes, “Analog quantum simulation of gravitational waves in a Bose-Einstein condensate”, *EPJ Quantum Technology*, vol. 2, no. 1, 2015.

[9] T. N. Theis and H. S. Philip Wong, “The End of Moore’s Law: A New Beginning for Information Technology”, *Computing in Science & Engineering*, vol. 19, no. 2, pp. 41–50, 2017.

[10] B. M. Mihalcea, *Dynamic stability for a system of ions in a paul trap*, 2019. arXiv: `1904.13393 [physics.atom-ph]`.

[11] C. E. Wieman, D. E. Pritchard, and D. J. Wineland, “Atom cooling, trapping, and quantum manipulation”, *Reviews of Modern Physics*, vol. 71, no. 2, S253–S262, 1999.

[12] W. D. Phillips and H. J. Metcalf, “Cooling and trapping atoms”, *Scientific American*, vol. 256, pp. 50–56, 1987.

[13] H. J. Metcalf and P. Van der Straten, “Laser cooling and trapping of neutral atoms”, *The Optics Encyclopedia: Basic Foundations and Practical Applications*, 2007.

[14] W. D. Phillips, “Nobel lecture: Laser cooling and trapping of neutral atoms”, *Reviews of Modern Physics*, vol. 70, no. 3, pp. 721–741, 1998.

[15] M. Lipka, M. Parniak, and W. Wasilewski, “Optical frequency locked loop for long-term stabilization of broad-line dfb laser frequency difference”, *Applied Physics B*, vol. 123, no. 9, 2017.

[16] M. C. Wu, C. Chang-Hasnain, E. K. Lau, and Xiaoxue Zhao, “High speed modulation of semiconductor lasers”, in *2008 International Nano-Optoelectronics Workshop*, IEE, 2008, pp. 9–10. DOI: `10.1109/INOW.2008.4634411`.

[17] G. Puentes, “Laser frequency offset locking scheme for high-field imaging of cold atoms”, *Applied Physics B*, vol. 107, no. 1, pp. 11–16, 2012. DOI: 10.1007/s00340-012-4898-8.

[18] K. J. Aström and R. M. Murray, “Introduction”, in *Feedback Systems: An Introduction for Scientists and Engineers*, Princeton University Press, 2008, pp. 1–25.

[19] ——, “PID Control”, in *Feedback Systems: An Introduction for Scientists and Engineers*, Princeton University Press, 2008, pp. 293–313.

[20] H. Wang, *Digital PID control algorithm*, (visited on 3.01.2020): http://robotics.sjtu.edu.cn/upload/course/1/files/Chapter51.pdf, 2016.

[21] Gorinevsky, *Sampled Time Control*, (visited on 13.01.2020): https://web.stanford.edu/class/archive/ee/ee392m/ee392m.1056/Lecture5_Digital.pdf.

[22] A. V. Oppenheim, J. R. Buck, and R. W. Schafer, “The Z transform”, in *Discrete-Time Signal Processing; 2nd ed.* Princeton University Press, 1998, pp. 94–140.

[23] S. Raghunath, *Digital Loop Exemplified*, 2011.

[24] Xilinx, *UltraScale Architecture DSP Slice — User Guide*, (visited on 13.01.2020): https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf.

[25] C. Vogt, M. Woltmann, H. Albers, D. Schlippert, S. Herrmann, E. M. Rasel, *et al.*, *Evaporative cooling from an optical dipole trap in microgravity*, 2019. arXiv: 1909.03800 [physics.atom-ph].

[26] K. Frye, S. Abend, W. Bartosch, A. Bawamia, D. Becker, H. Blume, *et al.*, *The bose-einstein condensate and cold atom laboratory*, 2019. arXiv: 1912.04849 [physics.atom-ph].

[27] L. Liu, D. Lü, W. Chen, T. Li, Q. Qu, B. Wang, *et al.*, *Tests of cold atom clock in orbit*, 2017. arXiv: 1709.03256 [physics.atom-ph].

[28] NIST, *Open-Source Software for Quantum Information*, (visited on 2.01.2020): https://www.nist.gov/news-events/news/2015/01/open-source-software-quantum-information, 2015.

[29] V. Negnevitsky, “Feedback-stabilised quantum states in a mixed-species ion system”, PhD thesis, ETH Zurich, 2018. DOI: https://doi.org/10.3929/ethz-b-000295923.

[30] R. Losito and A. Masi, *CERN Uses NI LabVIEW Software and PXI Hardware to Control World’s Largest Particle Accelerator*, (visited on 2.01.2020): http://www.ni.com/pl-pl/innovations/case-studies/19/cern-uses-ni-labview-software-and-pxi-hardware.html.

[31] C. Darsow-Fromm, L. Dekant, S. Grebien, M. Schröder, R. Schnabel, and S. Steinlechner, *Nqontrol: An open-source platform for digital control-loops in quantum-optical experiments*, (visited on 2.01.2020): https://arxiv.org/abs/1911.08824, 2019.

[32] Z. Instruments, *First Commercial Quantum Computing Control System for Europe’s Quantum Computer*, (visited on 2.01.2020): https://www.zhinst.com/news/first-commercial-quantum-computing-control-system.

[33] M-Labs, *ARTIQ*, (visited on 2.01.2020): https://m-labs.hk/experiment-control/artiq/.

[34] National Instruments, *NI PXI Modular Instrument Design Advantages*, (visited on 2.01.2020): http://www.ni.com/pl-pl/innovations/white-papers/11/ni-pxi-modular-instrument-design-advantages.html.

[35] ——, *NI PXI Timing and Synchronization Design Advantages*, (visited on 2.01.2020): http://www.ni.com/pl-pl/innovations/white-papers/11/ni-pxi-timing-and-synchronization-design-advantages.html.

[36] ——, *PXI Express FAQ*, (visited on 29.01.2020): https://www.ni.com/pl-pl/innovations/white-papers/06/pxi-express-faq.html.

[37] Z. Instruments, *HDAWG Arbitrary Waveform Generator*, (visited on 3.01.2020): https://www.zhinst.com/products/hdawg#overview.

[38] ——, *PQSC Programmable Quantum System Controller*, (visited on 2.01.2020): https://www.zhinst.com/products/pqsc#overview.

[39] ——, *LabOne. All in One*, (visited on 2.01.2020): https://www.zhinst.com/products/quantum#overview.

[40] S. Bourdeauducq, *Timing control in ARTIQ*, (visited on 3.01.2020): https://m-labs.hk/docs/artiq/slides_timing.pdf.

[41] M-Labs and contributors, *ARTIQ Documentation*, (visited on 3.01.2020): https://m-labs.hk/artiq/manual.pdf.

[42] M-Labs, *Migen*, (visited on 3.01.2020): https://m-labs.hk/gateware/migen/.

[43] G. Kasprowicz, R. Jördens, J. Britton, D. Allcock, and S. Bourdeauducq, *Sinara Wiki Page*, (visited on 3.01.2020): https://github.com/sinara-hw/meta/wiki.

[44] P. Kulik, G. Kasprowicz, and M. Gąska, "Driver module for quantum computer experiments: Kasli", in *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2018*, R. S. Romaniuk and M. Linczuk, Eds., International Society for Optics and Photonics, vol. 10808, SPIE, 2018, pp. 1255–1258. DOI: 10.1117/12.2501709. [Online]. Available: https://doi.org/10.1117/12.2501709.

[45] C. Grzeschik, M. Krutzik, and A. Peters, *BECCAL - Bose-Einstein Condensate and Cold Atom Lab*, (visited on 3.01.2020): https://www.physics.hu-berlin.de/en/qom/research/BECCAL/standardseite.

[46] A. Dinkelaker, C. Grzeschik, J. Pahl, M. Schiemangk, M. Krutzik, and A. Peters, *QUANTUS - Bose Einstein condensates in microgravity*, (visited on 3.01.2020): https://www.physics.hu-berlin.de/en/qom/research/droptower.

[47] G. Kasprowicz, J. Britton, D. Allcock, and S. Bourdeauducq, *Urukul Wiki Page*, (visited on 20.01.2020): https://github.com/sinara-hw/Urukul/wiki.

[48] K. J. Aström and R. M. Murray, "Frequency Domain Design", in *Feedback Systems: An Introduction for Scientists and Engineers*, Princeton University Press, 2008, pp. 315–344.

[49] G. Kasprowicz, J. Britton, R. Jördens, D. Allcock, S. Bourdeauducq, and P. Kulik, *Sampler Wiki Page*, (visited on 20.01.2020): `https://github.com/sinara-hw/Sampler/wiki`.

[50] G. Kasprowicz, R. Jördens, D. Allcock, and S. Bourdeauducq, *Zotino Wiki Page*, (visited on 27.01.2020): `https://github.com/sinara-hw/Zotino/wiki`.

[51] *10 MHz, 20 V/s, G = 1, 10, 100, 1000 iCMOS Programmable Gain Instrumentation Amplifier*, 2012.

[52] *32-Channel, 16-/14-Bit, Serial Input, Voltage Output DAC*, 2011.

[53] K. J. Åström, *Control System Design Lecture Notes: PID Control*, (visited on 25.01.2020): `https://www.cds.caltech.edu/~murray/courses/cds101/fa02/caltech/astrom-ch6.pdf`, 2002.

[54] J. G. Ziegler, N. B. Nichols, *et al.*, “Optimum settings for automatic controllers”, *trans. ASME*, vol. 64, no. 11, 1942.

[55] C. E. Shannon, “Communication in the presence of noise”, *Proceedings of the IEEE*, vol. 72, no. 9, pp. 1192–1201, Sep. 1984, ISSN: 1558-2256. DOI: `10.1109/PROC.1984.12998`.

List of Symbols and Abbreviations

ADC – Analog–to–Digital Converter

AOM – Acousto–Optic Modulation

ARTIQ – Advanced Real–Time Infrastructure for Quantum physics

ASF – Amplitude Scale Factor

BS – Beam Splitter

BEC – Bose–Einstein Condensate

CERN OHL – CERN Open Hardware License

CFG – Configuration register

CPU – Central Processing Unit

CV – Control Variable

DAC – Digital–to–Analog Converter

DDS – Direct Digital Synthesizer

DIO – Digital Input/Output

DRTIO – Distributed Real Time Input/Output

DSP – Digital Signal Processing

EEM – Eurocard Extension Module

FIFO – First In, First Out

FIR – Finite Impulse Response

FPGA – Field–Programmable Gate Array

FSM – Finite State Machine

FTW – Frequency Tuning Word

GUI – Graphical User Interface

HUB – Humboldt University of Berlin

IC – Integrated Circuit

IIR – Infinite Impulse Response

I/O – Input/Output

JILA – Joint Institute for Laboratory Astrophysics

JTAG – Joint Test Access Group

LED – Light–Emitting Diode

LO – Local Oscillator

LP filter – Low–Pass filter

LVDS – Low-Voltage Differential Signalling

MOT – Magneto–Optical Trap

NIST – National Institute of Standards and Technology

PFD – Phase–Frequency Detector

PGIA – Programmable–Gaine Instrumentation Amplifier

PI – Propotional–Integral

PID – Propotional–Integral–Derivative

POW – Phase Offset Word
PV – Process Variable
QCCS – Quantum Computing Control System
QOM – Optical Metrology Group
QSPI – Quad SPI
RAM – Random–Access Memory
RF – Radio Frequency
RPC – Remote Procedure Call
SPI – Serial Peripheral Interface
USB – Universal Serial Bus
VHDL – Very High Speed Integrated Circuit Hardware Description Language
VLSI – Very Large–Scale Integration

List of Figures

List of Tables

List of Appendices

Appendix 1. Details on the used boards

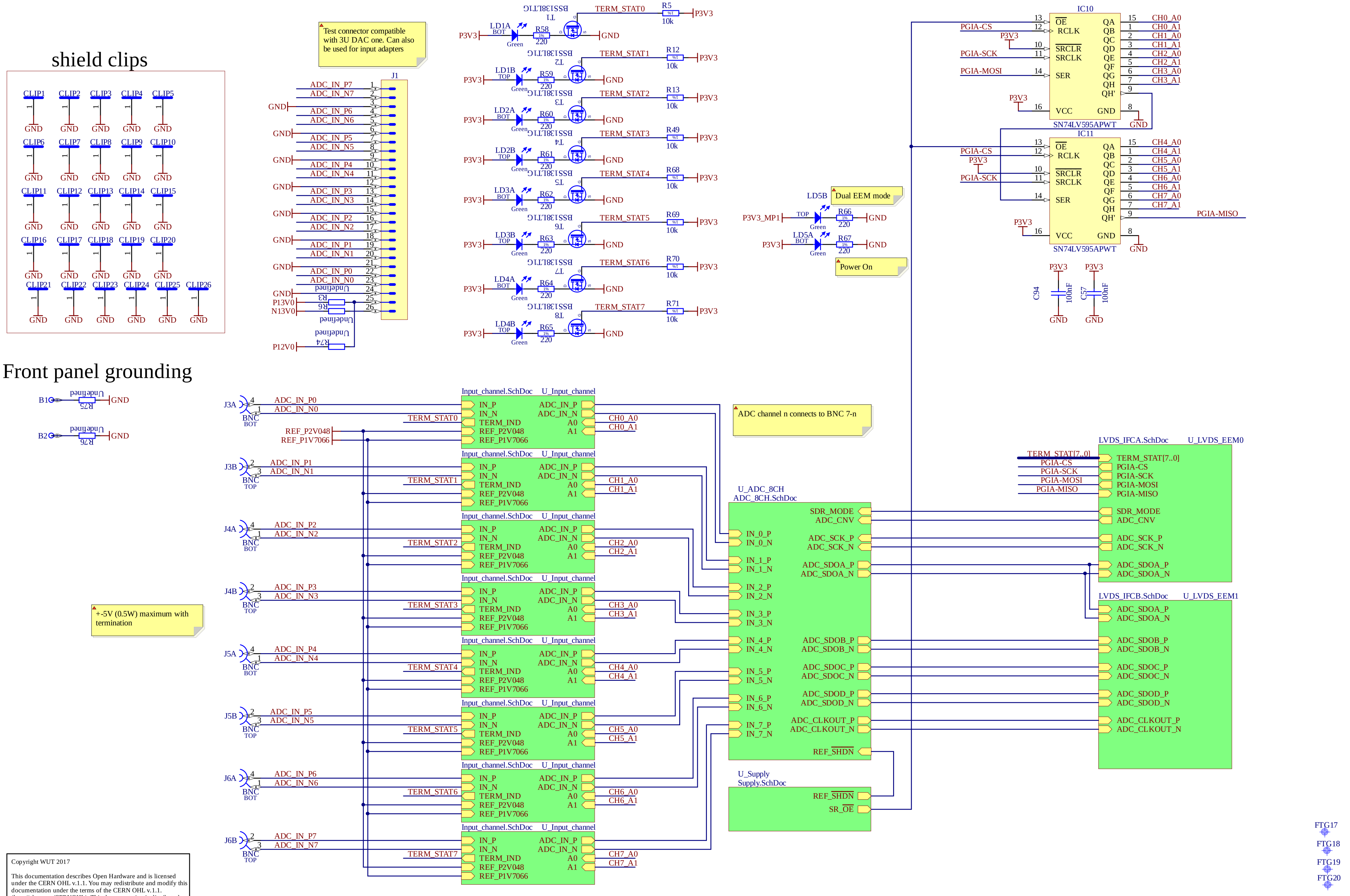
Appendix 1.1. Details on Sampler

According to the Sampler project page [49], the board itself ‘is an 8-channel, 16-bit ADC EEM with an update rate of up to 1.5MSPS (all channels simultaneously). It has low-noise differential front end with a digitally programmable gain, providing full-scale input ranges between +-10mV (G=1000) and +-10V (G=1).’

What is more, the Sampler’s key features are [49]:

- ‘Current hardware revision: v2.2
- Width: 8HP
- Channel count: 8
- Resolution: 16-bit
- Sample rate: up to 1.5 MHz
- Sustained aggregate data rate in single-EEM mode (8 channel readout): 700 kHz
- Sustained per-channel data rate in dual-EEM mode (SU-Servo): 1 MHz
- Note that the bandwidth specifications on this page are for the hardware only; ARTIQ kernel and RTIO overhead often make the effective sample rate lower.
- Bandwidth: 200kHz -6dB bandwidth for G=1, 10, 100, 90kHz for G=1000
- Input ranges: +-10V (G=1), +-1V (G=10), +-100mV (G=100), +-10mV (G=1000)
- DC input impedance:
  - Termination off: 100k from input signal and ground connections to PCB ground
  - Termination on: signal 50Ohm terminated to PCB ground, input ground shorted to PCB ground
- ADC: LTC2320-16
- PGIA: AD8253
- EEM connectors: power and digital communication supplied by one or two EEM connectors.’

Furthermore, the Sampler can operate in one of the two modes: as a standard SPI device, or in a fast mode via a source–synchronous LVDS interface [49]. According to the Sampler wiki page, the Sampler’s channels can be read out at 1.5 MHz via the source–synchronous interface.


shield clips
CLIP1 CLIP2 CLIP3 CLIP4 CLIP5
CLIP6 CLIP7 CLIP8 CLIP9 CLIP10
CLIP11 CLIP12 CLIP13 CLIP14 CLIP15
CLIP16 CLIP17 CLIP18 CLIP19 CLIP20
CLIP21 CLIP22 CLIP23 CLIP24 CLIP25 CLIP26
GND
Front panel grounding
B1 R75 Undefined GND
B2 R76 Undefined GND
J1
ADC_IN_P7 ADC_IN_N7 ADC_IN_P6 ADC_IN_N6 ADC_IN_P5 ADC_IN_N5 ADC_IN_P4 ADC_IN_N4 ADC_IN_P3 ADC_IN_N3 ADC_IN_P2 ADC_IN_N2 ADC_IN_P1 ADC_IN_N1 ADC_IN_P0 ADC_IN_N0
P13V0 N13V0 P12V0
R3 R6 R74 Undefined
LD1A LD1B LD2A LD2B LD3A LD3B LD4A LD4B
TOP BOT Green
R58 R59 R60 R61 R62 R63 R64 R65 220
T1 T2 T3 T4 T5 T6 T7 T8 BSS138LT1G
TERM_STAT0 TERM_STAT1 TERM_STAT2 TERM_STAT3 TERM_STAT4 TERM_STAT5 TERM_STAT6 TERM_STAT7
R5 R12 R13 R49 R68 R69 R70 R71 10k
P3V3
LD5B LD5A R66 R67 220 P3V3_MP1
IC10 IC11 SN74LV595APWT
OE RCLK SRCLR SRCLK SER VCC QA QB QC QD QE QF QG QH QH' GND
PGIA-CS PGIA-SCK PGIA-MOSI PGIA-MISO
CH0_A0 CH0_A1 CH1_A0 CH1_A1 CH2_A0 CH2_A1 CH3_A0 CH3_A1
CH4_A0 CH4_A1 CH5_A0 CH5_A1 CH6_A0 CH6_A1 CH7_A0 CH7_A1
C94 C57 100nF
J3A J3B J4A J4B J5A J5B J6A J6B BNC TOP BOT
REF_P2V048 REF_P1V7066
Input_channel.SchDoc U_Input_channel
IN_P IN_N TERM_IND REF_P2V048 REF_P1V7066 ADC_IN_P ADC_IN_N A0 A1
U_ADC_8CH ADC_8CH.SchDoc
IN_0_P IN_0_N IN_1_P IN_1_N IN_2_P IN_2_N IN_3_P IN_3_N IN_4_P IN_4_N IN_5_P IN_5_N IN_6_P IN_6_N IN_7_P IN_7_N
SDR_MODE ADC_CNV ADC_SCK_P ADC_SCK_N ADC_SDOA_P ADC_SDOA_N ADC_SDOB_P ADC_SDOB_N ADC_SDOC_P ADC_SDOC_N ADC_SDOD_P ADC_SDOD_N ADC_CLKOUT_P ADC_CLKOUT_N REF_SHDN
U_Supply Supply.SchDoc REF_SHDN SR_OE
LVDS_IFCA.SchDoc U_LVDS_EEM0
TERM_STAT[7..0]
LVDS_IFCB.SchDoc U_LVDS_EEM1
FTG17 FTG18 FTG19 FTG20

Test connector compatible with 3U DAC one. Can also be used for input adapters

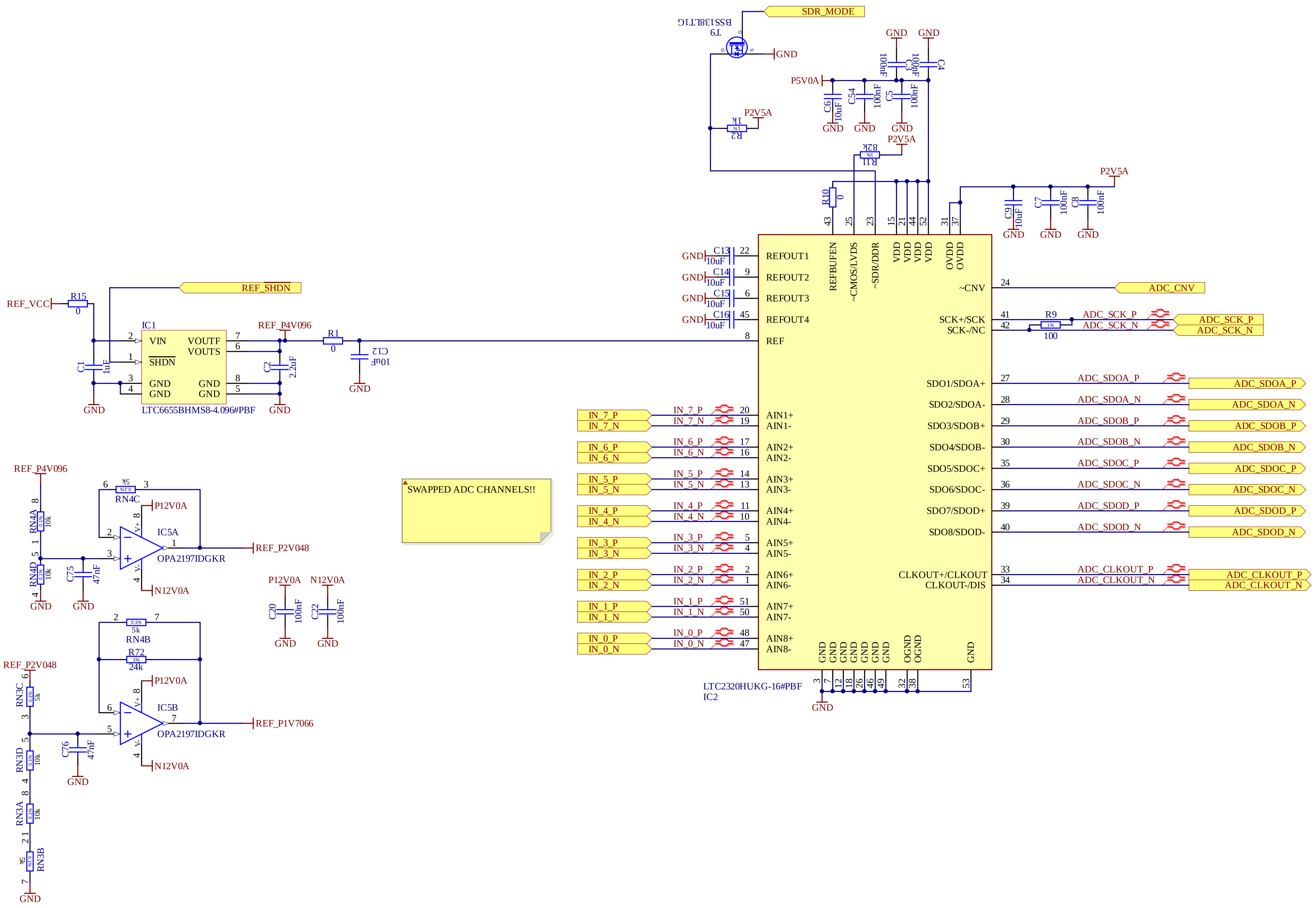
Dual EEM mode

Power On

ADC channel n connects to BNC 7-n

+-5V (0.5W) maximum with termination

Project/Equipment	Sampler		
Document		Designer G.K.	
		Drawn by G.K.	XX/XX/XXXX
Cannot open file D:\Dropbox\DESIGNS\MTCA_projects\SI	***Sampler - top*** ***8 channel ADC v2.2***	Check.by	-
		Last Mod. -	27.07.2018
		File Sampler.schdoc	
		Print Date 28.01.2020 03:34:21	Sheet 1 of 5
	Warsaw Univeristy of Technology *Nowowiejska 15/19* ***ISE***	ARTIQ	Size A3 Rev -



Project/Equipment Sampler

Document

Cannot open file D:\Dropbox\DESIGNS\MTCA_projects\SI

ADC + reference v2.2

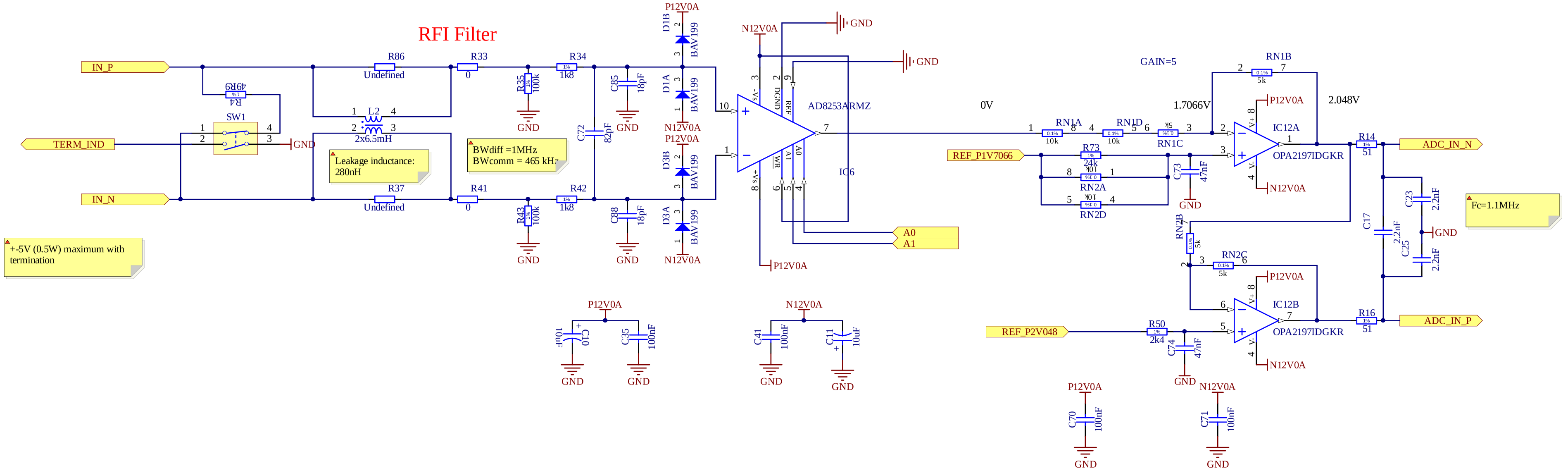
Designer	G.K.	
Drawn by	G.K.	XX/XX/XXXX
Check.by		-
Last Mod.	-	27.07.2018
File	ADC_8CH.SchDoc	
Print Date	28.01.2020 03:34:21	Sheet 2 of 5

Warsaw Univeristy of Technology
Nowowiejska 15/19

ISE

ARTIQ

Size A3 Rev -



Project/Equipment Sampler

Document

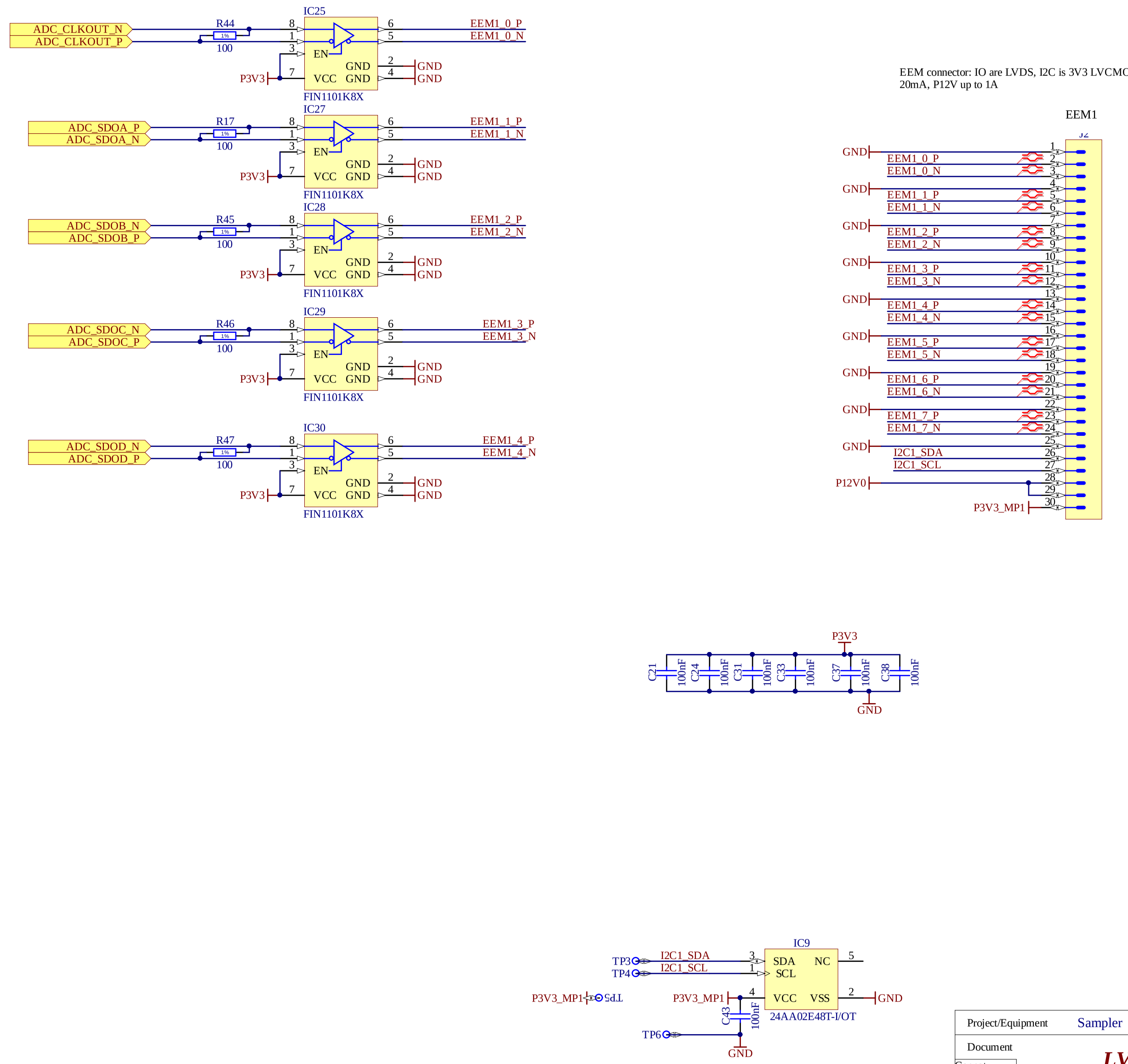
Cannot open file D:\Dropbox\DESIGNS\MTCA_projects\SI

Input chnannel
DPGA v2.2

Warsaw Univeristy of Technology ISE
Nowowiejska 15/19

Designer G.K.
Drawn by G.K. XX/XX/XXXX
Check.by -
Last Mod. - 27.07.2018
File Input_channel.SchDoc
Print Date 28.01.2020 03:34:22 Sheet 3 of 5

ARTIQ Size A3 Rev -



EEM connector: IO are LVDS, I2C is 3V3 LVCMOS, P3V3_MP up to 20mA, P12V up to 1A

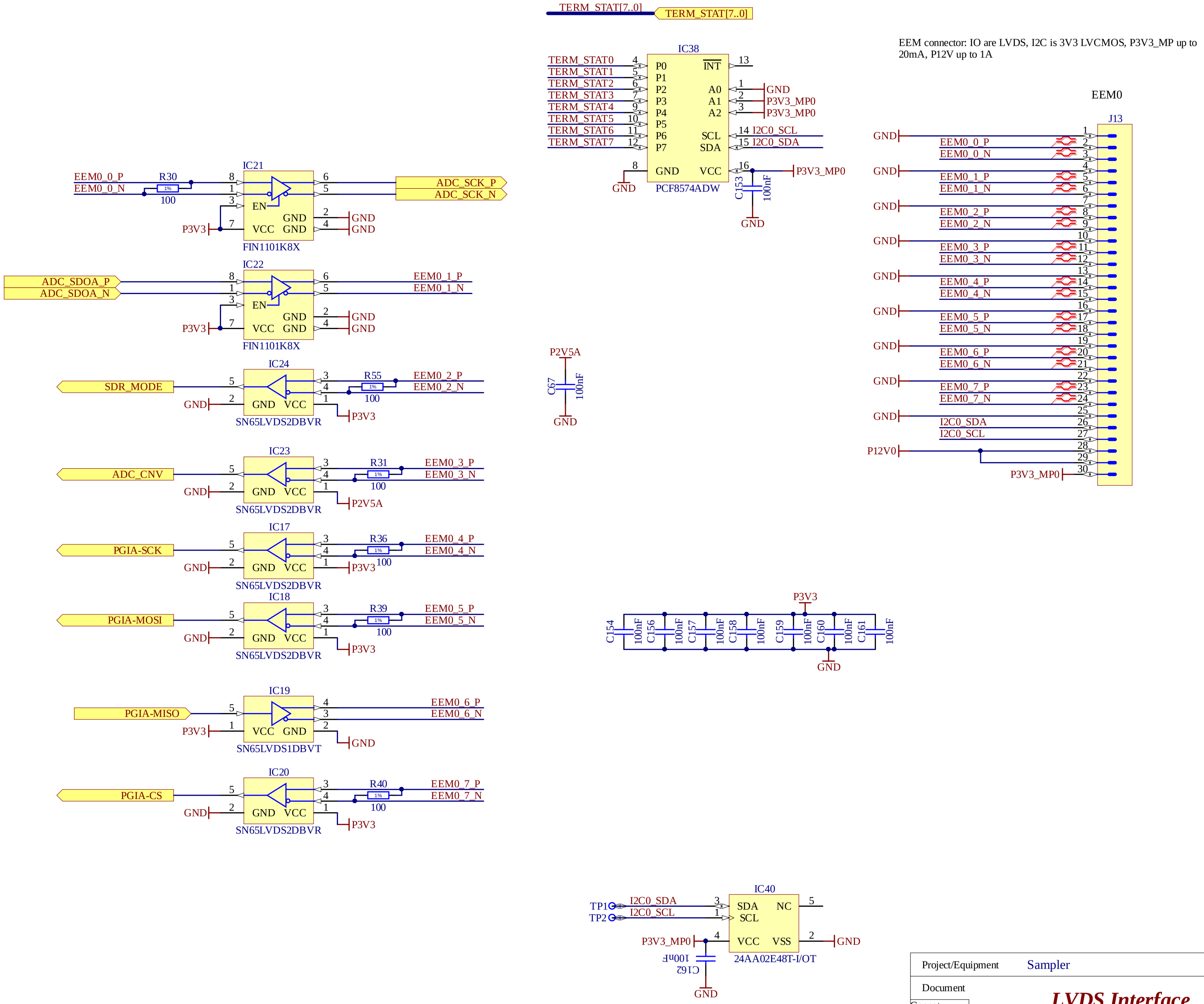
Project/Equipment Sampler

Document Cannot open file D:\Dropbox\DESIGNS\MTCA_projects\SI

LVDS Interface I2C logic v2.2

Warsaw Univeristy of Technology ISE
Nowowiejska 15/19

Designer G.K.
Drawn by G.K. XX/XX/XXXX
Check.by -
Last Mod. - 27.07.2018
File LVDS_IFCB.SchDoc
Print Date 28.01.2020 03:34:22 Sheet 4 of 5
ARTIQ Size A3 Rev -



EEM connector: IO are LVDS, I2C is 3V3 LVCMOS, P3V3_MP up to 20mA, P12V up to 1A

Project/Equipment Sampler

Document

Cannot open file D:\Dropbox\DESIGNS\MTCA_projects\SI

LVDS Interface
I2C logic v2.2

Warsaw Univeristy of Technology *ISE*
Nowowiejska 15/19

Designer	G.K.	
Drawn by	G.K.	XX/XX/XXXX
Check.by		-
Last Mod.	-	27.07.2018
File	LVDS_IFCA.SchDoc	
Print Date	28.01.2020 03:34:22	Sheet 4 of 5

ARTIQ

Size A3 Rev -

Power budget (max ratings):

P2V5	
LTC2320	38
RAIL POWER	38mA*2.5=0.095W
P3V3:	
SN65LVDS20DRFT	7*13.5=94.5
SN65LVDS2DBVR	5*8=40
SN65LVDS1DBVT	8
LEDs	8*5=40
RAIL POWER	183mA*3.3=0.603W
P5V0	
LTC2320	60
RAIL POWER	60*5=0.3W
P12V0	
AD8253ARMZ	8*4mA=32mA
OPA2197	8*2mA=16mA
AD8253 load (10V/20k)	8*0.5mA=4mA
OPA2197 load (500k:0.3V/50R)	8*4mA=32mA
RAIL POWER	84*12 = 1W
N12V0	
AD8253ARMZ	8*4mA=32mA
OPA2197	8*2mA=16mA
OPA2197 load (500k:0.3V/50R)	8*4mA=32mA
RAIL POWER	84*12 = 1W
DC/DC converter losses	
TPS62175:3.3 eff .95	0,05*(0.18+0.038)*2,5+26e-6*12=0.022W
TPS62175:6 eff .95	0,05*0.06*6+26e-6*12=0.018W
CC6-1212DF-E:13V eff 70%	0.3*13*0.084=0.33W
CC6-1212DF-E:-13V eff 70%	0.3*13*0.084=0.33W
LDO losses	
6V->5V	0.06*1V = 60mW
13V->12V	84mW
-13V->-12V	84mW
3.3V -> 2.5V	(3.3-2.5)*38 =30mW
Total power from 12V rail	12V 4.16W
Total current from 12V rail	0.35A

SHDN threshold is 0.8V

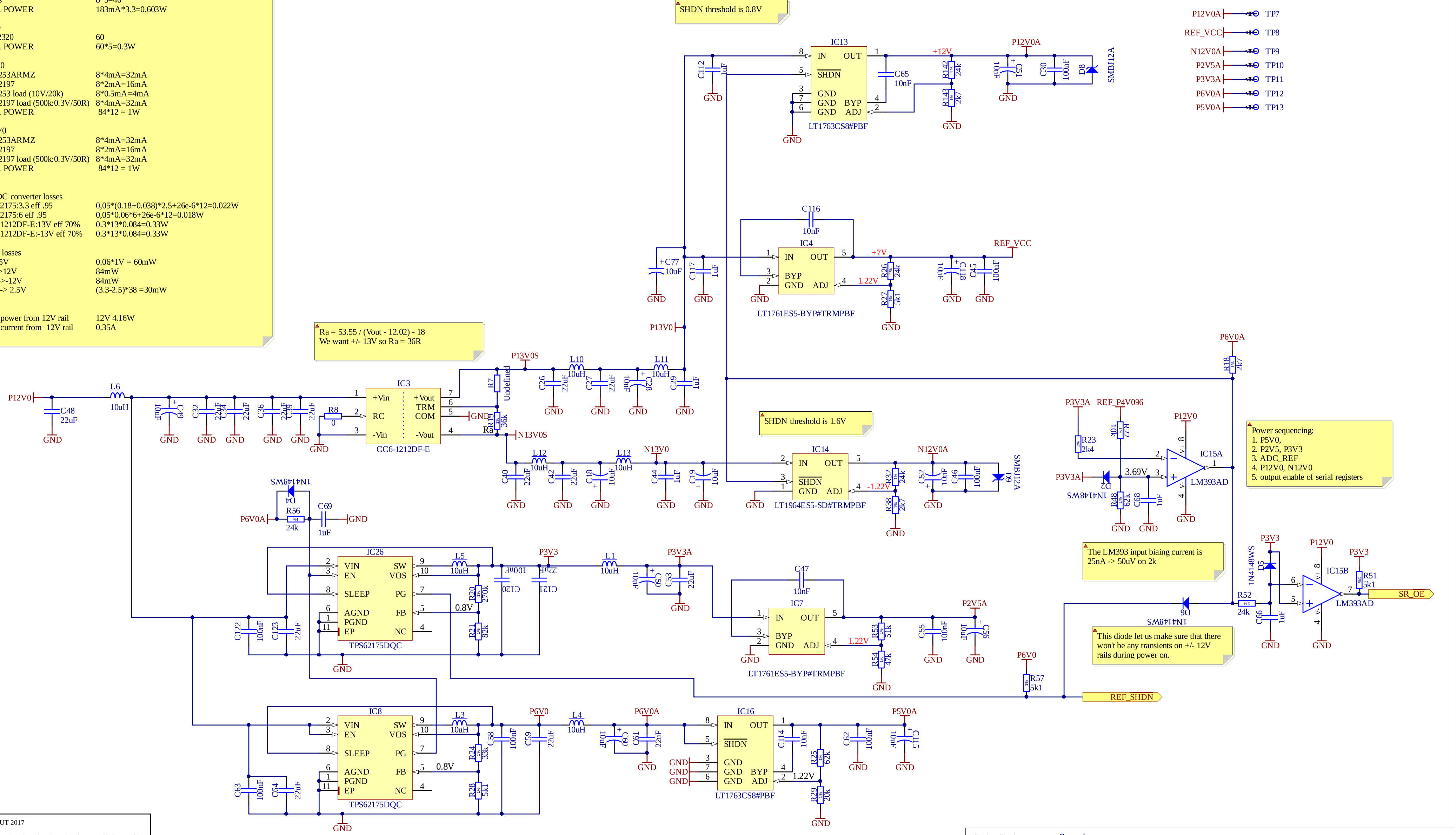
Ra = 53.55 / (Vout - 12.02) - 18
We want +/- 13V so Ra = 36R

SHDN threshold is 1.6V

Power sequencing:
1. P5V0,
2. P2V5, P3V3
3. ADC_REF
4. P12V0, N12V0
5. output enable of serial registers

The LM393 input biaing current is 25nA -> 50uV on 2k

This diode let us make sure that there won't be any transients on +/- 12V rails during power on.



Copyright WUT 2017

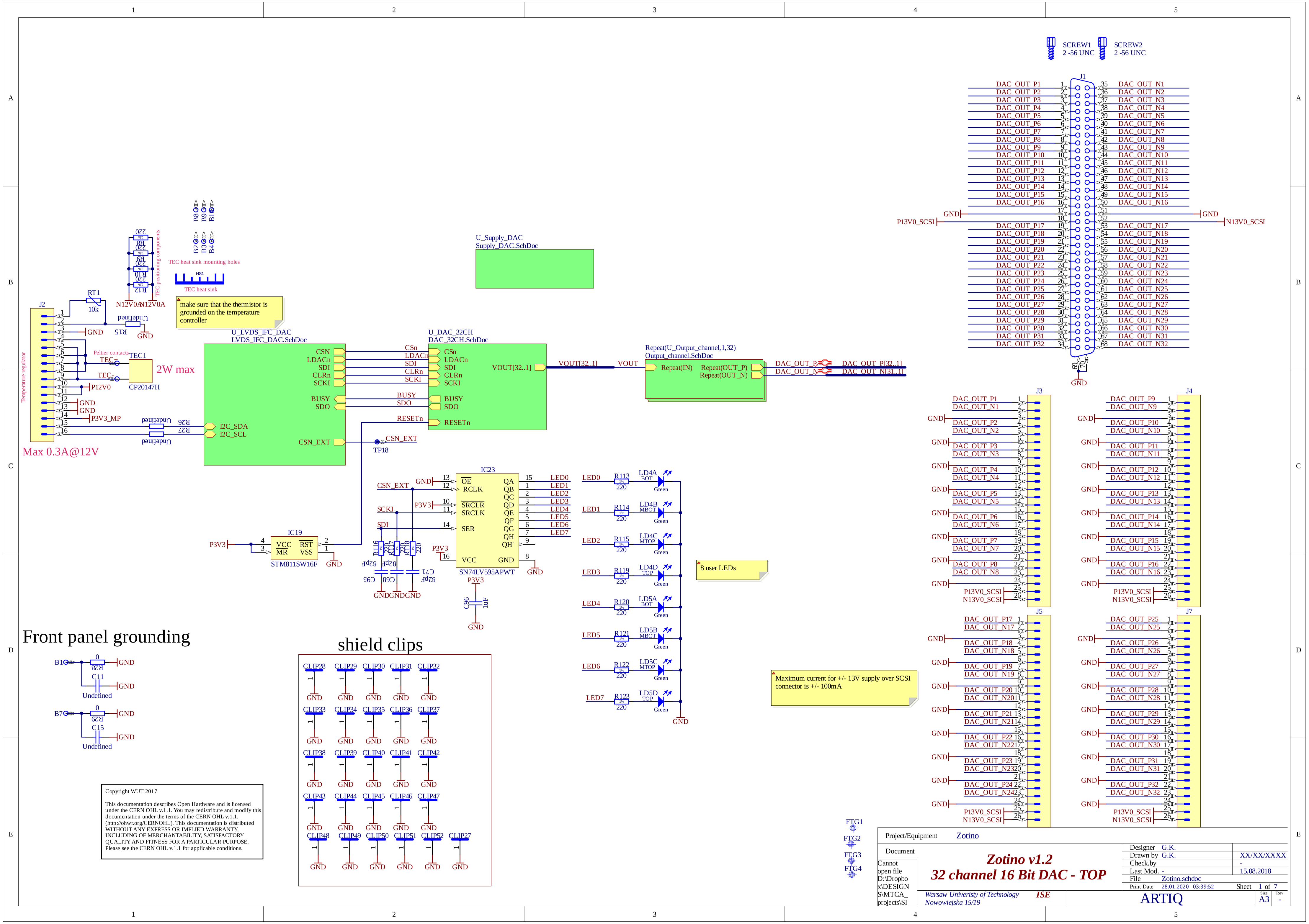
Project/Equipment	Sampler		
Document	**Power Supply converters + LDOs v2.2**	Designer G.K.	
Cannot open file D:\Dropbox\DESIGN S\MTCA_ projects\SI		Drawn by G.K.	XX/XX/XXXX
		Check.by	-
		Last Mod. -	27.07.2018
		File Supply.SchDoc	
		Print Date 28.01.2020 03:34:23	Sheet 5 of 5
Warsaw Univeristy of Technology Nowowiejska 15/19	*ISE*	ARTIQ	Size A3 Rev -

Appendix 1.2. Details on Zotino

Following the Zotino project page [50], it ‘is a 32-channel, 16-bit DAC EEM with an update rate of 1MSPS (divided between the channels). It was designed for low noise and good stability’.

The Zotino’s key features are:

- ‘Current hardware revision: Rev 1.1
- Width: 4HP
- Channel count: 32
- Resolution: 16-bit
- Update rate: 1MSPS, which may be divided arbitrarily between the channels
- Analogue bandwidth: 3rd-order Butterworth response with 75kHz cut-off; ?V/s slew-rate
- Output voltage: ±10V
- Output impedance: 470Ohm in parallel with 2.2nF
- DAC: AD5372BCPZ
- EEM connectors: power and digital communication supplied by a single EEM connector.
- Power consumption: 3W without load, 8.7W with max load on all channels’.



make sure that the thermistor is grounded on the temperature controller

8 user LEDs

Maximum current for +/- 13V supply over SCSI connector is +/- 100mA

Copyright WUT 2017

This documentation describes Open Hardware and is licensed under the CERN OHL v.1.1. You may redistribute and modify this documentation under the terms of the CERN OHL v.1.1. (http://ohwr.org/CERNOHL). This documentation is distributed WITHOUT ANY EXPRESS OR IMPLIED WARRANTY, INCLUDING OF MERCHANTABILITY, SATISFACTORY QUALITY AND FITNESS FOR A PARTICULAR PURPOSE. Please see the CERN OHL v.1.1 for applicable conditions.

Project/Equipment Zotino

Document

Zotino v1.2
32 channel 16 Bit DAC - TOP

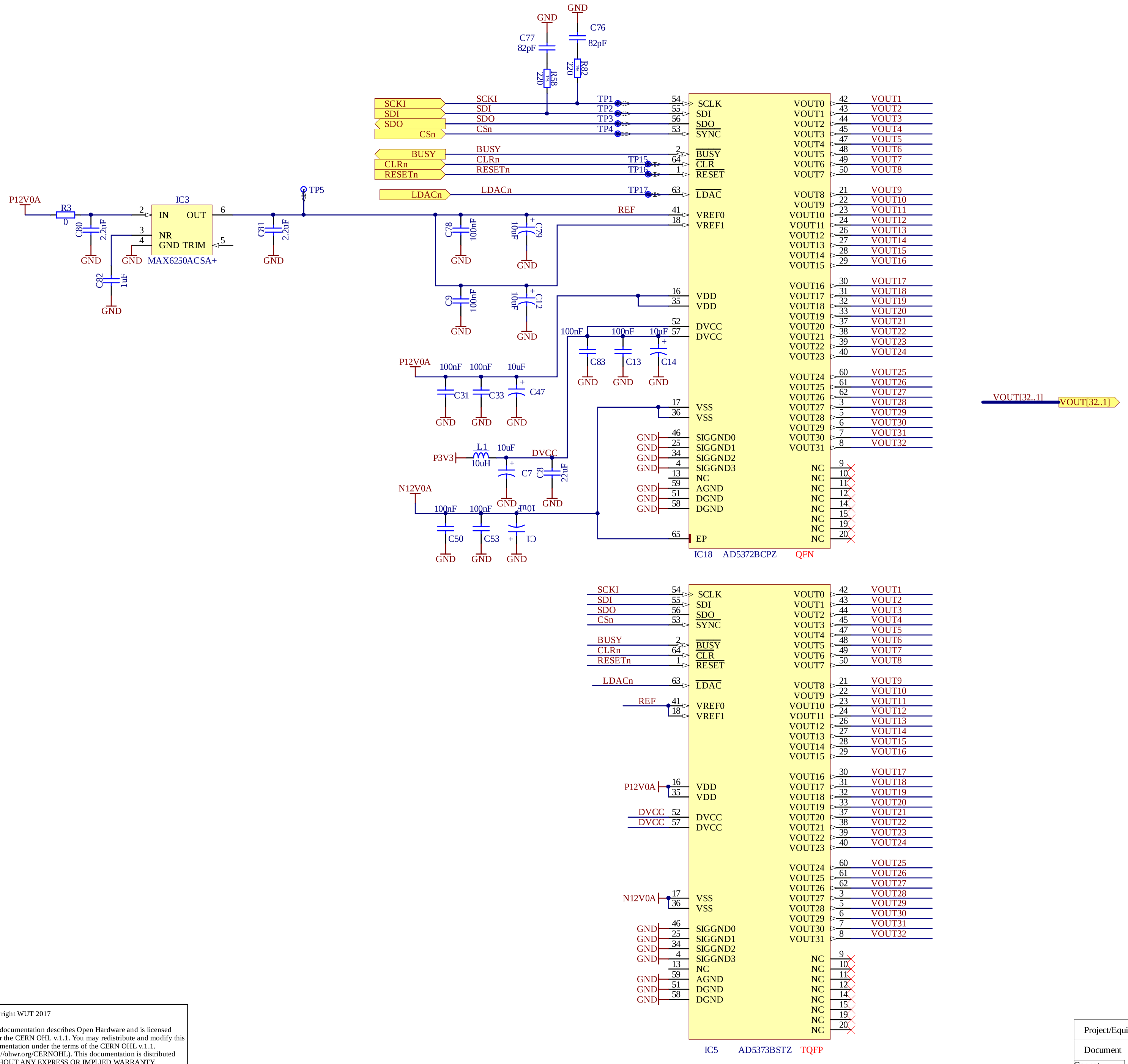
Cannot open file D:\Dropbox\DESIGN S\MTCA_projects\SI

Warsaw Univeristy of Technology ISE
Nowowiejska 15/19

Designer	G.K.	
Drawn by	G.K.	XX/XX/XXXX
Check.by		-
Last Mod.	-	15.08.2018
File	Zotino.schdoc	
Print Date	28.01.2020 03:39:52	Sheet 1 of 7

ARTIQ

Size A3 Rev -



Copyright WUT 2017

Project/Equipment Zotino

Document

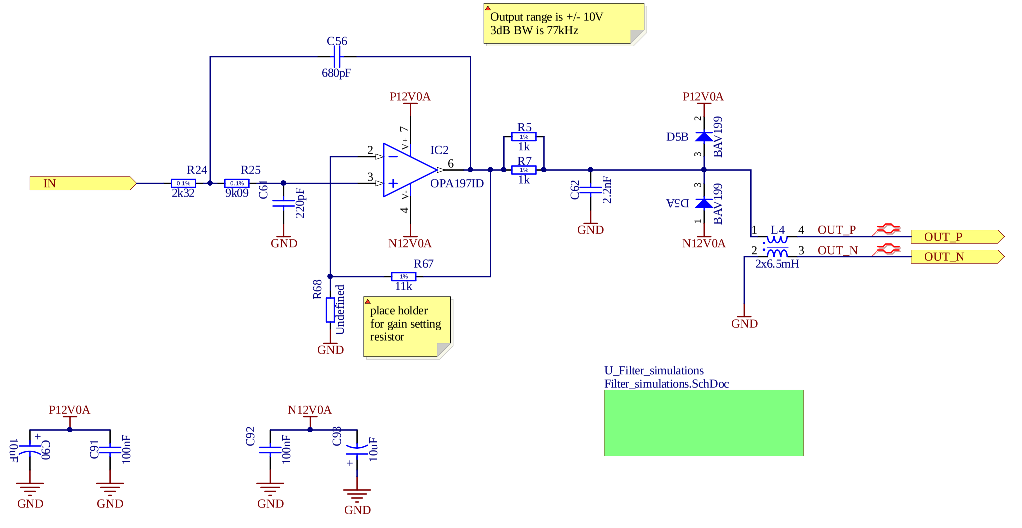
Cannot open file D:\Dropbox\DESIGNS\MTCA_projects\SI

DAC
& reference v1.2

Warsaw Univeristy of Technology ISE
Nowowiejska 15/19

Designer G.K.
Drawn by G.K. XX/XX/XXXX
Check.by -
Last Mod. - 15.08.2018
File DAC_32CH.SchDoc
Print Date 28.01.2020 03:39:53 Sheet 2 of 7

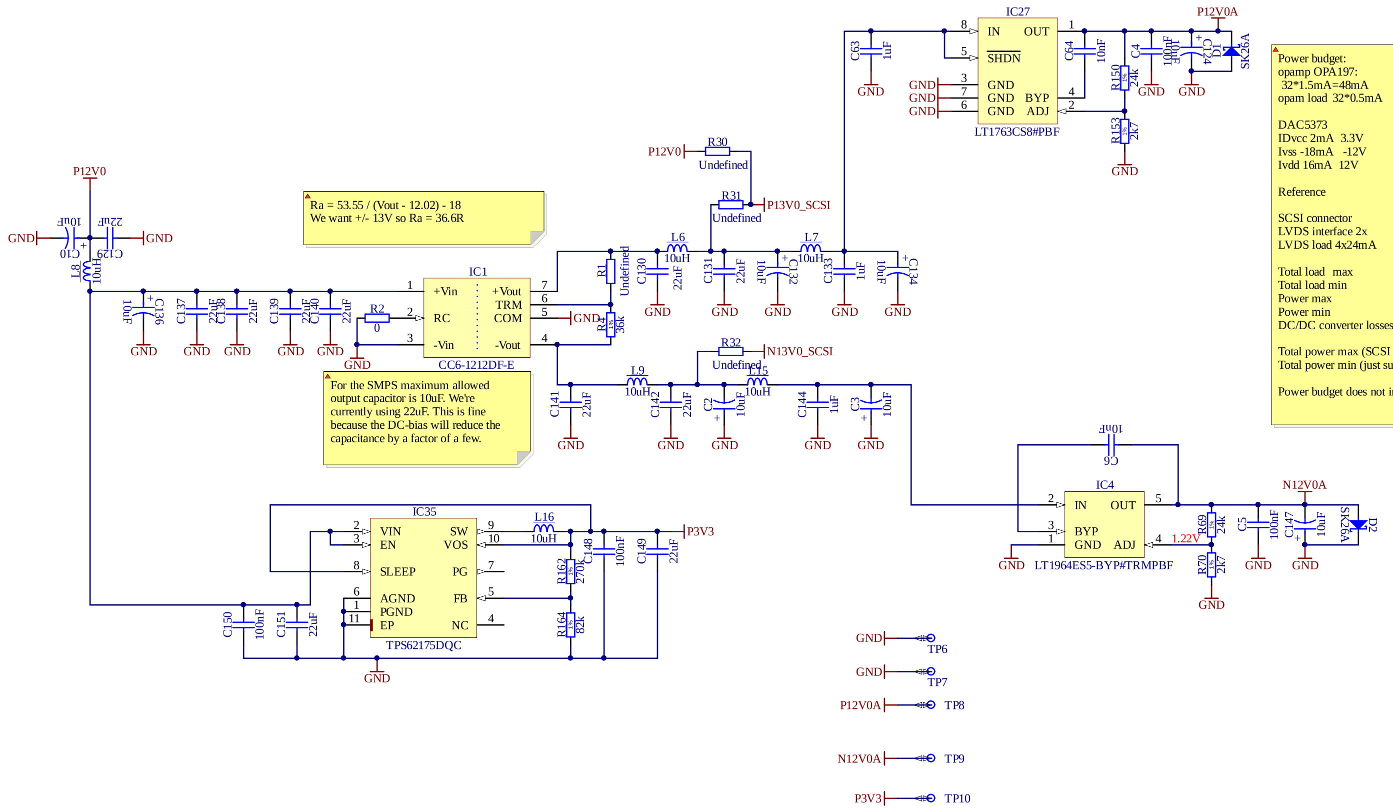
ARTIQ Size A3 Rev -



Output range is +/- 10V
3dB BW is 77kHz

place holder
for gain setting
resistor

Project/Equipment	Zotino		
Document	**Output filter v1.2**	Designer G.K.	
Cannot open file D:\Dropbox\DESIGNS\MTCA_projects\SI		Drawn by G.K.	XX/XX/XXXX
		Check.by	-
		Last Mod. -	15.08.2018
		File Output_channel.SchDoc	
		Print Date 28.01.2020 03:39:53	Sheet 4 of 7
Warsaw Univeristy of Technology *Nowowiejska 15/19*	***ISE***	ARTIQ	Size A3 Rev -



Ra = 53.55 / (Vout - 12.02) - 18
We want +/- 13V so Ra = 36.6R

For the SMPS maximum allowed output capacitor is 10uF. We're currently using 22uF. This is fine because the DC-bias will reduce the capacitance by a factor of a few.

Power budget:

opamp OPA197:	+13V rail	-13V rail	3.3V
32*1.5mA=48mA	48mA	48mA	
opam load 32*0.5mA	16mA	16mA	
DAC5373			
IDvcc 2mA 3.3V			2mA
Ivss -18mA -12V		18mA	
Ivdd 16mA 12V	16mA		
Reference	3mA		
SCSI connector	0mA	0mA	
LVDS interface 2x			330mA
LVDS load 4x24mA			96mA
Total load max	183mA	182mA	428mA
Total load min	64mA	63mA	250mA
Power max	2.3W	2.3W	1.4W
Power min	0.83	0.83	0.8
DC/DC converter losses	max: 0.92W; min:0.32		

Total power max (SCSI 2x 0mA + 0.5mA per out amp + 4 active LVDS outputs) 5.1W = 0.42A@12V
Total power min (just supply current, 2 active LVDS) 3.02W = 0.25A@12V

Power budget does not include TEC driver

Project/Equipment Zotino

Document

Cannot open file D:\Dropbox\DESIGNS\MTCA_projects\SI

Zotino
Power supply v1.2

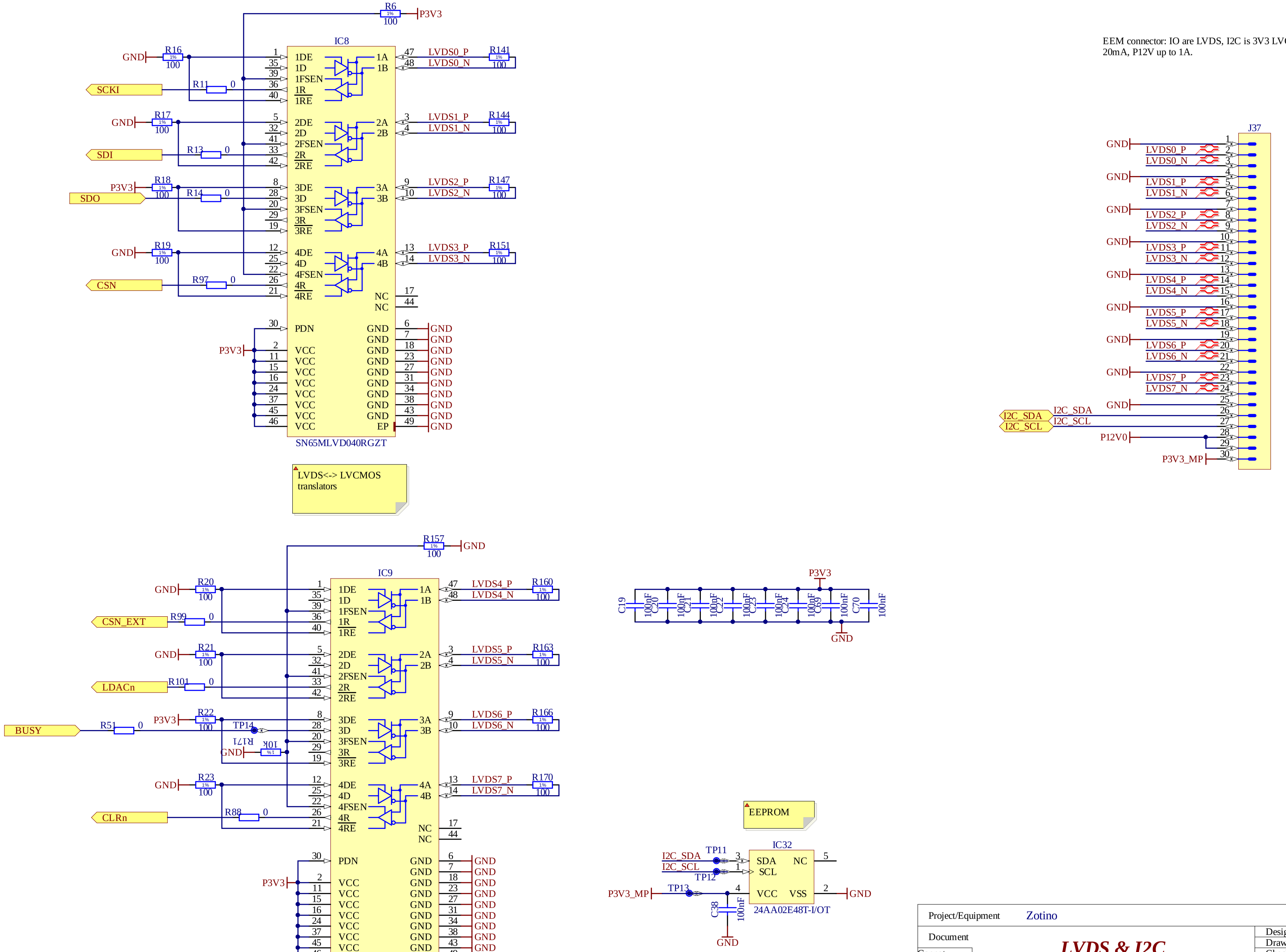
Warsaw Univeristy of Technology **ISE**
Nowowiejska 15/19

Designer	G.K.	
Drawn by	G.K.	XX/XX/XXXX
Check.by		-
Last Mod.	-	15.08.2018
File	Supply_DAC.SchDoc	
Print Date	28.01.2020 03:39:54	Sheet 5 of 7

ARTIQ

Size	Rev
A3	-

EEM connector: IO are LVDS, I2C is 3V3 LVCMOS, P3V3_MP up to 20mA, P12V up to 1A.



Copyright WUT 2017

This documentation describes Open Hardware and is licensed under the CERN OHL v.1.1. You may redistribute and modify this documentation under the terms of the CERN OHL v.1.1. (http://ohwr.org/CERNOHL). This documentation is distributed WITHOUT ANY EXPRESS OR IMPLIED WARRANTY, INCLUDING OF MERCHANTABILITY, SATISFACTORY QUALITY AND FITNESS FOR A PARTICULAR PURPOSE. Please see the CERN OHL v.1.1 for applicable conditions.

Project/Equipment	Zotino		
Document		Designer G.K.	
Cannot open file D:\Dropbox\DESIGNS\MTCA_projects\SI	**LVDS & I2C interfaces v1.2**	Drawn by G.K.	XX/XX/XXXX
		Check.by	-
		Last Mod. -	15.08.2018
		File LVDS_IFC_DAC.SchDoc	
		Print Date 28.01.2020 03:39:54	Sheet 7 of 7
Warsaw Univeristy of Technology Nowowiejska 15/19	***ISE***	ARTIQ	Size A3 Rev -

Appendix 2. Code used for tests and measurements

Listing Appendix 2.1: Code used for declaration of *Servo* module and all of its parameters

```
from migen import *
from migen.build.platforms.sinara import kasli
from artiq.gateware.szservo import servo
from artiq.gateware.szservo.pads import ZotinoPads, SamplerPads, pgiaPads
from .eem2 import *
# number of channels used for control
channels_no = 2
Kps = [0.1007 for i in range(channels_no)]
Kis = [839.19 for i in range(channels_no)]

plat = kasli.Platform(hw_rev="v1.1")
# numbers of EEM extentions to which particular boards are connected
sampler_conn = 3
sampler_aux = 2
zotino_conn = 4
# getting EEM extentions' and theirs subsignals' names
adc_io = Sampler.io(sampler_conn, sampler_aux)
dac_io = Zotino.io(zotino_conn)
# mapping EEM extensions' signals to physical pins
plat.add_extension(adc_io)
plat.add_extension(dac_io)
# extracting name used by other functions
adc_eem = adc_io[sampler_aux][0].split("_")[0]
pgia_eem = adc_io[2][0].split("_")[0]
dac_eem = dac_io[zotino_conn][0].split("_")[0]
# creating pads wrapper
adc_pads = SamplerPads(plat, adc_eem)
pgia_pads = pgiaPads(plat, pgia_eem)
dac_pads = ZotinoPads(plat, dac_eem)
# creating params used by each of the modules used by Servo
adc_p = servo.ADCParams(width=16, channels=channels_no, lanes=int(channels_no/2),
                    t_cnvh=4, t_conv=57 - 4, t_rtt=4 + 4)
iir_p  = servo.IIRWidths(state=25, coeff=18, adc=16, asf=16, word=16,
                    accu=48, shift=11, channel=3, profile=1)
dac_p = servo.DACParams(data_width = 24, clk_width = 2,
                    channels=adc_p.channels)
pgia_p = servo.PGIAParams(data_width = 16, clk_width = 2)
# initial values of PGIAs' gains - for every amplifier there are two bits of information; all of them
# are concatenated into 16-bits-wide vector
pgia_init_val = 0x0000
# creating an instance of a Servo class
m = servo.Servo(adc_pads, pgia_pads, dac_pads, adc_p, pgia_p, iir_p, dac_p,
                pgia_init_val, Kps, Kis)
m.submodules += adc_pads, pgia_pads, dac_pads
```

```
m.comb += m.start.eq(1)  # PID controller's start  pin driven constantly high
clk_signal = Signal()
clk125 = plat.request("clk125_gtp")
m.specials += [
    Instance("IBUFDS_GTE2", i_I = clk125.p, i_IB = clk125.n, o_O = clk_signal),
    Instance("BUFG", i_I = clk_signal, o_O = m.cd_sys.clk)
]
plat.build(m, run=True, build_dir = "building/pid/{}ch/pgia{:0>4x}/Kp_{}_Ki_{}".format(
    channels_no, pgia_init_val, Kps[0], Kis[0]), build_name = "top")
```

Listing Appendix 2.2: Code used for gaining access to corrseponding FPGA's peripheral signals and pins

```
from migen import *
from migen.build.generic_platform import *
from migen.genlib.io import DifferentialOutput
from artiq.gateware.szservo import pads as servo_pads
from artiq.gateware.szservo import servo

def _eem_signal(i):
    n = "d{}".format(i)
    if  i  == 0:
        n += "_cc"
    return n

def _eem_pin(eem, i, pol):
    return "eem{}:{}_{}".format(eem, _eem_signal(i), pol)

class  _EEM:
    @classmethod
    def add_extension (cls, target, eem, *args, **kwargs):
        name = cls.__name__
        target.platform.add_extension(cls.io(eem, *args, **kwargs))
        print("{}␣(EEM){})␣starting␣at␣RTIO␣channel␣{}"
                .fromat(name, eem, len(target.rtio_channels)))

class  Sampler(_EEM):
    @staticmethod
    def io(eem, eem_aux, iostandard = "LVDS_25"):
        ios  = [
            ("sampler{}_adc_spi_p".format(eem), 0,
                Subsignal("clk", Pins(_eem_pin(eem, 0, "p"))),
                Subsignal("miso", Pins(_eem_pin(eem, 1, "p"))),
                IOStandard(iostandard),
            ),
            ("sampler{}_adc_spi_n".format(eem), 0,
                Subsignal("clk", Pins(_eem_pin(eem, 0, "n"))),
                Subsignal("miso", Pins(_eem_pin(eem, 1, "n"))),
                IOStandard(iostandard),
            ),
            ("sampler{}_pgia_spi_p".format(eem), 0,
                Subsignal("clk", Pins(_eem_pin(eem, 4, "p"))),
                Subsignal("mosi", Pins(_eem_pin(eem, 5, "p"))),
                Subsignal("miso", Pins(_eem_pin(eem, 6, "p"))),
                Subsignal("cs_n", Pins(_eem_pin(eem, 7, "p"))),
                IOStandard(iostandard),
            ),
            ("sampler{}_pgia_spi_n".format(eem), 0,
                Subsignal("clk", Pins(_eem_pin(eem, 4, "n"))),
                Subsignal("mosi", Pins(_eem_pin(eem, 5, "n"))),
                Subsignal("miso", Pins(_eem_pin(eem, 6, "n"))),
                Subsignal("cs_n", Pins(_eem_pin(eem, 7, "n"))),
                IOStandard(iostandard),
            ),
        ] + [
            ("sampler{}_{}".format(eem, sig), 0,
                Subsignal("p", Pins(_eem_pin(j, i, "p"))),
                Subsignal("n", Pins(_eem_pin(j, i, "n"))),
                IOStandard(iostandard)
            ) for i, j, sig in [
                (2, eem, "sdr"),
                (3, eem, "cnv")
            ]
        ]
        if eem_aux is not None:
            ios += [
                ("sampler{}_adc_data_p".format(eem), 0,
                    Subsignal("clkout", Pins(_eem_pin(eem_aux, 0, "p"))),
                    Subsignal("sdoa", Pins(_eem_pin(eem_aux, 1, "p"))),
                    Subsignal("sdob", Pins(_eem_pin(eem_aux, 2, "p"))),
                    Subsignal("sdoc", Pins(_eem_pin(eem_aux, 3, "p"))),
                    Subsignal("sdod", Pins(_eem_pin(eem_aux, 4, "p"))),
                    Misc("DIFF_TERM=TRUE"),
                    IOStandard(iostandard),
                ),
                ("sampler{}_adc_data_n".format(eem), 0,
                    Subsignal("clkout", Pins(_eem_pin(eem_aux, 0, "n"))),
                    Subsignal("sdoa", Pins(_eem_pin(eem_aux, 1, "n"))),
                    Subsignal("sdob", Pins(_eem_pin(eem_aux, 2, "n"))),
                    Subsignal("sdoc", Pins(_eem_pin(eem_aux, 3, "n"))),
                    Subsignal("sdod", Pins(_eem_pin(eem_aux, 4, "n"))),
                    Misc("DIFF_TERM=TRUE"),
                    IOStandard(iostandard),
                ),
            ]
        return ios

class Zotino(_EEM):
    @staticmethod
```

```
def io(eem, iostandard="LVDS_25"):
    return [
        ("zotino{}_spi_p".format(eem), 0,
            Subsignal("clk", Pins(_eem_pin(eem, 0, "p"))),
            Subsignal("mosi", Pins(_eem_pin(eem, 1, "p"))),
            Subsignal("miso", Pins(_eem_pin(eem, 2, "p"))),
            Subsignal("cs_n", Pins(_eem_pin(eem, 3, "p"))),
            IOStandard(iostandard),
        ),
        ("zotino{}_spi_n".format(eem), 0,
            Subsignal("clk", Pins(_eem_pin(eem, 0, "n"))),
            Subsignal("mosi", Pins(_eem_pin(eem, 1, "n"))),
            Subsignal("miso", Pins(_eem_pin(eem, 2, "n"))),
            Subsignal("cs_n", Pins(_eem_pin(eem, 3, "n"))),
            IOStandard(iostandard),
        ),
    ] + [
        ("zotino{}_{}".format(eem, sig), 0,
            Subsignal("p", Pins(_eem_pin(j, i, "p"))),
            Subsignal("n", Pins(_eem_pin(j, i, "n"))),
            IOStandard(iostandard)
        ) for i, j, sig in [
            (5, eem, "ldac_n"),
            (6, eem, "busy"),
            (7, eem, "clr_n"),
        ]
    ]
```