KU LEUVEN

FACULTY OF ENGINEERING TECHNOLOGY

GROUP T LEUVEN CAMPUS

Versatile and Compact Control Unit & GUI Platform for Electroporation Devices

Bob VAN ELST

Promotor: Gert Van Loock

Co-promotor: Matej Reberšek (LBK) University of Ljubljana, Slovenia

Master Thesis submitted to obtain the degree of Master of Science in Engineering Technology: Electronics-ICT option Intelligent Electronics

Academic Year 2019 - 2020

Acknowledgements

I wish to express my sincere gratitude to Damijan Miklavcic for believing in my capabilities for the past three years and letting me be a part of something I can take pride in knowing that it will make a difference in the world, even if it is a small one. The big thanks for Matej Reberšek for guiding, and helping me in making this project come to fruition. Without him I would not have ventured as far as I have during all this. A big thanks to all the people at the Laboratory of Biocybernetics who have been kind and welcoming to me over the past three years. And lastly a big thanks to Lucija, without whom this project would not have become a reality.

Abstract

Commercially available electroporators that are heavy, immobile, and some require permanent installations. This creates high costs for maintenance, setup, and operation of the device. We proposed a new design that can be highly configurable and is capable of a wide spectrum of pulses that it can generate. The main targets for this device are research and educational institutions. The control unit of the electroporator was implemented in an FPGA and thanks to the utilization of the structured design method; we were able to achieve 10 nanosecond pulse resolutions. Along with being able to generate mono-polar pulses, we were also able to generate symmetric and asymmetric bi-polar pulses. By utilizing a System on Chip (SoC), we were able to combined two sub-modules of an electroporator that is normally separated, the control unit and the graphical user interface platform. With system modularity being a big requirement, the final outcome of the electroporator is a highly modular and compact system with four different types of generators incorporated into one.

Keywords: Control Unit, Electroporator, Embedded Linux, FPGA, GUI, VHDL

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

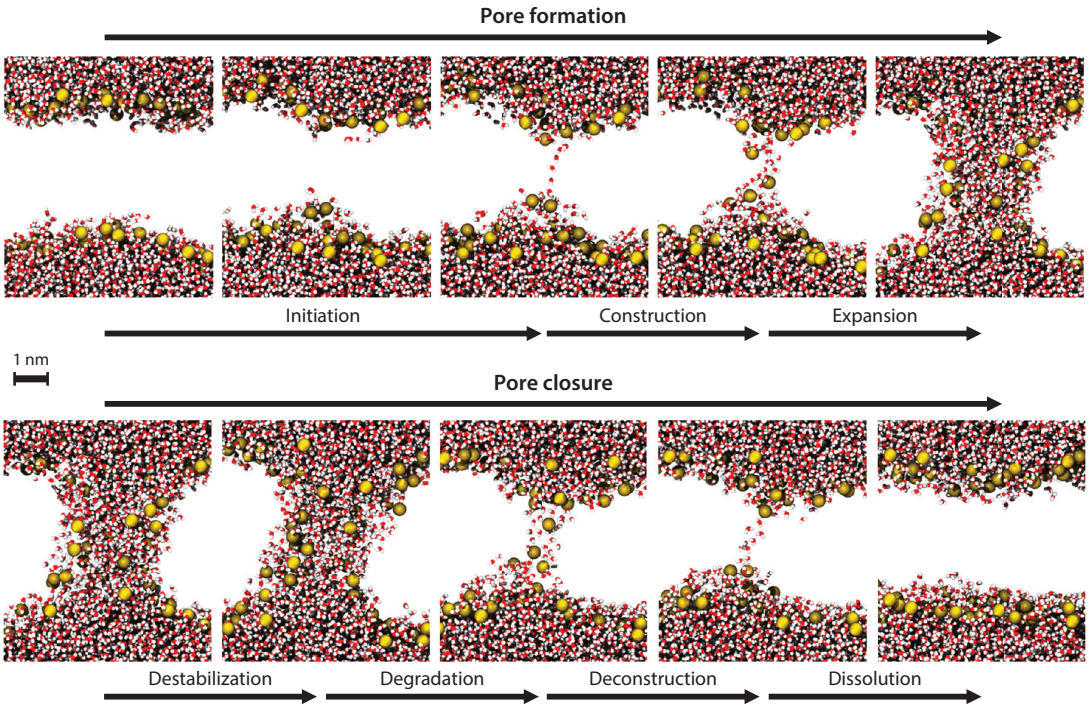

Figure 1.1: Process of pore formation and closure by means of electroporation taken from Kotnik et al. (2012)

Biological cells are the smallest structural and functional unit of living organisms according to Ellinger and Ellinger (2014). The human body alone contains approximately 37,2 trillion, Bianconi et al. (2013), while unicellular organisms contain only one. Every cell is enveloped by a cell membrane which is selectively permeable to ions and organic molecules in its surrounding. This selective permeability is used to regulate the transport of essential, redundant, and at times even harmful substances in and out of the cell, Miklavčič (2017). The survivability of the cell and its organism is dependant on this regulation. Due to the selective permeability of the cell, the effectiveness of certain biomedical and biotechnological treatments are reduced or in some instances rendered ineffective, Miklavčič (2017). To increase the success of these treatments, the cell's permeability would have to be increased to permit the entry or departure of molecules and ions that were previously restricted. There exists a few methods that are able to temporarily increase the cell's permeability, and electroporation is one of them, Kotnik et al. (2012).

1.1 Electroporation and its applications

Electroporation is a technique in which biological cells and tissues are exposed to a series of short electric pulses,Kotnik et al. (2012). With enough amplitude, the cells become permeable. Molecules and ions, which are normally unable to enter the cell, can now enter or exit thanks to the formation of pores, fig. 1.1, which are several nanometers in diameter on the cell membrane, Kotnik et al. (2012). If the cell is able to recover after electroporation it is considered reversible electroporation. In the event of the cell not being able to recover, due to sustained injury to the cell membrane or even destruction, it is considered irreversible electroporation.

The application of electroporation depend on these two phenomena, reversible and irreversible electroporation, Haberl et al. (2013). Electroporation pulses are delivered through electrodes where a pulsed electric field is generated between them and the cells are subject to this field, Miklavčič (2017). Within Biotechnology, electroporation is used for microbial deactivation, fig. 1.2, Haberl et al. (2013). The presence of bacterial, viral, unicellular organism pathogens, and their byproducts in our food or water represent serious threat to human health. According to Haberl et al. (2013), the preservation of foods using electroporation does not result in any form of hazardous by-products nor does it affect the colour, flavor, and antioxidant levels.

In medicine, electroporation is used to treat cancer patients in aiding drug delivery as is known as electrochemotherapy (ECT). Since most chemotherapeutic drugs act on dividing cells, they also affect the surrounding tissue. This side-effect is undesired and electroporation helps mitigate that, Haberl et al. (2013). Some chemotherapeutic drugs have weak membrane permeability, and in order to increase its effectiveness, a higher dosage is required. But there's a risk with higher dosages as side-affects become more pronounced, Haberl et al. (2013). With electroporation increasing local cell permeability, a high dosage is not required and the side-effects on surrounding tissue can be mitigated by only targeting cancer cells. This local anti-tumor treatment is done by first injecting an anticancer drug in the vicinity of the tumor cells and then applying electroporation to increase the permeability of the tumor, fig. 1.3. Once the drug has been delivered, it kills the cancer cell.

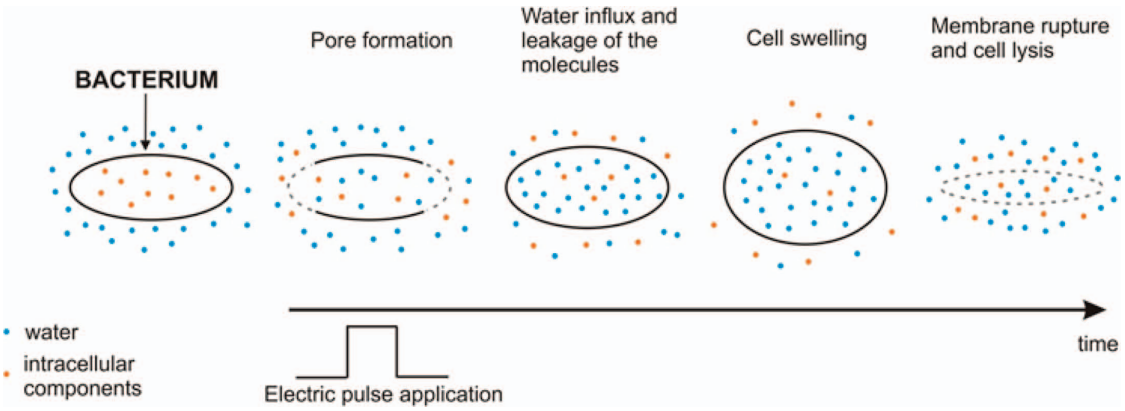

Figure 1.2: Electroporation application for microbial deactivation, taken from Haberl et al. (2013)

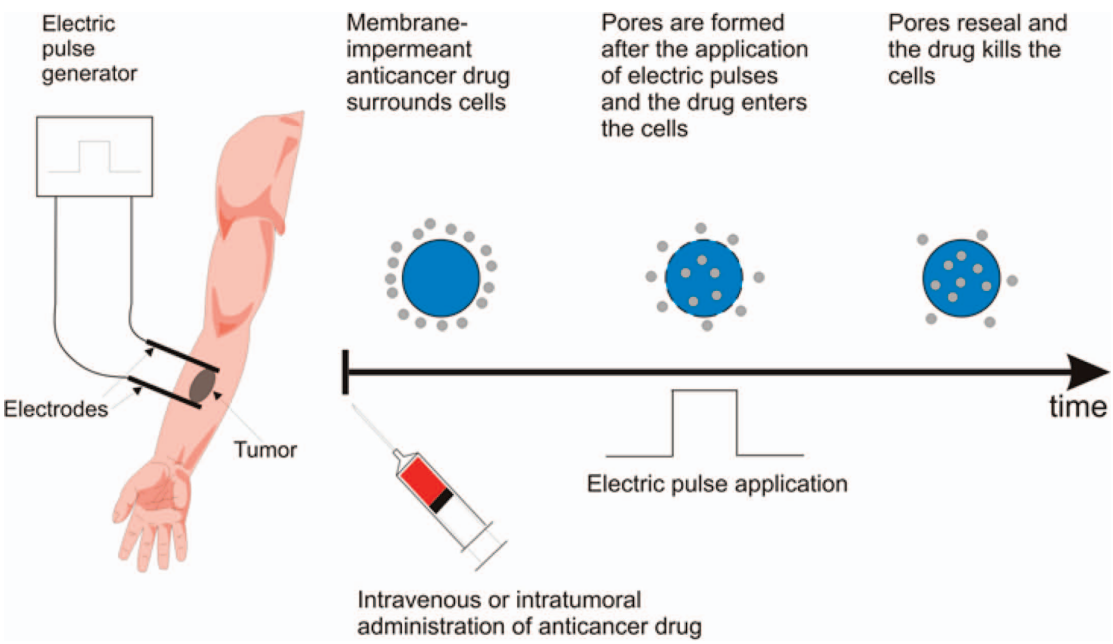

Figure 1.3: Electroporation application in Electrochemotherapy for drug delivery, taken from Haberl et al. (2013)

1.2 Electroporation Instruments

The instruments that perform electroporation are known as electroporators. Electroporators are able to generate pulses in varying sizes for electroporation. The size of electroporators vary depending on their applications. From food processing, medical, and to laboratorial research, each requires their own set of parameters specialized to their needs. Food processing requires a high a big voltage potential to accommodate the high volume of food being electroporated. Medical field requires electroporators to be build to certain safety standards. Laboratorial electroporators are required to be able to generate a wide range of pulses to cover the research spectrum. Most of the commercial electroporators on the market are propriety and closed source, limiting their availability to researchers and the research spectrum. However, there have been some cases of documented designs of electroporators. Bienkowski and Trzaska (2017) offers a full comprehensive list of commercially available electroporators and their advertised specifications.

1.3 Nature and scope of problem

Commercially available electroporators that are heavy, immobile, and some require permanent installations. This creates high costs for maintenance, setup, and operation of the device. Manufacturers of commercial electroporators incorporate predefined electroporation protocols that cannot be fine-tuned outside its given specifications. This limits the research spectrum of electroporation applications and studies. In some instances, the basic pulse parameters such as the pulse duration and amplitudes are not provided and the only information given is a predefined protocol boasting

of its performance, Pirc et al. (2017). Commercial devices are cost incentive driven, which results in limited availability to researchers and educational institutions such as universities.

1.4 Aims

The aim of this paper is to provide an alternative option to commercially available electroporators for research and educational institutions. The electroporator should be able to cover a wide range of electroporation applications with full control over the pulse parameters to the user. The end goal of this electroporator is to be an open-source standard for high performance electroporators that can deliver a wide spectrum of pulses that the current commercial electroporators on the market cannot. A small compact size, high modularity, high customizability, and high speed performance is expected.

Chapter 2

Requirements Analysis

The Electroporator will consist of a Control Unit implemented on an Field Gate Programmable Array (FPGA) that will generate trigger pulses that turn the high voltages on and off by means of a high voltage switch at very high speeds. The pulse parameters will be inputted by the operator by a means of a Graphical User Interface running on an Embedded Linux software platform, which runs on the ARM architecture. The Graphical User Interface should be possible to be implemented using the Qt5 cross-platform application framework and cross-compiled from a host machine running Linux. The communication between the Control Unit and Embedded Linux will be through the use of an existing communication protocol standard and the design will focus on system stability, security, and safety. Overall system design will be oriented towards clinical and laboratorial applications.

2.1 Electronic Requirements

The following are minimum requirement specification for the electronics:

1. Timing sensitive aspects of the design will be implemented on an Field-Programmable Gate Array (FPGA)
2. System Outputs have to be capable of triggering H-Bridge switching configurations.
3. System Outputs have to be capable of triggering boolean logic switching configurations
4. Utilizing AXI protocol for memory mapped communications from the User Space.
5. Utilizing Master SPI controller.

2.2 Function Requirements

The following are the minimum required functional capabilities:

1. Able to generate 100 nanosecond pulses with 10 nanosecond pulse resolution.
2. Able to generate Mono-polar and Bi-polar pulses.
3. Able to generate symmetric and asymmetric Bi-polar pulses
4. Burst mode functionality, (multiple repetitions of a given pulse function).
5. Gentle Shutdown system for In Case of Emergency (ICE).

2.3 Software Platform Requirements

The Software Platform needs to be on an embedded linux operating system that is running on the ARM architecture. Qt binaries have to be integrated in the system and have cross-compilation capabilities for quick development and deployment.

Chapter 3

Design

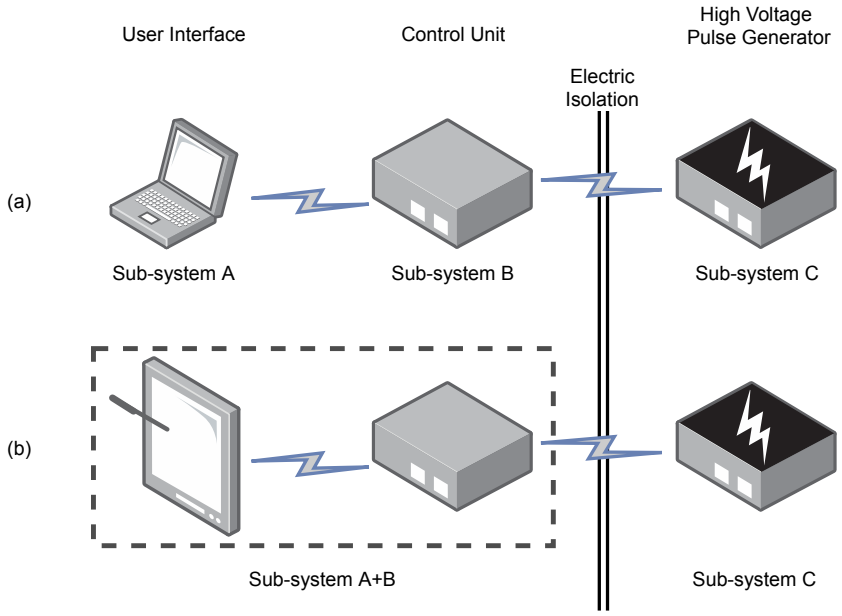

Figure 3.1: (a) Traditional Electroporator system design versus (b) new compact design

Traditional Electroporator system designs, fig. 3.1 (a), consists of three main parts, the User Interface (A), Control Unit (B), and the High Voltage Pulse Generator (C). These three parts are generally three different sub-systems that run on different hardware. But the common point between all the three sub-systems was that they were physically isolated from each other, and in the case of sub-system C also electrically isolated. Traditional designs suffered from portability and high-costs. This resulted in large machines that had limited mobility and a high price tag which limited the availability of researches and patients to their use. The new proposed design tries to mitigate those shortcomings by reducing its size and price thanks to more affordable technologies. This is achievable by combining the two sub-systems A and B together, fig. 3.1 (b), for a more compact and economical design by utilizing System on Chip (SoC) hardware. Aside from reducing the size and cost of the system, we aim to provide a better user experience by cleaning up the peripherals and utilizing a touch-screen interface for the configuration of the system.

3.1 Design of the Control Unit for Electroporation Pulse Generation

The core part of an Electroporator is its control unit. The main function of this part, as the name suggests, is to control the generation of pulses. It does this by generating trigger pulses that turn

the high voltage on and off by means of a high voltage switch. The implementation of the trigger can vary based on the type of generator is being created, mono-polar or bi-polar, and the number of electrodes. Since electroporation pulse repetition rate can range anywhere between 0,1Hz to a few MHz, Miklavčič (2010), the timing of our trigger pulses have to be very precise to enable generation of 100 nanosecond pulses with 10 nanosecond accuracy in order to meet our target requirements. The previous iterations of the Control Units for Electroporators from the Laboratory of Bio-cybernetics in the University of Ljubljana, Slovenia, were developed on an FPGA, as well as other Electroporators on the market. Accroding to Xilinx (2019b), "Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects". A good comparison to FPGA's would be microcontrollers as they can both be programmed to a specific design but their implementation and interpretation of said design vastly differs. The former implements the design in hardware with their use of CLB's, while the latter reads the program as a set of instructions and executes them. The main benefits of an FPGA over a microcontroller is its parallelism speed and accuracy. With an FPGA you can achieve higher speeds with accurate results in comparison to microcontrollers. While some Electroporators have been developed on microcontrollers, they are only capable around the 10kHz range and have a considerable amount of error in their precision. Our device has to be capable of at least a 10ns (nanoseconds) accuracy, Mega-Hertz (MHz) range, and thus the use of an FPGA has become the optimal choice for our Electroporator.

3.1.1 Field Gate Programmable Gate Array (FPGA) Selection

The selection of an FPGA manufacturers came down to a matter of personal preference and experience as the top two FPGA vendors in the world, Xilinx and Altera/Intel, have similar competing architectures but what differs among them are the features they provide and their development environment. The manufacturer of choice was Xilinx as we had prior experience working with their FPGA's as well as their development environment. We opted for Xilinx's Zynq architecture System on Chip's (SoC) with integrated FPGA Fabric and ARM Hard Cores which are labeled Programmable Logic (PL) and the Processing System (PS), respectively. The main purpose of selecting an SOC for our system was to combine the User Interface with the Control Unit, as shown in figure 3.1 (b), and eliminate the physical link between them as it is a potential security and safety risk. The control unit was implemented in the PL while the GUI was deployed on the PS.

3.1.2 Pulse Generator Design

The Pulse Generator resides inside the Control Unit and it is responsible to generate the trigger pulses that turn the high voltage power banks on and off. The functionality of this part is similar to a square wave function generator. In order to generate a trigger pulse, the user will have to input pulse parameters such as the duration of the pulse, the number of pulses, and the pulse repetition rate. The pulse parameters will differ depending on the type of pulses they wish to generate, mono-polar or bi-polar. Mono-polar pulses, as it suggests, are pulses of a single polarity. On the end of the dual electrodes, one will be electrically positive and the other negative. For bi-polar pulses, both polarities are used. During generation of pulses, at a given time, the polarity of the positive and negative electrodes will switch, hence giving us bi-polar pulses. In terms of complexity, the mono-polar pulse generator is the simplest to implement.

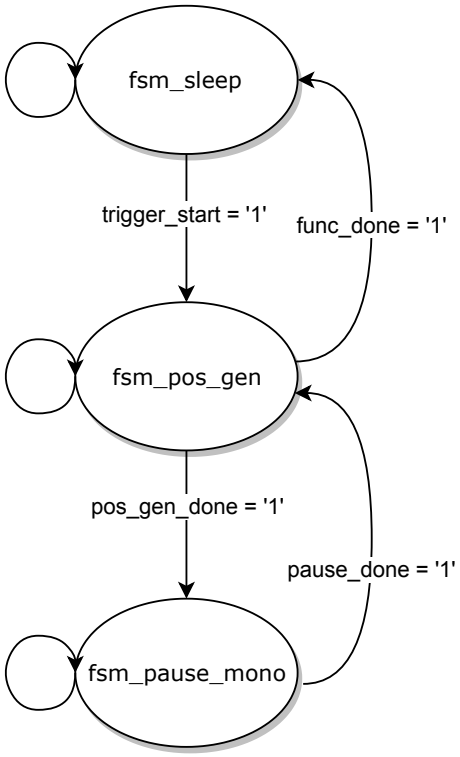

Figure 3.2: Finite State Machine of a simple Mono-polar Pulse Generator

3.1.2.1 Finte State Machine (FSM)

In order to implement a complex function in hardware, it is typically broken down into a set of states and rules. In digital electronics this is often referred to as a Finite State Machine (FSM). A state is a specific operation mode that the system is instructed to execute, and the flow of one state to another is governed by a set of rules. Figure 3.2, shows a simple implementation of a mono-polar pulse generator. Initially, the system is in state `fsm_sleep` and continues to stay in this state until the condition of `trigger_start` becomes true. Afterwards, the system starts generating positive pulses as it stays in `fsm_pos_gen`. The following state can be either back to `fsm_sleep` or continue on to `fsm_pause_mono` depending on the number of pulses you initially wanted to generate.

In accordance with the system requirements, the Control Unit had to be able to produce four different types of pulses. The four different types of pulses are simple mono-polar, full mono-polar, simple bi-polar, and full bi-polar. The key differences among them is the number of states required for each type of pulse generation. Simple types are the most basic, requiring only one state for positive/negative pulses, while the full types require three. This is because the extra states before and after the pulse, preparation and discharge respectively, yield better timing results for high voltage pulses by factoring in the rise and fall time of the pulse delivery, Miklavčič (2010) and Rebersek et al. (2014). A full list of all the states that will be utilized are shown in listing 3.1.

```
type state_type is (
    fsm_sleep, --Do nothing here
    fsm_pos_prep, --Charge for pulsing (positive)
    fsm_pos_gen, --Start pulsing (positive)
    fsm_pos_dis, --Discharge pulsing (positive)
    fsm_pause_mono, --Pause after positive pulsing
    fsm_neg_prep, --Charge for pulsing (negative)
    fsm_neg_gen, --Start pulsing (negative)
    fsm_neg_dis, --Discharge pulsing (negative)
    fsm_pause_bipol, --Pause after negative pulsing
    fsm_pause_burst, --Pause between two pulse functions
    fsm_ICE --In Case of Emergency state, discharge everything
);
```

Listing 3.1: All the states of the Pulse Generator in VHDL

3.1.2.2 Inputs and Outputs

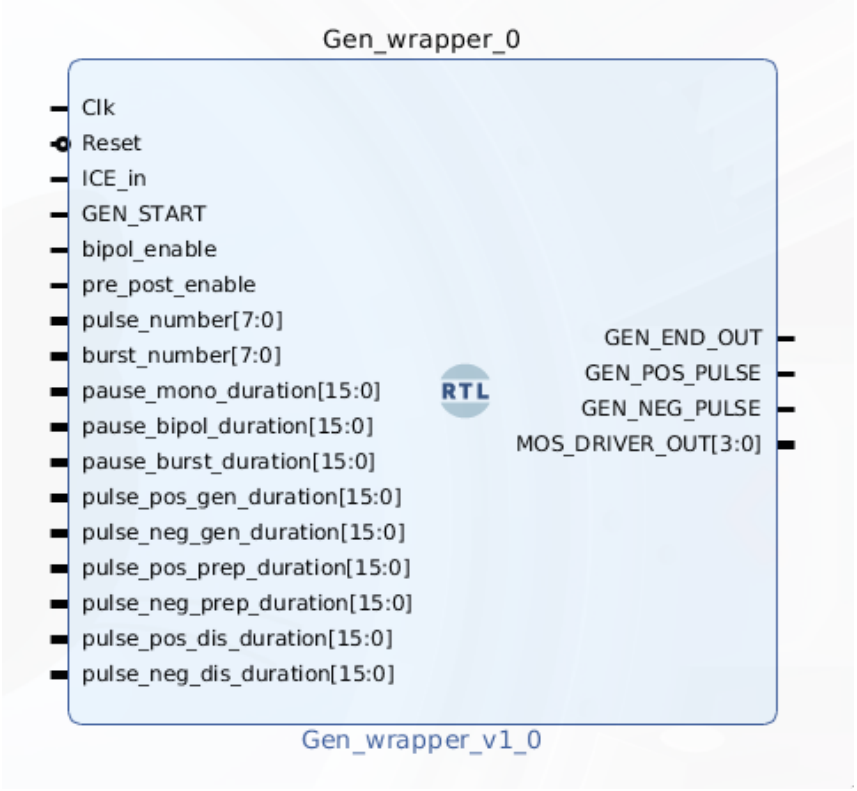

Figure 3.3: Pulse Generator inputs and outputs

The Pulse Generator requires a set of inputs which are used for configuring the type of pulses the operator wishes to generate. Parameters such as the pulse duration, repetition rate, and the number of pulses are required as the inputs as well as some control values for selecting the type of pulses. With keeping the requirements in mind of versatility, we opted for the operator to be able to configure as much as possible of the pulse function that is to be generated. This meant that the duration of each state depends on the user inputs. Figure 3.3 displays all the inputs for

the Pulse Generator. Mono-polar and bi-polar pulses are enabled by `bipol_enable`, and the simple or full types by `pre_post_enable`. The number of pulses and bursts by `pulse_number` and `burst_number`, respectively. The `*_duration` inputs are for configuring the duration of the corresponding states, and `GEN_START` is the trigger signal for the Pulse Generator to start running. `ICE_in` stands for "In Case of Emergency", which the functionality will be discussed during the safety section.

The Pulse Generator contains only four outputs, as seen on figure 3.3. For mono-polar pulses we have `GEN_POS_PULSE`, and bi-polar pulses we have an extra output `GEN_NEG_PULSE`. The last output, `MOS_DRIVER_OUT`, is specific for triggering H-Bridge generators. The vector output correspond with the MOSFET driver pins of a H-Bridge high voltage pulse generator. The first output, `GEN_END_OUT` is a control output that can be used to observe if the generator is running. This can be observed in the timing diagram, fig. 3.4. The specific values per state of `MOS_DRIVER_OUT` depends on the initial configuration of the whole Control Unit, it is not something that can be changed by inputs as it is hard-coded/wired in the FPGA.

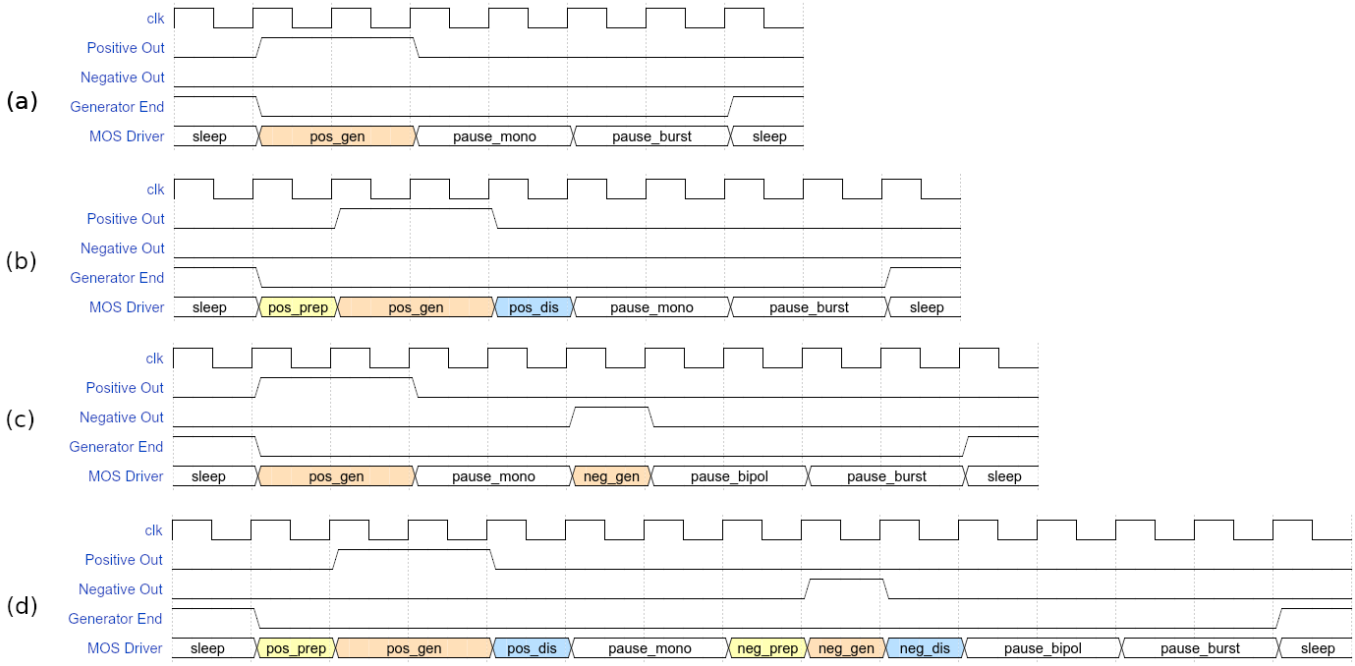

Figure 3.4: Example timing Diagram of all four Generators in the Control Unit. (a) Simple mono-polar, (b) full mono-polar, (c) simple bi-polar, and (d) full bi-polar.

3.1.2.3 Hardware Description Language (HDL)

To program on an FPGA, we needed to use a Hardware Description Language (HDL). It is a lot different than traditional software programming languages, where they are used to instruct the computer to perform a certain task, while HDL's are used to describe the behavior and structure of a digital circuit. The mindset needed to write HDL differs greatly from software programming as the former is based on a great deal of parallelism. For our Control Unit, we opted for VHDL, (Very High Speed Integrated Circuit Hardware Description Language), as it is a strongly typed language and

more often used in the medical field, Xilinx (nd). Strongly typed languages have the advantage with the early detection of errors, which in turn speeds up development. It also helped that we had prior knowledge and experience of VHDL over the other HDL's such as Verilog and SystemVerilog.

3.1.2.4 Design Methodology

Previous Electroporators developed by the Laboratory of Bio-cybernetics in the University of Ljubljana, Slovenia, used traditional dataflow coding model. Our first attempt at build the Control Unit was done using this style along with the designs of previous Electroporators. Since the older models were already at their peak it was difficult to push the design to much tighter timing constraints of 10 nanoseconds. There came a point in the first prototype where it passed all simulation testing but failed during implementation on hardware. The design consisted of over 20 parallel processes which made debugging challenging. It was soon realized that the dataflow method provided no efforts in making the system modular and versatile as per our requirements. It also made code re-usability difficult as each process depended on another for their functionality and was useless when standalone.

	Two-process method	**Dataflow coding**
Adding ports	• Add field in interface record type	• Add port in entity declaration • Add port to sensitivity list (input) • Add port in component declaration • Add signal to port map of component • Add definition of signal in parent
Adding registers	• Add field in register record type	• Add two signal declaration (d & q) • Add q-signal in sensitivity list • Add driving signal in comb. process • Add driving statement in seq. process
Debugging	• Put a breakpoint on first line of combination process and step forward • New signal values visible in local variable v	• Analyze how the signal(s) of interest are generated • Put a breakpoint on each process or concurrent statment in the path • New signal value not immediately visible
Tracing	• Trace the r-signal (state) • Automatic propagation of added or deleted record elemenets	• Find all signals that are used to implement registers • Trace all found signals • Re-iterate after each added or deleted signal

Table 3.1: Dataflow vs. two-process comparison from Gaisler (2011)

After many iterations, it was concluded that starting over using a new design methodology was the next step. Having consulted some experts in this field, we were forwarded to Jiri Gaisler's Two Process Design Methodology, or commonly known as, Structured VHDL Design Method, Gaisler (2011). Originally this design method was intend for fault-tolerant space applications, primarily used by the European Space Agency ESA (2019), and this reassured us that it was a suitable method for medical applications as the medical field lacked any kind of standard design method for FPGA's. This method utilizes only two process and works by using serial execution of statements instead of the parallel flow used in dataflow coding, Gaisler (2011). A full comparison of the versatility of

the two process method can be found in Gaisler's paper and on table 3.1. The application of this method had clear advantages over the dataflow method, such as code readability, fast simulation and synthesis runs, and a high degree of code reusability. To validate this, the new generator synthesis designs took sub-30 seconds to finish, as opposed to the five minute runs with the dataflow method. Another benefit of the two process method was the utilization of records and packages from the VHDL-2008 standard, Lewis and Training (nd). VHDL-2008 packages are comparable to header files in C/C++ where you can define the value of a specific macro before compilation, and records to C/C++ structs where it was used for port definitions and signals. Records were used to organize the ports and signals, while packages containing records acted like "interfaces" that could be imported in each module. The two process method can be observed on figure 3.5. The inputs and outputs of the system go into the combinational process and get clocked in the sequential process. The expected latency of this design is two clock pulses if using an FSM.

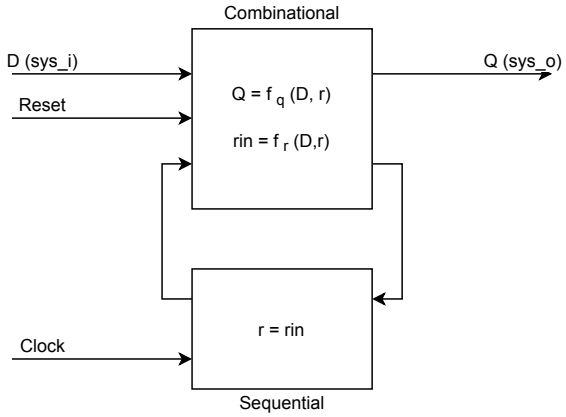

Figure 3.5: Generic two process circuit, Gaisler (2011)

3.1.3 Reducing Complexity By Embracing Simplicity

The first iteration using the new design method led us to create a universal generator that worked in simulation as well as hardware. However, since all four types of generators were included in the universal generator, it made code reusability difficult as the design had high complexity due to a very large FSM. Aside from the large size, our implementation wasn't clean, we had "if" statements nested to the third degree, which is frowned upon in FPGA designs as that can cause heavy latency by creating large sequential statements. To mitigate this, we opted to spilt apart the generator and focus on each individual generator types. This lead us to having a design of four separate generators inside the universal generator. The result can be observed in figure 3.6 (c), where only one generator is selected for the duration of the pulse function. The splitting up of the generators as well as their external outputs, fig. 3.6(d)(e), increased code readability while also creating small modules that when combined are able to create a complex system. These modules can also work standalone, as a single generator can be picked out and applied in another design. The simplicity of the design greatly improves debugging as we could isolate individual modules for testing.

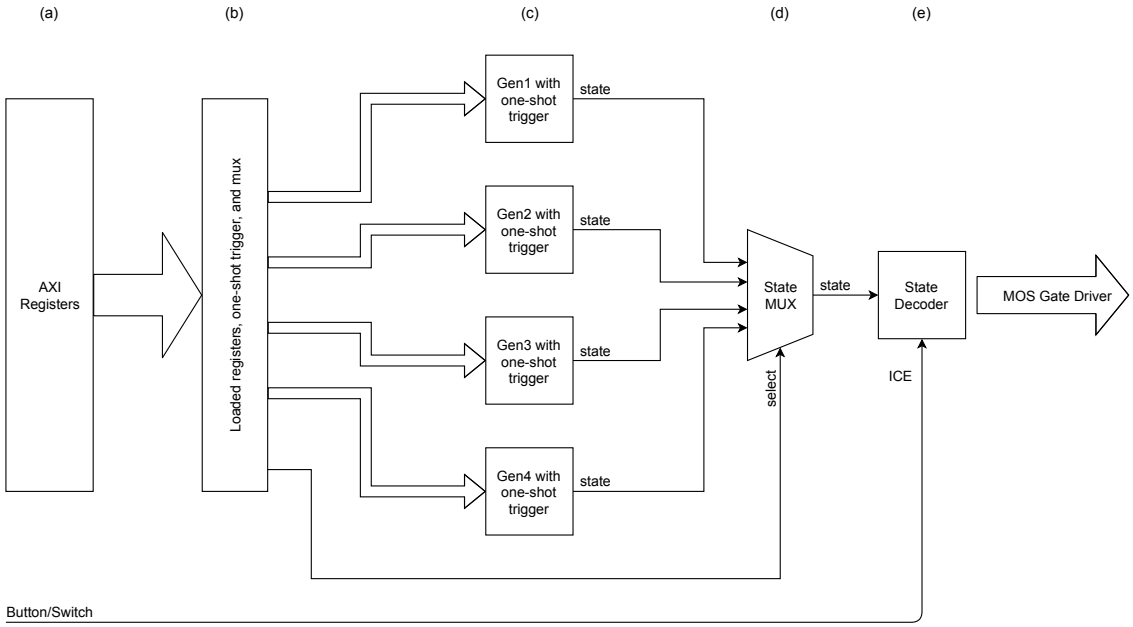

Figure 3.6: Universal Control Unit for Electroporation Pulse Generation containing the (a) AXI Registers, (b) latched registers, (c) four generators (two mono-polar and two bi-polar), (d) state MUX, and the (e) state decoder.

3.1.4 Acquiring Electroporation Pulse Parameters

For the Pulse Generator in the Control Unit to generate custom trigger pulses, it requires the pulse parameters given by the operator. Since the pulse parameters will be inputted by the user through a graphical interface, we needed a way to store these values and be able to read them on the Programmable Logic (PL), FPGA Fabric. This requires communication between the Programmable Logic (PL) and the Processing System (PS), where the user interface is located. In Xilinx's Zynq Architecture, there exists a standardized protocol of communication between the PS and PL, the AXI Interface. AXI, or Advanced eXtensible Interface protocol is a part of the ARM AMBA,a family of micro-controller buses first introduced in 1996, Xilinx (2017). AXI is made for memory-mapped interfaces where the the parameters will be written to from the PS and the PL will read that portion of memory. This requires a module that utilizes the AXI Interface and stores the values in registers for the Pulse Generator to use.

While the AXI reference guide is well documented, we opted to use a tool to generate the AXI-Registers than to make our own to speed up development. The tool in question was by noasic GmbH (2019), `AirHDL.com`. Using this tool we were able to make a module specific to our design constraints that interfaces as an AXI Slave module. By simply making a memory mapped layout of our parameters and control signals the tool generates a synthesizable code that we can import into Xilinx's design tool, Vivado. The memory mapped layout can be observed on figure 3.7, and the layout of the `SETUP` register on figure 3.8. The AXI-Registers are positioned at the beginning of the Control Unit, fig. 3.6(a), where they can input the pulse parameters into the Pulse Generator.

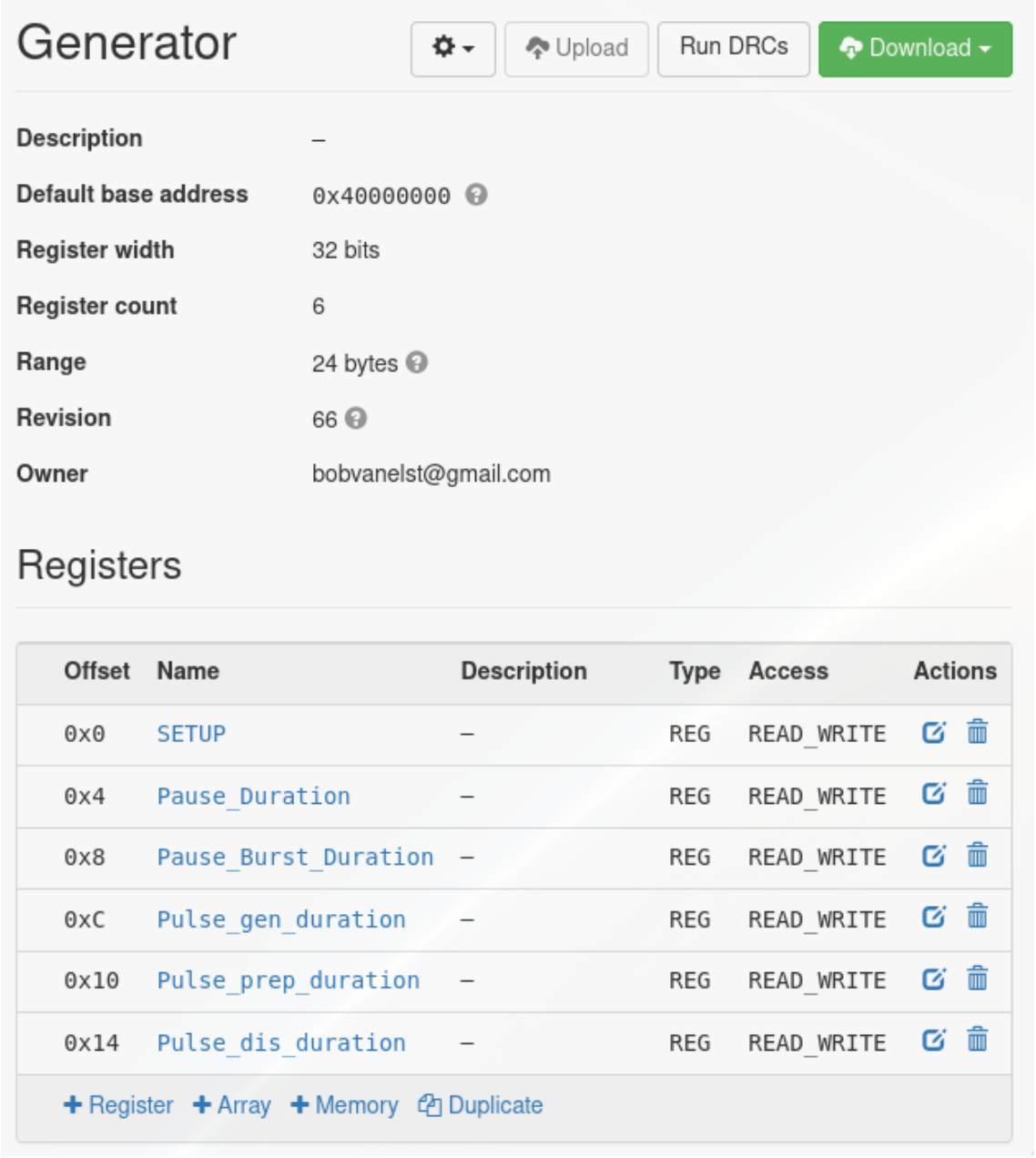

Figure 3.7: AXI-Registers memory map using AirHDL

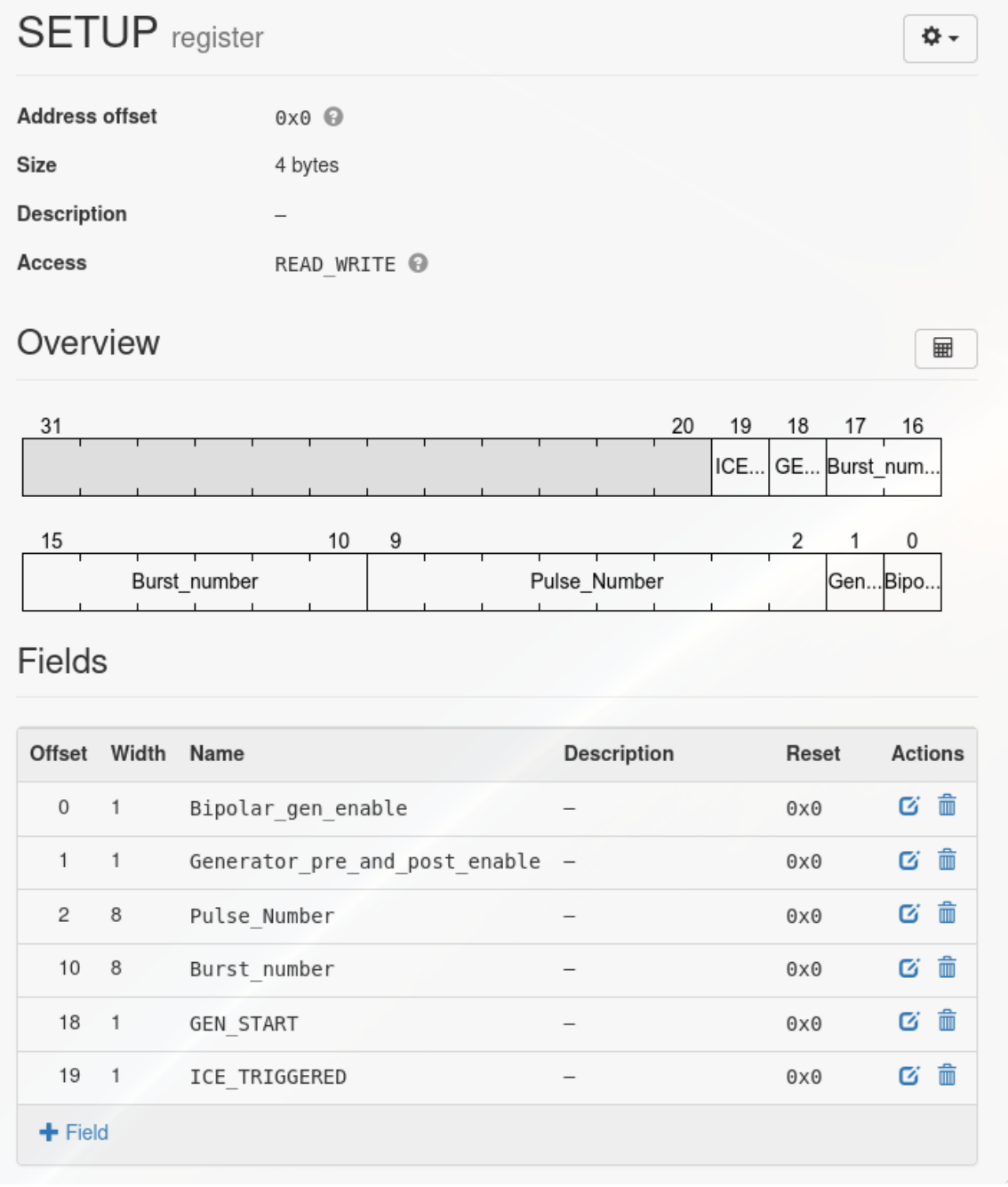

Figure 3.8: Layout of SETUP register in AirHDL

3.1.5 System Stability, Security, and Safety

For the versatility of the Control Unit, we had to make the FSM of the Pulse Generators dynamic. Meaning, the duration of each pulse was not hard coded in but was variable. However, this can be potentially dangerous if the design is not implemented correctly. If the parameters were to suddenly change while the generator was in operation, it would cause the generator to continue running based on the new parameters. This situation is dangerous because if the Control Unit was initially running Reversible Electroporation pulses it can suddenly switch to Irreversible Electroporation and destroy the cells between the electrodes, or even cause harm to the patient. To prevent this, we have to make sure that the pulse parameters don't change while the Pulse Generator is in operation. This is done by adding a buffer layer between the AXI-Registers and the Pulse Generators, fig. 3.6(b). This buffer layer will load the pulse parameters from the AXI-Registers and hold them for the entirety of the pulse generation. The loaded values won't change until the pulse function is completed or a system reset has been issued.

When the Control Unit is in operation, only one generator should be active. Our implementation of this was to use a multiplexer (MUX) on the outputs of all the individual generators to make sure that only one is forwarded to the `State Decoder`, fig. 3.6(d). The select input of the `State MUX` behaves the same way as the input parameters to the generators, it does not change during pulse generation. Aside from adding a MUX at the outputs of the generators, we also added them to the inputs of the generators. Between the Loading Registers and the Generators lies a MUX that forward the pulse parameters to the selected generator. This meant that only one generator gets the pulse parameters while the others are set to a reset state.

For the Pulse Generator to start generating, it requires a trigger pulse which comes from the operator, usually by means of an analogue or digital button, to the AXI-Registers. Due to the design of the FSM, that trigger signal is only read during its `fsm_sleep` state and ignored in others. When the Pulse Generator is done, it comes back to the sleep state, fig. 3.4, and checks the trigger pulse again. If the trigger is active, the generator would start the pulsing sequence again. This is the case of a runaway system where it will continue to generate pulses indefinitely as long as the trigger is active. It's not a behavior that is desired. Especially in the case of pulses in the sub 100 nanosecond range, if the operator cannot disengage the trigger in less than 100 nanoseconds, it results in several pulse function repetitions. To overcome this, we implemented a one-shot trigger system between the Load Registers and the Pulse Generators. A one-shot trigger system resembles the functionality of a Monostable Multivibrator MOSFET circuit but described in HDL, listing 3.2. It's function is to generate an output signal with a fixed width regardless of the input length, minimum length of one clock pulse. In our system, the input will be the trigger signal from the AXI-Registers and the output will be the modified one-shot trigger of two clock pulses for the Pulse Generators. An added benefit of having the one-shot trigger near the Loading Registers is that we were able pipe the output into it and have it trigger the loading of the pulse parameters. Individual one-shot trigger systems are included inside each of the generators and while this is redundant, as we have one preceding it already, if the generators were to be standalone modules they would require this one-shot trigger to prevent a runaway system.

```
 signal r, rin : reg_type := init;

begin
--Combinational process
  comb: process(sys_i, r, reset)
  variable v : reg_type;
  begin
    v := r;
  --------------------------------------------------------------------------------
    --One-shot trigger
    if r.trigger_flag = '0' AND sys_i.GEN_START = '1' then
      v.trigger_start := '1';
    else
      v.trigger_start := '0';
    end if;
  --------------------------------------------------------------------------------
    --latch/load the values

    if r.trigger_start = '1' then --load values
      v.latched_outputs := sys_i;
      v.latched_outputs.GEN_START := '1'; --for redundancy in case GEN_START
    goes low very quickly.
    else
      v.latched_outputs := v.latched_outputs; --hold values
      v.latched_outputs.GEN_START := '0'; --need this for consecutive
    Generations.
    end if;
  --------------------------------------------------------------------------------

  --------------------------------------------------------------------------------
  --------------------------------------------------------------------------------
    --save variables
    rin <= v;
  end process;

--Sequential process
  reg: process(clk)
  begin
    if rising_edge(clk) then
      r <= rin; --load new values
      r.trigger_flag <= sys_i.GEN_START; --update the init trigger
    end if;
  end process;
```

Listing 3.2: Code Snippet of One-Shot trigger for Loading Pulse Parameters in VHDL

Aside from having internal safety features, the Control Unit also required an external input called ICE. ICE stands for In Case of Emergency, and is often used by operators during Electroporation in the events of a disconnected electrode, misconfiguration of pulses, or other similar episodes to disconnect the high voltage power banks. Preceding Electroporator designs routed this signal through the FSM of the Generator, which caused some latency in the response time of the emergency. Our solution was to route it right to the end of the output, bypassing the Pulse Generators, to the `State Decoder`, fig. 3.6(e). By fixing the the State Decoder to the state `fsm_ICE`, listing 3.1, we are able to initiate the `Gentle Shutdown`. The purpose of the Gentle Shutdown is to not only disconnect the power to the electrodes but also discharge the capacitors at the same time to prevent risk of potential injury to the operator and the patient/medium. Once ICE is triggered the system is

stuck in this state and will no longer trigger any pulses, even if the operator configures new pulse functions. The whole Control Unit has to be reset, listing 3.3, in order to resume normal operation.

```
    if reset = '1' then
      v := init;
    elsif r.ICE_trigger = '1' then --once ice trigger is '1' only way to reset
    it is to reset the whole system
      v.state_load := fsm_ICE;
    else
      v.state_load := sys_i.state_load;
    end if;
  ------------------------------------------------------------------------------
    if sys_i.ICE = '1' then
      v.ICE_trigger := '1'; --ice trigger is set here and nowhere else
      v.state_load := fsm_ICE;
    end if;
```

Listing 3.3: Code snippet of In Case of Emergency (ICE) trigger in VHDL

3.2 Graphical User Interface Platform

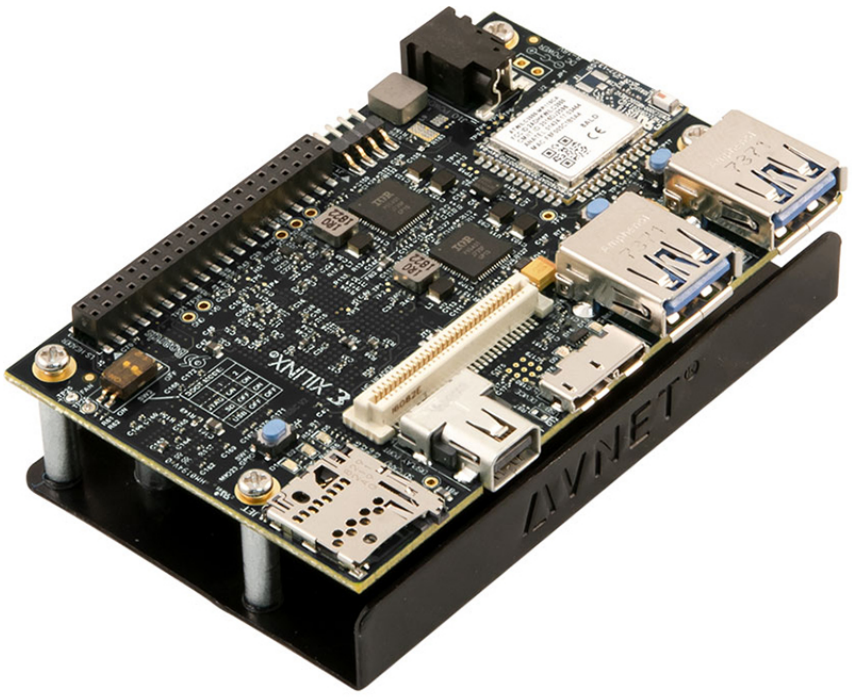

Figure 3.9: The Ultra96v2 from AVNET (2019c).

The Purpose of the User Interface (UI) is to send the pulse parameters for the Control Unit and start the pulsing sequence. The operator inputs their required pulse parameters through the UI and these parameters are written into memory where the AXI Interface is connected. Preceding Elecrtoporator devices have had varying types of UI's from simple character based displays to full graphical ones. The implementations differ based on the type of features they provide, but the basis

is to input data, pulse parameters. For our design, as per our requirements, we opted for a graphical user interface (GUI) platform. The GUI platform will have to be ready to run a Qt5 graphical application. The purpose was not to make a complete GUI but to offer the necessary binaries and environment to be able to deploy one. The reasoning behind this is so that the developers trying to implement our system can tailor their it to their graphical needs.

3.2.1 The Zynq Architecture

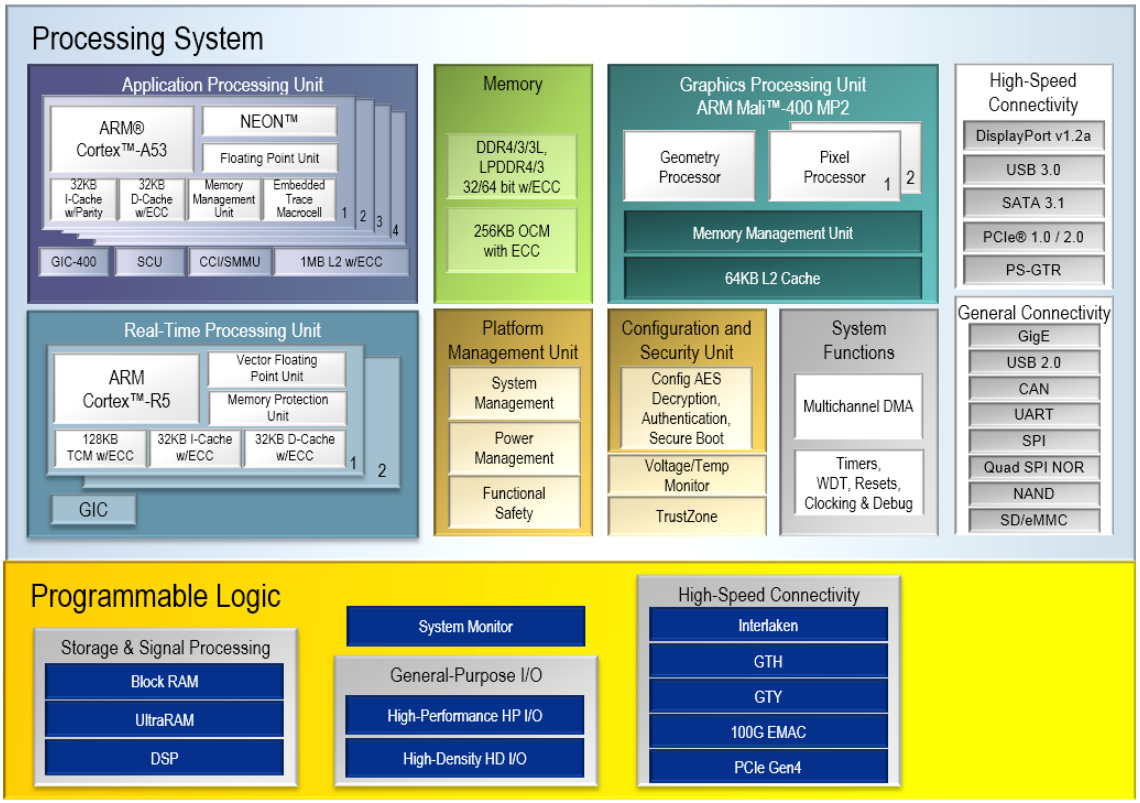

Figure 3.10: Block Diagram of Zynq UltraScale+ EG, Xilinx (2018f)

The Zynq architecture contains an integrated ARM hard cores where an embedded linux system, bare-metal application, or a mix of both can be run. The User Interface (UI) will be running on an embedded linux operating system. The embedded linux system can be run graphically or headless (non-graphically) as all the UI needs to do is write to the memory address where the AXI Interface is located. If the requirements called for a nongraphical user interface we would go with a headless embedded linux running on the Zynq7000 series, which is Xilinx's basic model that includes an FPGA and ARM hard cores, Xilinx (2018d). Choice of development boards for the Zynq7000 series included, but are not limited to, the following: Zybo, Zed Board, Mini-Zed, Micro-Zed, and Cora Z7. However, per requirements, we need to run a graphical user interface. In order to make the development process more streamlined and not to delve too deep into software rendering engines, we opted for the Zynq UltraScale+ family which contains an integrated Mali 400 Graphics Processing Unit (GPU). The graphical rendering will be handled by the GPU as well as the hardware acceleration for graphic intensive applications.

The development board, Ultra96v2 running a Zynq UltraScale+ (xczu3eg), from `AVNET.me/ultra96-v2` was chosen as it met all out requirements for a GUI platform. The Ultra96v2 is the updated version of the previous Ultra96 board and boasts an impressive specification sheet with 2GB RAM, integrated Wifi/Bluetooth, Mini DisplayPort interface, Quad-core ARM-A53 hard cores, Dual-core ARM-R5 realtime cores, and 154K Logic Cells, Xilinx (2018f) and fig. 3.10. The Realtime cores were not used, but they are available for new features in the future. The Ultra96v2 is a Multi-Processor System on Chip (MPSoC) since it contains more than one type of processor and is the cheapest model provided by Xilinx that has an integrated GPU. The listing price of this development board is currently cheaper than buying the BGA chip of this FPGA, at around 250 USD, (230 Euros).

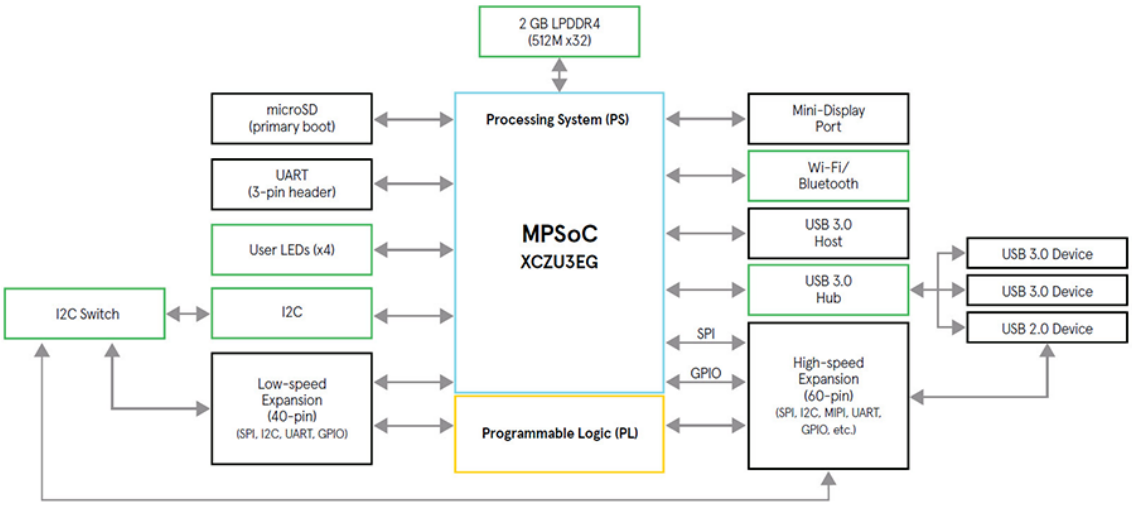

Figure 3.11: Block Diagram of the Ultra96v2 from AVNET (2019c)

Chapter 4

Implementation

4.1 Part 1 - Control Unit of Electroporation Pulse Generation

The Zynq family of FPGA's provided the necessary hardware for us to implement the Control Unit in. The purpose of the control unit was to generate trigger pulses for the power switches, but also configure the power switches for the pulses by setting the voltage. The triggers pulses were generated by the Pulse Generator, and the voltage was set by a Serial Peripheral Interface (SPI) to a Digital-to-Anaglue Converter (DAC). Thanks to the Zynq having a builtin SPI controller, there was no need for an external module. Alternatively, if we lacked an SPI controller, we could have implemented one in the PL by using Xilinx's AXI Quad SPI IP, or making our own.

4.1.1 Build Environment Setup

The build environment ran on a linux host machine. While the control unit can be built on a Windows machine, the GUI Linux cannot be. As a result, a machine running linux was chosen as the host. The host was running Archlinux, but it's not strictly necessary to be running on Archlinux. According to the Xilinx instructions for installing Vivado, supported Linux Operating Systems (OS) are Red Hat, SUSE, CentOS, and Ubuntu. More in depth notes on the supported systems and their versions were available in the installation guidelines for Vivado provided by Xilinx, Xilinx (2018c). In order to develop on the Ultra96v2, or any other Xilinx 7 series or higher FPGA's, Vivado is needed. According to the Memory Recommendations for the Vivado Design Suite the memory usage is dependant on the type of FPGA being used. For the Ultra96v2, ZU3EG FPGA, the typical memory usage is 4GB, up to a peak of 6GB, Xilinx (2019a). Our host system contained 8GB of RAM. For this particular project, Vivado 2018.3 was used. The version of Vivado is important because at the time of this writing, the latest Board Support Package (BSP) available for the Ultra96 is for Vivado version 2018.3. The BSP is collection of files containing the drivers for peripherals as well as software to that is used to initialize the target hardware and communicate with hardware. Before installing, the installation directory for Vivado was setup up with `sudo mkdir /opt/Xilinx && sudo chown $USER:$USER /opt/Xilinx`. During the installation, the path to the newly created install directory was given as well as making sure that the Xilinx SDK was selected as a module to be installed as we needed it later. Once Vivado was installed, the Board Definition File (BDF) was added following the instructions AVNET (2019a) provided. The BDF contains information such as the layout of the board, its peripherals, and initial configuration for the board. If any errors occurred during installation, we consulted the installation document, Xilinx (2018c).

1. Vivado 2018.3 - Xilinx (2018a)
2. Ultra96v2 Board Definition File (BDF) - AVNET (2019a)
3. Ultra96v2 Board Support Package (BSP) - AVNET (2019b)

4.1.2 Register-Transfer Level (RTL) Synthesis and Implementation

In order to implement some logic in the PL, we had to write it using a Hardware Description Language (HDL). Our HDL of choice was VHDL (Very High Speed Integrated Circuit Hardware Description Language) as it is a strongly typed language. The main advantage of this is the early detection of errors, which in turn speeds up development. In the Programmable Logic (PL) there are three main blocks of the Electroporator, the Zynq Processing System (Zynq PS), AXI Registers from AirHDL noasic GmbH (2019), and our Control Unit. The Zynq PS is an Xilinx IP block that is the gateway to many peripherals built into the Zynq, as well as a connection to the ARM hard cores. This is needed for us to be able to input pulse parameters from the PS into the PL where they will be stored. The AXI registers, generated from AirHDL, are where the user pulse parameters will be initially stored. And finally, the Control Unit takes in these parameters from the AXI registers and starts pulsing using these predefined values. To start development, a block design was created and the Zynq PS was added to it though the IP catalogue.

The AXI-registers were generated with the help of an online tool from `AirHDL.com`. The primary reason for the use of this tool as opposed to manually making the AXI registers was the rapid prototyping capabilities of the tool. You input your register map and the tool generates IP package that can be loaded in Vivado block diagram where it can be connected to the ZynqPS and the Control Unit. A memory map was created that contains a register for every parameter of a pulse, as well the control parameters. After a quick Design Rule Check (DRC) from `AirHDL.com` the VHDL Package and Components were downloaded and imported into Vivado as a source. Once the registers are added to the block design, the design assistance wizard prompted a message stating `Run Connection Automation` and the tool was ran. The tool connected the AXI registers to the Zynq PS through a newly added block called the `AXI Interconnect`.

The final, and core part, of the Electroporator is the control unit. The control unit contains four generators which of only one can be selected at once and used. The purpose of the control unit is to turn on and off the high voltage power banks. It does this by sending out square wave pulses, and the duration of these trigger pulses depend on the user configuration, it is comparable to a square wave function generator. The generators contained in the control unit are of two types, mono-polar and bi-polar. Each type has a simple and complex variant, giving us a total of four in total. The generators were build using Finite State Machines (FSM's) where each state corresponded to a particular pulse parameter. What differentiates the simple and complex types are the two extra added states during the generation process, charging and discharging. In order for the control unit to be a versatile, compact, as well as safe, it was developed using the Jiri Gaisler's Two Process Design Methodology, or commonly known as, Structured VHDL Design Method, Gaisler (2011). Using VHDL-2008 features such as packages and records during the development of the control unit helped make it highly configurable and modular where if one needed Lewis and Training (nd), one can take a generator out of the control unit and use it as a standalone module, given the right configuration of peripherals. VHDL-2008 packages are comparable to header files in C/C++ where you can define the value of a specific macro before compilation, and records to C/C++ structs where it was used for port definitions and signals. In order to use our control unit in our design, we needed to place it in the block design where the Zynq PS and AXI Registers are, this is normally done by right-clicking on an empty part of the block design and selecting `Add Module...`. However, since Vivado 2018.3 only supports IP packaging with ports defined as `std_logic_vector` we could not add our generators without making a wrapper around it where it translates the port definitions. Only

one wrapper over the whole control unit was made as it didn't make sense to individually wrap every module as that would create more complexity in the design.

As the connection of the Zynq PS to the AXI registers were already configured thanks to the configuration wizard, all that was left was connect the control unit. Signals were drawn on the block design diagram between the outputs of the AXI registers and the input of the control unit wrapper, and the `Clk` and `Reset` signals were connected to the fabric clock (`pl_clk0`) and Processor System Reset (`peripheral_aresetn`), respectively. Afterwards, the output ports of the pulse generator wrapper were made external be selecting them, right-clicking, and selecting `Make External`. Since we were using the Ultra96v2, the initial configuration of the Zynq Ultrascale+ PS configured the SPI controller to the dedicated pins, Multiplexed I/O's (MIO). If we were using the Zynq7000 series of chips, we would have to enable the SPI controller from the configuration menu of the Zynq PS and make the pins external to a Peripheral Module Interface (Pmod).

Once everything was connected in the block design, fig. 4.1, a main wrapper was created by right-clicking on the block design file, followed by `Generate Output Products...` and `Create HDL Wrapper`. Following a quick validation of the design, the RTL Analysis was run in order to configure the I/O Ports. The I/O Ports definitions were needed to bind a signal on the FPGA fabric to a physical pin on the chip. Synthesis, Implementation, and Bitstream Generation tools were run afterwards in order to create a `.bit` file which can be loaded into the FPGA to configure it. After generating the bitstream, it was exported, `File->Export->Export Hardware->Include bitstream`.

4.1.3 Software Development Kit (SDK) Setup

The Xilinx SDK is used for developing bare-metal applications that can be run on the Zynq. The SDK can be installed as a standalone program or as a part of Vivado. We installed it alongside Vivado. In order to use the SDK with our project we had to export the hardware from Vivado, including the bitstream. This is required by the SDK to set up the Hardware Definition File (HDF), which is used to configure the peripherals. Once the hardware was exported, the SDK was launched through Vivado, `File->Launch SDK`.

4.1.4 Testing Procedure

4.1.4.1 Pulse Generator

In order to test whether our control unit is working as intended, we have to set our pulse parameters and start the pulse generator. This is done by writing to our AXI Registers. The register locations are fixed in memory and can be accessed by running a baremetal application using a pointer to a specific memory address. Each memory address holds 32-bits as that was the word length in our AXI Registers. Since we had several parameters inside one 32-bit register we needed a way to write specific bits at a certain offset of said memory. This could be accomplished by bit manipulation by keeping track of where the offset of a certain field is, or by mapping the memory. We used structs in C to map the memory to match the layout that we had created. This was greatly required for the control register as it selects the type of generator we need and starts it. This method of mapping the memory with structs gave us easier code readability in comparison to using bit manipulation and writing 32-bit words to a register without easily knowing what the fields are. By casting the pointer of the memory address into our structs, we can address each field individually and with

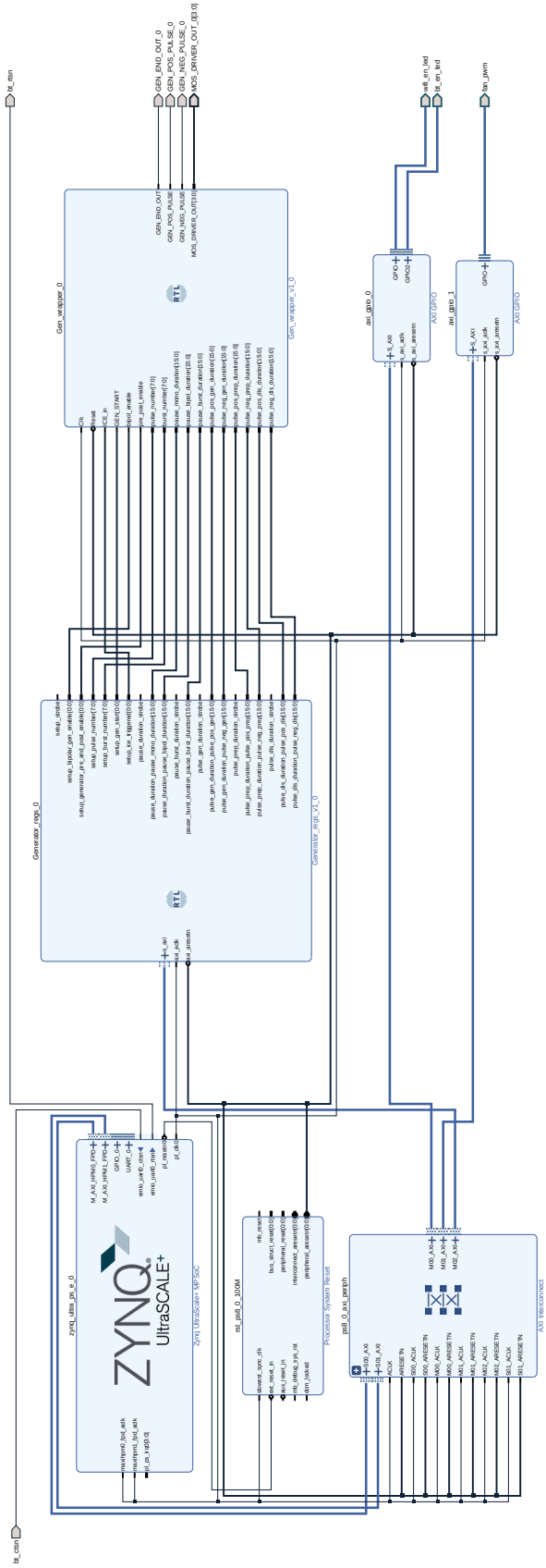

Figure 4.1: Complete board view of the Control Unit in Vivado

great clarity.

```
//---------------------------------------------------------------------
//struct decleration with specific bit length fields
typedef struct {
  int32_t bipol_enbl:1;
  int32_t gen_pre_pos:1;
  int32_t pulse_number:8;
  int32_t burst_number:8;
  int32_t gen_start:1;
  int32_t ice_trigger:1;
} map_setup_t;
//---------------------------------------------------------------------
//macro for easy access
#define SETUP ((volatile map_setup_t *) 0x40000000)
//---------------------------------------------------------------------
//write to individual fields
SETUP->bipol_enbl = 1;
SETUP->gen_pre_pos = 1;
SETUP->pulse_number = 3;
SETUP->burst_number = 5;
SETUP->gen_start = 0;
SETUP->ice_trigger = 0;
//---------------------------------------------------------------------
```

Listing 4.1: Mapping memory address with structs example

4.1.4.2 Serial Peripheral Interface (SPI)

Thanks to the example code provided by the Viavdo SDK for the SPI controller, it was easy to set up a working SPI Master. The example code that we referenced was found through the `system.mss` file. This file was generated by creating a new Application Project. The examples were then imported and ran, such as the `selftest_example`. We configured the SPI as a master as well as timing specific settings for our test Digital-to-Analogue Converter (DAC). Such settings included a master role, active low clock, maximum of 130MHz clock, and 12-bit resolution. Using the `XSpiPs_SetOptions` method with the options, `XSPIPS_MASTER_OPTION`, `XSPIPS_MANUAL_START_OPTION`, and `XSPIPS_FORCE_SSELECT_OPTION`, the SPI controller was configured. All these options have to be written in the second argument of `XSpiPs_SetOptions`, using the OR operator, denoted by the pipe symbol |, in-between each option. The Read/Write First In First Out (FIFO) memory of the SPI controller is fixed to a width of 8-bits, and since our DAC required 16-bit wide word transmissions we had to set the `XSPIPS_MANUAL_START_OPTION` and the `XSPIPS_FORCE_SSELECT_OPTION`. However, since the FIFO width is fixed to 8-bits, the first and last 8-bits of the 16-bit transmission are flipped. This is because the SPI controller iterates through the given array of data and splits them into 8-bit chunks for the FIFO. Then every two 8-bit chunks from the FIFO are sent in the order they were inputted into the FIFO. To get accurate transmission of data, the first and last two bytes had to be flipped and this was done by simple bit manipulation and shifting. For the clock frequency, that was configured using the `XSpiPs_SetClkPrescaler` method with the parameter `XSPIPS_CLK_PRESCALE_16`. All the configuration options of the SPI controller could be found in the notes of the `xspips.h` file, which is located in the `libsrc` directory of the project or by control-clicking the `#include "xspips.h"` line.

```
//----------------------------------------------------------------------
//Needed libraries
#include "xparameters.h"
#include "xspips.h"
//----------------------------------------------------------------------
//Device ID from xparameters.h
#define SPI_DEVICE_ID   XPAR_XSPIPS_0_DEVICE_ID
//----------------------------------------------------------------------
//Variable declaration
XSpiPs Spi;     //The instance of the SPI device
XspiPs_Config *SpiConfig;
//----------------------------------------------------------------------
//Link the SPI controller
SpiConfig = XspiPs_LookupConfig(SPI_DEVICE_ID);
if (NULL == SpiConfig) {
  return XST_FAILURE;}
//Initialize the controller
Status = XspiPs_CfgInitialize(&Spi, SpiConfig, SpiConfig->BaseAddress);
if (Status != XST_SUCCESS) {
  return XST_FAILURE;}
//quick test if it's running and accessible
Status = XspiPs_SelfTest(&Spi);
if (Status != XST_SUCCESS) {
  return XST_FAILURE;}
//----------------------------------------------------------------------
//Configure the controller
XspiPs_SetOptions(&Spi, (XSPIPS_MASTER_OPTION | XSPIPS_FORCE_SSELECT_OPTION |
    XSPIPS_CLK_ACTIVE_LOW_OPTION | XSPIPS_MANUAL_START_OPTION));
//Set the clock frequency. Reference clock is set to 187.5MHZ -> rCLk/16 ~= 12
    Mhz
XSpiPs_SetClkPrescaler(&Spi, XSPIPS_CLK_PRESCALE_16);
//Select the slave
XSpiPs_SetSlaveSelect(&Spi, 0);
//----------------------------------------------------------------------
//Function to flip the first and last 8-bits. (u16 is a typedef for short)
u16 Flip_single_value(u16 in){
    //simple bit manipulation
    return (((in & 0xff00) >> 8) | ((in & 0x00ff) << 8));}
//----------------------------------------------------------------------
//Buffers for the read and write data.
u16 TxBuffer[4];
u16 RxBuffer[4];
//Example data input, the values have to be flipped.
TxBuffer[0] = Flip_single_value(0x000f);
TxBuffer[1] = Flip_single_value(0x00f0);
TxBuffer[2] = Flip_single_value(0x0f00);
TxBuffer[3] = Flip_single_value(0x0fff);

//Data transmission
//Send two bytes and ignore incoming data.
XSpiPs_PolledTransfer(&Spi, (u8*)&TxBuffer[i], NULL, 2);
//Send two bytes and receive two bytes.
XSpiPs_PolledTransfer(&Spi, (u8*)&TxBuffer[i], (u8*)&RxBuffer, 2);
//----------------------------------------------------------------------
```

Listing 4.2: SPI Master setup in C for Zynq

4.2 Part 2 - Graphical User Interface Platform

The Zynq Ultrascale+ with the integrated GPU that's on the Ultra96v2 gave us the possibility of running a full graphical interface for the Electroporator. There are many different graphical toolkits and frameworks available such as GTK+, Qt5, PyQt5, and Mono, to name a few. We decided on Qt5 as it had some tools and features which made developing on it a delight. But before we could run any graphical application on the Zynq, we needed an operating system. An Embedded Linux operating system is usually built from source to the specifications of the target hardware. In order to optimise this process there are tools that can help automate and build Embedded Linux OS's such as BuildRoot and the Yocto Project. Xilinx has their own tool for their SoC's called Petalinux which is based on the Yocto Project.

To be able to test and debug the graphical capabilities of the Ultra96v2, some peripherals were required. Aside from the development board were required, an AVNET (2018) `Ultra96 USB-to-JTAG/UART Pod`, USB keyboard and mouse, and a Mini-DisplayPort (mDP) adapter. We had opted for a mDP adapter to HDMI and, according to the Xilinx online support, it had to be an active adaptor as the integrated DisplayPort controller does not support DisplayPort++, Xilinx (2018e). A normal mDP to DP adaptor would also suffice, but we did not have a DisplayPort capable monitor available. The display that we were using was a generic 10-inch touchscreen display designed for RaspberryPi's.

4.2.1 Petalinux Setup

Petalinux only supports running on linux host machines, and officially supports Redhat, CentOS, and Ubuntu, according to their guide, Xilinx (2018b). However, our host machine is not among them. But that did not mean that we could not use the tool. With the right libraries installed and correct environment setup we could run it. However, Petalinux was not designed to run on a rolling release operating system like Arch Linux and thus required some modifications to the environment as well as links to libraries. Doing so would jeopardize the system's structure as some programs are reliant on it. According to the Petalinux reference guide, an alternative to running Petalinux natively was through a Virtual Machine (VM), such as Virtual Box, but that came at a cost of performance, Xilinx (2018b).

4.2.1.1 Supported Operating Systems

For supported operating systems, we followed the reference guide, Xilinx (2018b). In accordance to the reference guide, the development environment had to be initially set up before Petalinux can be installed. Setting up the environment meant that all the required packages for Petalinux needed to be installed and configured. A full list of required packages are listed in the reference guide from page 11 to 12 and categorized by supported operating system, Xilinx (2018b). Once the environment was set up, the tool was downloaded and installed in a predefined directory under `/opt/`. Before the Petalinux tools can be used, they had to be sourced into the current shell, (command interpreter), with `source "/opt/Petalinux/2018.3/settings.sh"`.

4.2.1.2 Unsupported Operating Systems

For unsupported operating systems, the only method available according to the reference manual was to use a virtual machine running one of the supported operating systems. But this option came at a cost of performance. However, there was an alternative that was not listed in the reference guide, and that was to use a Docker container. According to Docker (2019), "A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another". This solution provided almost no cut to performance as opposed to using a virtual machine, Felter et al. (2015). To get a Docker Container running, we needed to first setup Docker and then create a configuration file for the Docker container that contains Petalinux. Docker is available in all the major Linux Distributions from their package manager, in our case it was from Pacman and was installed with `sudo pacman -S docker`. A Docker configuration file for Petalinux 2018.3 was used to setup the container where the Petalinux tool was installed. There was also a need to setup a shared directory for the Docker container to write into as everything done inside the container is not persistent and will be deleted once the container is closed.

4.2.2 Petalinux Configuration and Build

There are two ways to build a Petalinux image for the Zynq, from scratch or from a Board Support Package (BSP). Since the BSP already comes with drivers and configurations for the Ultra96v2, we opted for this method. To get started, a Petalinux project was created by inputting the BSP, in the `petalinux-create` command. This created a project directory in which all the build files pertaining to this build will be located in. To configure the build, the `petalinux-config` command is issued with specific arguments to components that need to be configured. For the first build we used the stock PL configuration that was already in the BSP and ran the config command with the argument `\--get-hw-description=<path-to-HDF-file>`. For the following builds, we linked the new HDF file from the Xilinx SDK where our control unit is. Kernel and Root File System (rootfs) components were configured afterwards. The only thing that needed modification was the rootfs so that we could add the Qt5 binaries. This was done by selecting the `packagegroup-petalinux-qt` and `populate_sdk_qt5` options. The final step to getting a Linux image was to run the `petalinux-build` command. In order for the build to be successfully completed, it required a least 100-GB of free space, Xilinx (2018b). Once the build was complete, we needed to package the build files so that it can be booted on hardware. This was done by using the `petalinux-package` command from inside the images directory, `<path-to-project-directory>/images/linux`. To get the Petalinux image booting, we had to put it on an Micro-SD card. The Micro-SD card had to be first be partitioned according to the Petalinux reference guide. To partition the card we had to use one of the most popular GUI partitioning tools available on Linux, `GParted`. However, other tools such as the command line `fdisk, parted` can also be used. The Micro-SD card needed two partitions, the boot partition and the root file system (rootfs) partition. The boot partition had to be `Fat32` type and the rootfs an `ext4` type. To the boot partition the following files had to be transferred, `BOOT.bin` and `image.ub`. On the rootfs partition, the contents of `rootfs.tar.gz` had to be extracted into. However, during the extraction we noted that the extraction of this file had to be done under super user privileges, such as running it with the `sudo` command. This was done to keep the file permissions in order.

```
#Create Petalinux Project from Board Support Package (BSP) at current location
petalinux-create -t project -s <path-to-BSP>/<BSP-name>.bsp

#Add Hardware Defintion File (HDF) found in Vivado project.sdk directory
petalinux-config --get-hw-description=<path-to-HDF-file>

#Configure the Petalinux Kernel
petalinux-config -c kernel

#Configure the Root Filesystem (Qt5 is configured here)
petalinux-config -c rootfs

#Build the Petalinux project
petalinux-build

#Package the Petalinux build by first going into the image directory then running petalinux-package
cd <path-to-petalinux-project-directory>/images/linux/
petalinux-package --boot --fsbl zynqmp_fsbl.elf         \
                  --u-boot u-boot.elf                   \
                  --pmufw pmufw.elf                     \
                  --fpga system.bit
```

Listing 4.3: Petalinux Build Command Order

4.2.3 Qt5 Setup and Deployment on Hardware

4.2.3.1 Qt5 Setup on Host Machine

The development of the UI was done on the host machine where Qt5 application will be cross-compiled for the target device and deployed to it. Since Qt is widely used application framework for embedded linux as well as desktop linux, it can be easily installed and set up. Majorty of Linux based distributions contain Qt5 binaries that can be installed though their package manager. However, the binaries in those package managers are the latest up-to date versions. Since Petalinux uses precompiled binaries for their packages, the versions of these are lagged behind. Petalinux 2018.3 contains Qt5 version 5.9.6, while the latest version at this time is 5.13.2. To prevent version miss-match between the host and target hardware, one version has to be use on both. As a result, an archived version of Qt5 was downloaded from their repository and installed in a predefined location, `https://download.qt.io/official_releases/qt/5.9/5.9.6/`. The resulting applications, Qt-Creator and Qt-Designer, was for writing code logic for the graphical application and for setting up the graphical layouts and display, respectively.

4.2.3.2 Software Development Kit (SDK) cross compiling

In order to be able to build our Qt5 application on our host machine, we need to build an SDK for it. This was done by running `petalinux-build --sdk`, and it required at least 120-GB of free space for the whole project. Once the SDK was built and packaged, it had to be installed on the host system. The installation varied if we were running petalinux natively or on a Docker container. For a native setup, we needed to run the command `petalinux-package --sysroot`. Running the command for the native setup on the Docker container installs the SDK in the Docker container, but it is erased after the container is closed and making it unusable. We had to install the SDK on the host machine, and not the Docker container. This was done by running the script generated by Petalinux, `<path-to-project-directory>/images/linux/sdk.sh`.

4.2.3.3 Deployment on Hardware

To make development easier for cross-platform and cross-configuration easier, Qt-Creator groups settings that are use for building and running applications into `Kits`. Kits contain information about the target environment such the device type, architecture, compiler, debugger, and miscellaneous metadata. Currently configured kits are located in `Tools->Options->Build&Run->Kits`. Before we can make a kit for our target hardware, we need to fist add the SDK that was compiled by Petalinux. This was done by adding a field in the `Qt Versions` tab under the `Build&Run` options and linking the `qmake` file that was created by the SDK. The `qmake` file is located inside the SDK directory, `<sdk-dir>/sysroots/x86_64-petalinux-linux/usr/bin/qt5/qmake`. Afterwards, the compilers of the SDK were linked to the compilers tab, C compiler `aarch64-xilinx-linux-g++` , and C++ compiler `aarch64-xilinx-linux-g++`. The kit was created once all the compilers and qt versions were linked to Qt-Creator. To be able to automatically deploy to the target hardware, we had to add a device to Qt-Creator. The device category in options needs a way to log into the device remotely in order to transfer the compiled application to the target device. This is done through a Secure Shell (SSH) connection where the files are transferred through the Secure Copy Protocol (SCP) and the launched application launched. For the device to be accessible by Qt-Creator, its credential have to be inputted. In our case, the root user, its plain-text password, and ip address was issued. However, a plain-text password is not is not recommended as it's insecure and can be accessed just by going to the settings and viewing them. An alternative, and preferred method, is to use a key generated from the `ssh-agent`. The device was tested afterwards by the test function built into Qt-Creator.

A new application was created on Qt-Creator and the Petalinux Kit was selected to be its deployment target. But before an application can be run on the target hardware, some extra arguments have to be added to the run configuration. According to Xilinx's support, the argument `-qws` had to be set, and under the `Build Environment` the following was needed `DISPLAY=0.0`, Xilinx (2014).

4.2.4 Accessing the Control Unit from User Space

Accessing the Control Unit was primarily done by running a bare-metal application from the Xilinx SDK, and writing to specific memory addresses where the AXI-Registers were located. In the User Space, the same method of accessing the AXI-Registers cannot be used. User space is a portion of memory allocated for running applications on an operating system. A work around to this is to

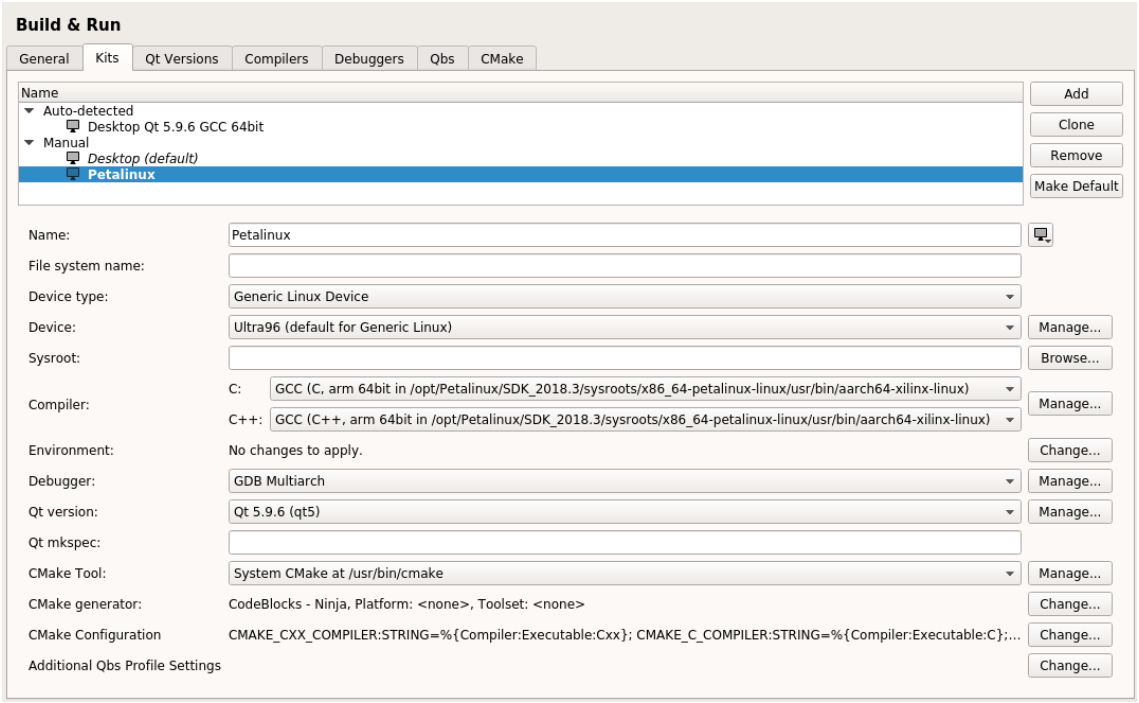

Figure 4.2: Setup details of the Petalinux Kit in Qt-Creator

use `/dev/mem/` and the `mmap` function, which provides access to system's physical memory, Linux (2015), and maps the device into memory, Linux (2019), respectively. Super user privileges were required to access `/dev/mem/`. The memory address offset for the AXI-Registers was found in Vivado address editor and also provided by the generated AirHDL C Header file. We used Python to test the access to the Control Unit since it provided an interactive environment where we could change variables easily without needing to recompile the program, as opposed to it being made in C.

```
1  ##-----------------------------------------------------------------
2  #Required libraries
3  import mmap
4  import os
5  ##-----------------------------------------------------------------
6  #Open /dev/mem
7  mem = open("/dev/mem", "r+b")
8  ##-----------------------------------------------------------------
9  #map the memory address from the start to 24 bytes, length of
       register
10 mm = mmap.mmap(mem.fileno(), length=24, offset=0x40000000)
11 ##-----------------------------------------------------------------
12 #Write to address
13 mm[0] = 0xf
14 ##-----------------------------------------------------------------
```

Listing 4.4: Physical Memory access using /dev/mem in python

Chapter 5

Evaluation

5.1 Part 1 - Control Unit (Pulse Generator)

5.1.1 Bare-Metal Testing

Simulation testing was first carried out using Vivado's built-in simulator. Each sub-module of the Control Unit was tested before deployment on hardware. Simulations were set for a duration of 2500 nanoseconds and was testing for stability, repeatability, and accuracy. Test-benches were written in VHDL for each sub-module and the Pulse Generator Wrapper. Stability testing was done by altering the input parameters of the generator, while it was active and pulsing, to determine if the outputs will change from the preconfigured values. Repeatability testing was done by triggering consecutive pulses of the same type in burst mode and comparing the variances of each pulse widths. Accuracy testing was done by triggering a range of pulses starting from 10 ns to 300000 ns, measuring the pulse widths, and comparing the results to the expected pulse widths. Testing on the hardware, Utra96v2, was carried out similarly to simulation, using the same testing methods with a bare-metal application through the Xilinx SDK.

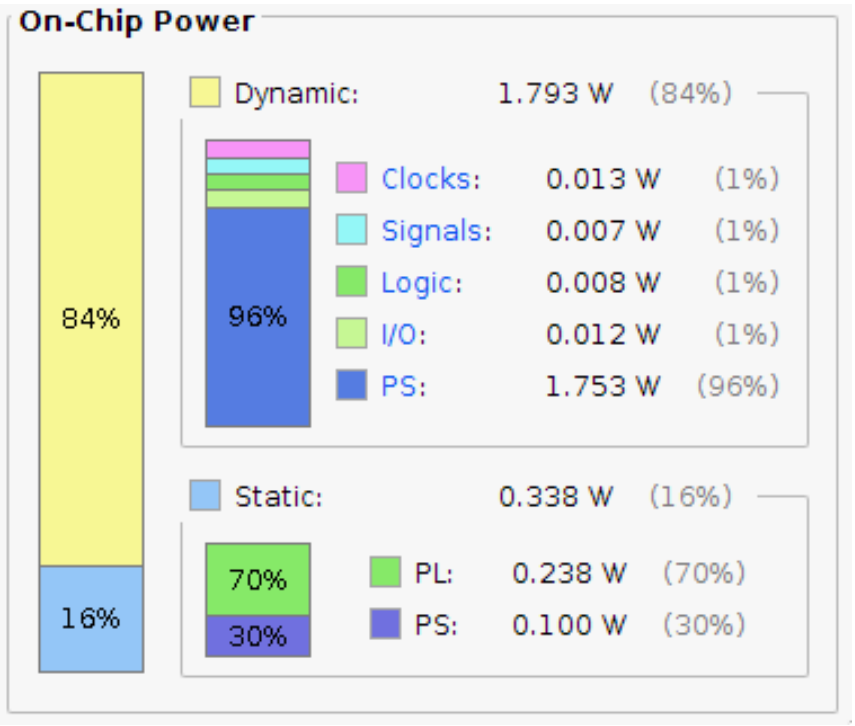

Figure 5.1: Power Analysis of the Control Unit on the Ultra96v2

5.1.2 Synthesis and Implementation

Synthesis of the Control Unit took between 30-35 seconds to complete running on 8 threads and 8GB of RAM. During consecutive Synthesis runs, there seemed to be some sort of memory leak which caused Vivado to throw Critical Error Messages regularly of `Fork failed:Cannot allocate memory`. Restarting Vivado helped to fix this. There were no timing violations nor errors from synthesis and the constraint file only contained pin assignments. Implementation took between 2:15-2:26 minutes on the Ultra96v2, (ZU3EG), to complete. Alternatively, the implementation on a Zynq7000 FPGA was between 53-55 seconds while the synthesis had similar time. There were no timing violations nor errors. But the timing summary states that the Worst Hold Slack (WHS) was 0.010ns. Similar WHS was noted when using a larger AXI Register size, i.e. 32-bit parameters instead of 16-bit. All of the top 10 WHS's were located inside the AXI-Interconnect. A total number of 3979 Look Up Tables (LUT), and 4352 Flip-Flops (FF), 5,64% and 3,08% utilization, respectively, were used for the entire deign. The estimated Total On-Chip Power from Vivado of the implementation was 2,132 Watts, fig. 5.1. The majority of the power consumption came from the dynamic load of the PS, at 96% of the total power. The PL had 0,238 Watts of static load, 11,2% of the total power.

5.1.3 Performance

Performance of the system was determined based on the stability, repeatability, and accuracy testing. Simulations testing these three factors yielded perfect results, no error nor deviations. Hardware yielded similar results with slight deviations. The stability testing resulted in no changes on the outputs of the Pulse Generator while the inputs were changing during pulse generation. The system created continuous duplicate pulses with little to no differences in pulse widths. The results of the accuracy test are shown on Table 5.1 containing the pulse samples from 10ns to 300000ns. 300000ns was chosen as the limit because the AXI registers were configured to 16-bit wide pulse parameters. The results of Table 5.1 were acquired by using a 1,5GHz high speed probe, running at full bandwidth, measuring at least 2.4V on the rising and falling edge of the pulse. Since the output is using the LVCMOS33 I/O Standard, minimum of 2.4V equals to a logical 1 (HIGH). Mean Absolute Error (MAE), eq. (5.1), and Maximum Jitter, eq. (5.2), of the measured pulses are compared on table 5.1, column 6 and 7, respectively.

Pulse Accuracy						
Expected	A	B	C	D	MAE	MAX Jitter
10ns	9,9ns	9,8ns	9,9ns	9,9ns	0,2ns	0,125ns
20ns	20,1ns	20,1ns	20,1ns	20,1ns	0,1ns	0,1ns
50ns	50,1ns	50ns	50,1ns	50ns	0,1ns	0,05ns
100ns	100ns	100ns	100ns	100ns	0ns	0ns
500ns	500ns	500ns	500ns	500ns	0ns	0ns
1μs	1μs	1μs	1μs	1μs	0ns	0ns
10μs	10μs	10,0001μs	10μs	10μs	0,0001μs	0,000025μs
100μs	100μs	100,0007μs	100μs	100μs	0,0007μs	0,000175μs
300μs	300,00238μs	300,00176μs	300,00236μs	300,00156μs	0,00238μs	0,002015μs

Table 5.1: Expected and measured pulsed widths

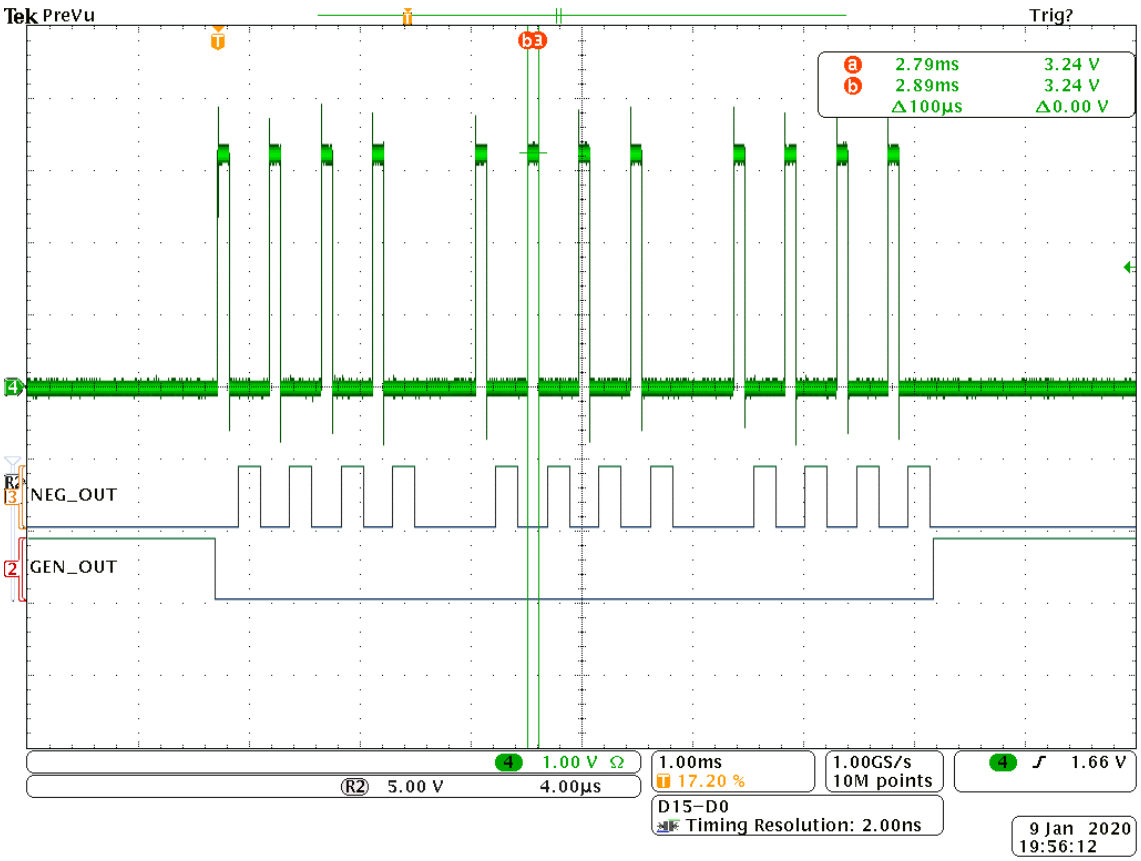

Figure 5.2: Measuring generated pulse widths

$$MAE = \frac{1}{n}\sum_{j=1}^{n}\left|y_j - \hat{y}_j\right| \tag{5.1}$$

$$MaximumJitter = MAX\left|y - \hat{y}\right| \tag{5.2}$$

$$Maximum\,Pulse\,Width = Max\,Signed\,State\,Counter\,Value * Min\,Pulse\,Width \tag{5.3}$$

5.1.4 Pulse Resolution

The minimum pulse width the Control Unit was able to produce was 10ns for each state of the generation, with the clock running at 100MHz. The maximum pulse width of the generator, according to formula 5.3, is determined by multiplying the maximum signed state counter value, in our case we used a 16-bit counter, by the minimum pulse width, which resulted in $32767 * 10ns = 327670ns$. Due to the limitation of the AirHDL tool, the AXI Registers can be configured to a maximum of 32-bits per register, which results in a maximum configurable pulse width of $2147483647 * 10ns \approx 20seconds$. The state counters increment by one very clock cycle which resulted in a $1:1$ ratio of the clock period to the minimum pulse width that it can be generated. The highest clock frequency the Pulse Generator can run at is 130MHz, minimum of 7,69ns pulse width. This was measured by

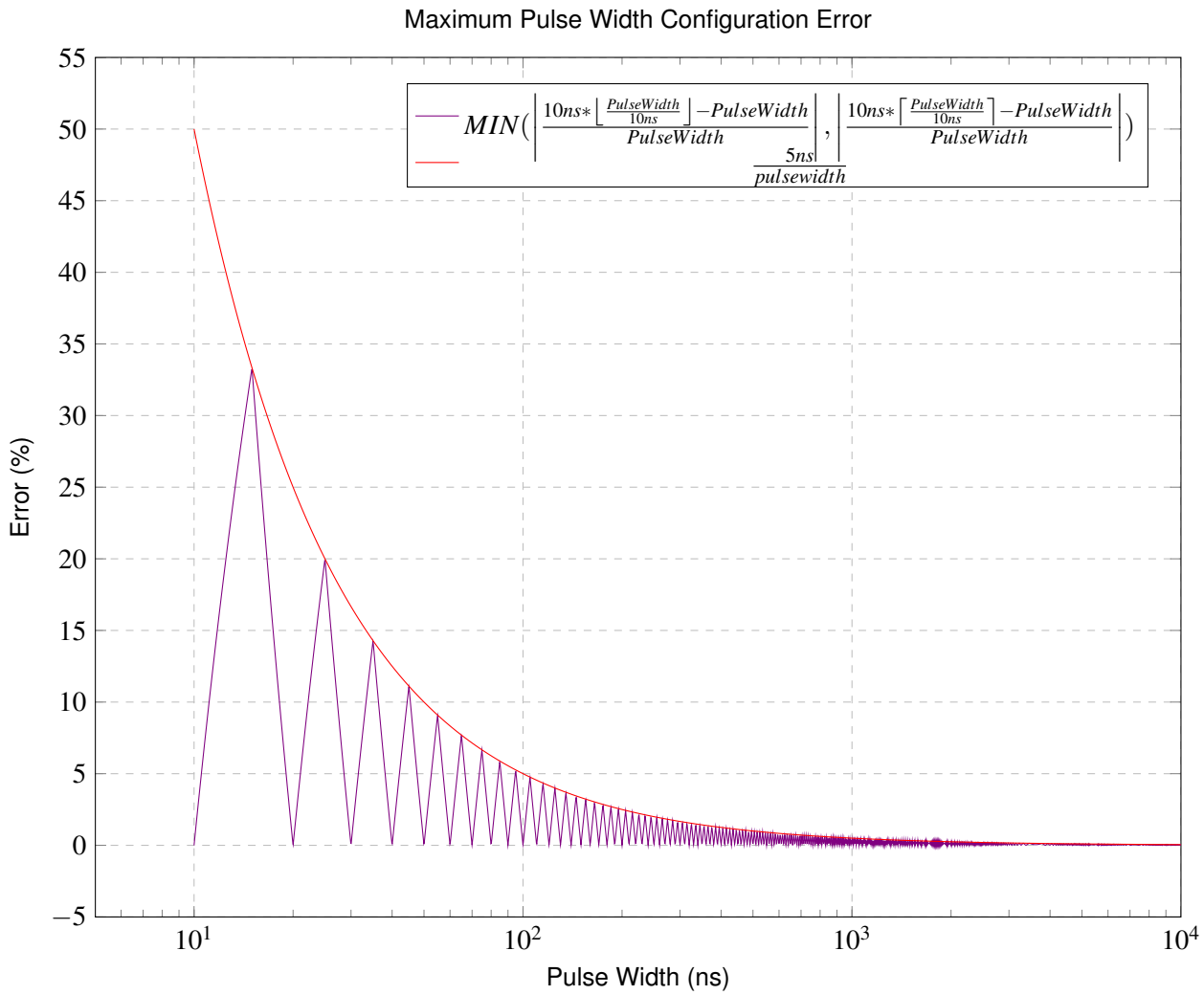

Figure 5.3: The Error Rate of Pulse Width Configuration Error

removing the AXI-Registers from the design, wiring constants to the inputs of the pulse parameters, and wiring the trigger of the Pulse Generator through the GPIO interface. With a 10ns pulse resolution, a power switch's maximum pulse width configuration error can be defined by the formula 5.4. The maximum error occurs at the every 10 nanoseconds with a base of 5, (15, 25, 35, etc). This results in a decaying triangle, as shown on figure 5.3. The formula $5ns/pulsewidth$ is the simplified version of the decaying triangle formula, but it doesn't account for the zero percent error for pulses with a base of ten.

$$MIN(\left|\frac{10ns*\lfloor\frac{PulseWidth}{10ns}\rfloor-PulseWidth}{PulseWidth}\right|,\left|\frac{10ns*\lceil\frac{PulseWidth}{10ns}\rceil-PulseWidth}{PulseWidth}\right|) \tag{5.4}$$

5.2 Part 2 - Graphical User Interface (GUI) Platform

5.2.1 Petalinux

A bootable Petalinux image was created using the Petalinux tool running on the Docker container. There were no issues during the build of the image as long as the commands were followed in order. A single case occurred where the whole project became corrupted because the u-boot module was altered after the project was already built. Rebuilding the project caused the corruption and the only solution was to recreate the project. This didn't occur with the other modules, rootfs, kernel, and default. Whether this issues happens with other modules other than he u-boot one is unknown. The initial build of the image took 30 minutes to complete. Succeeding build times varied based on quantity of changes as well as the module that's being altered. The SDK build crashed several times as disk space kept running out through out the build, but the estimated time for the SDK build was 40 minutes. The extraction of the rootfs onto the micro-SD card had to be done as a super user. Otherwise, during boot, kernel panics were thrown noting that it cannot mount the rootfs.

The image booted successfully into a graphical environment running Matchbox as the Window Manager. All the input peripherals were detected, USB keyboard and mouse, and were working as intended. We also tested a portable touchscreen display, and the touchscreen was detected and the driver loaded. This was observed by running the commands `dmesg`, and `xinput list` afterwards, fig. 5.4. The SPI Controller driver was not loaded into the kernel as it was not listed under `/dev/`, but the contents of it could be found under `/sys/class/spi_master`. There were no attempts to get it working without the spidev driver. Aside from direct access to the system through the graphical environment, the UART connector was used. Using a UART-to-USB module connected to the host machine, gave an alternative method to access the Ultra96v2. To use the UART interface, the `screen` utility was used from the host machine, `sudo screen -L /dev/ttyUSB1 115200`, fig. 5.5. The difference between the UART and the graphical interface was that during boot time, the UART interface displayed a verbose output of the system booting as well as the kernel messages, while the graphical interface only stated a `System booting up` message.

Internet connectivity was setup using the Ultra96v2's web application for configuring WiFi. The connection speed was tested using the `ping` command and observing the packet drops and the latency. On average, the latency to the ip address 1.1.1.1, (the free DNS service), was 40 milliseconds. However, at random intervals, the latency would fluctuate between 150-2000 milliseconds or even drop. Attempting to Secure Shell (SSH) into the device yielded mixed results as the unstable connection at random intervals made it difficult to type commands and to observe the outputs.

5.2.2 Qt5 Application Deployment

The installation of Qt5 as well as the cross compiled SDK was successful and yielded no issues. A new Qt kit was added and Qt Creator was able to access the Ultra96v2 remotely for deployment of applications. A test application, `Qt Quick Demo - Calqlatr`, was compiled from the examples of Qt Creator for the Petalinux kit. Qt Creator threw the error `No screens found` during the deployment of the application, fig. 5.6. As a result, the application was not automatically launched and displayed on the target hardware. However, the compiled application executable file was present on the Ultra96v2. The application executable file was able to be manually run through a terminal window from the Ultra96v2 and launched without issues. The Qt virtual keyboard exam-

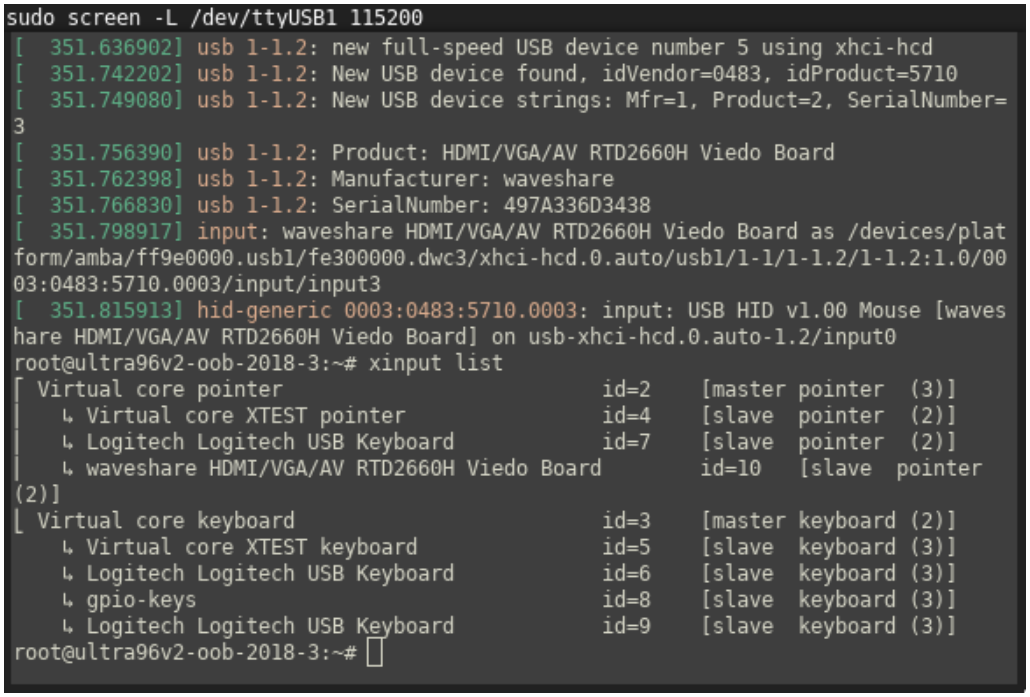

Figure 5.4: Dmesg and Xinput output from the Ultra96v2

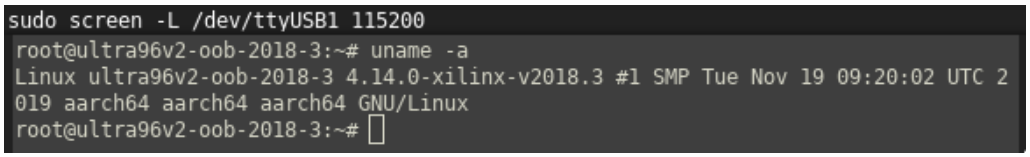

Figure 5.5: Terminal access using the screen command

ple, `Qt Quick Virtual Keyboard - Basic Example`, was also compiled and set up for deployment on the Petalinux kit. When deploying the application, the same, `No screens found`, error persisted. However, manually running the executable from the linux terminal threw a new error, `QML module not found (QtQuick.VirtualKeyboard)`.

```
Starting /usr/share/qt5/examples/quick/demos/calqlatr/calqlatr...

QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
qt.qpa.screen: QXcbConnection: Could not connect to display 0
Could not connect to any X display.

Application finished with exit code 1.
```

Figure 5.6: Qt Creator failed to find screen 0 on the Ultra96v2

5.2.3 Control Unit Access from User Space

Since the performance of the Control Unit was already tested using a bare-metal application, there was no need to retest it from Petalinux. But we needed to test that the Control Unit can be accessed from Petalinux and triggered. To do that, we used `/dev/mem` and `mmap` to access the AXI-Registers from the user space and verified that we were able to write to the AXI-Registers as well as read from them. The verification was done with two steps, writing and reading. First, we mapping out every 4-bytes from the offset address, 0x40000000, for a total of 24-bytes, writing pulse parameters to them, and closing the map. Afterwards, a new map was opened in the same offset address and its contents were read comparing them to the written values. The test was successful in writing and reading. Validation testing was done by a logic analyzer measuring the outputs of the Control Unit while it was generating pulses and verifying the results with the inputted values. The Control Unit was able to be accessed from the user space from Petalinux and the pulses triggered according to the input pulse parameters.

Chapter 6

Discussion

6.1 Part 1 - Control Unit (Generator)

The Control Unit passed all tests that were required and performed its function precisely. This is not surprise as it was designed on an FPGA which is known for its precision. As long as the design met its timing constraints it would perform as described in the Register Transfer Level (RTL). The use of the bare-metal testing over the testing through the user space was a smart choice as it helped isolate the FPGA from the software platform. It was much easier to access the memory mapped AXI-Registers through bare-metal as there are no restrictions to memory access in comparison to typical linux user hierarchy systems. The FPGA utilization on the Ultra96v2 was low, but this is due to the very large size of the FPGA on the board. This extra space leaves room for features that are discussed in subsection 6.4.1. Moving forward, there seems to be no reason to continue using the dataflow method after having experienced the two process method. The code organization with records, having a global "interface" package that ever module loads makes the code much easier to understand and configurable by just modifying the package.

6.1.1 Design Methodology

The successful results of the Control Unit can be greatly credited to Gaisler's two process design method that was used. It was a real pleasure using this design over the traditional method as it made designing the parts of the Pulse Generator much simpler and faster. The resulting design is much easier to maintain and understand thanks to its higher abstraction level. While there is parallel flow in the design, the manner in which it was written, using sequential statements to structure complex statements resulted in the source code that greatly helps the readability and user understanding of how the system was designed. Code reusability was a big factor in the overall design as isolating a complex system into smaller modules helped by breaking the complexity down. The reduced complexity resulted in the application of an iterative design approach where a simple module was created every iteration and easily tested with simple test-benches.

The synthesis and implementation design runs took no more than three minutes all together. This is thanks to the greatly reduced number of processes in the design as well as the sequential statements used in the Pulse Generator. At most a module contained two processes in comparison to the dataflow model where there were over 20 processes. Simulation speeds also increased thanks to the two process method. The power analysis of the Control Unit was expected with the Programmable Logic (PL) consuming less power than the Processing System (PS). The PL uses a fixed clock while the PS has dynamic frequency scaling resulting in a large variance.

6.1.2 Numerical Performance

With the implementation of the Control Unit on the Programmable Logic (PL), it was expected that the results of the configured versus the generated pulse widths would be the same. According to the results on table 5.1 the maximum deviation was 0,2 ns from the expected pulse. However, since the implementation met its timing constraints we did not expect there to be any deviation from the expect pulse parameters, especially with a mid range FPGA. We speculate that the deviations are a result of inadequate measurements. Using the 1.5 GHz high speed probe, with the formula $Bandwidth = 0,34/t_{risetime}$, we get a window of 0,2 ns accuracy. With the short pulses, the rise and fall times are very quick, and with a low accuracy in the rise and fall times it is difficult to get an accurate measurement. Even if there is some jitter in the trigger pulses, it's negligible.

The maximum pulse width configuration error refers to how well the Control Unit can generate specific pulses required by the high voltage switching sub-system. The conversion ratio of the high voltage switching is is non linear, especially when it comes to high voltages. If you wanted 100 ns pulses at 3 kV these actual output from the high voltage switching will be much less than 100 ns. This is due to the rise and fall times of the pulse, and the actual pulse duration at 3 kV might come out to be around 71 ns. In order to get 100ns pulses on the electrodes, it might require a trigger pulse duration of 125 ns. But since our Control Unit has a pulse resolution of 10 ns we can either deliver 120 ns or 130 ns, and we will always have an error of 5ns. That is why figure 5.4 shows a decaying triangle function. Anything requiring a pulse duration lower than a base of 5 will be rounded down and higher than a base of 5 will be rounded up to get the minimum error. The error decays as the required pulses become larger, and at 1000 ns the error rate is negligible. This error is significant to pulses lower than 100 ns, anything higher can be neglected.

6.1.3 System Safety and Stability Design Decisions

The stability testing yielded results that we were expecting as the design in hardware would not allow any kind of alteration to the pulse parameters during its operation. This is reassuring because in an event of a malfunction in the software it would not jeopardize the hardware. The Pulse Generator part of the Control Unit is isolated from the rest of the system during operation, except the ICE trigger and Reset. Hardware malfunction is not likely to occur from our design but external interference is not ruled out. In case of external interference, due to the system's fault tolerance, the Control Unit will attempt to finish its pulse function. Since the Control Unit is electrically isolated from the high voltage switches, the likelihood of frying the FPGA's I/O pins is reduced. The safety levels build into the Control Unit was done so more for the clinical application of the device. The device has potential to be deployed in clinical environments and as a result safety equally important as system functionality. In the event of a failure in one of the sub-systems, it should not jeopardize the others. The Control Unit outputs, aside from the triggers, a control signal that can be configured to an active low or high. In the case of an active high configuration, the control signal will switch from high to low during operation and switch back when all of the pulse functions are delivered. This can be used to tell the high voltage switching system that charging the capacitors are only permitted during the time the control signal is low.

6.1.4 AXI-Registers

The utilization of the development tool, AirHDL, to generate the registers helped speed up the design process. The benefit of this tool greatly credited to its quick prototyping capabilities. This can be observed on the revision counter of figure fig. 3.7. There have been a total of 66 revisions before the final layout was decided upon. This tool comes highly recommended to anyone looking into using a simple communication medium between the Processing System and the Programmable Logic. The limitation of the tool is that it supports up to a register size of 32-bits. If you wanted to configure the Electroporator with very large pulses, e.g. using 64-bit counters, you'd need to split your pulse parameters in software and then concatenate it in hardware. It's not a difficult workaround, but it's extra steps that seem to break the simplicity image of the design.

During the test of writing to the PL, we did not realize that the memory addresses that we were writing to were 64-bits and not 32-bits long. This is because the Ultra96v2 ARM Quad cores are of the ARM 64-bit architecture (AArch64). This did not affect the results as the Zynq was little -endian. However, according to the ARM (2014) reference manual, the processor can store words in memory as either big-endian or little-endian, but instructions are always little-endian. Since little-endian systems are more preferred due to x86, the default configuration of the Petalinux Image was little-endian. This means that the upper half of the registers were empty and not utilized. Now, this is not an issue since we have a lot of memory to spare and we don't need to micro manage the memory usage. If full memory word usage is required, another tool would have to be used to generate the 64-bit AXI-Registers, or the use of a custom solution would have to be looked into.

6.1.5 Development Board Remarks

The Ultra96v2 is quite a popular board due to its price range and the number of features it contains in a small package. Due to this reason, the board is often out of stock from the AVNET resellers. At the early stages of the project, the board was not available from any of the resellers in the EU. As a result, we started developing on a Zynq7000 development board, the Zybo, until the Ultra96v2 was back in stock a few months later. The architecture differences are not too different as they are both System on Chips, with the Ultra96v2 having extra real time cores. The same implementation used for the Ultra96v2 can be used on the Zynq7000 without the graphical interface. The first stage of testing was done on the Zynq7000 then rerun on the Ultra96v2. Since our overall design was not too dependant on specific architecture, no major changes were needed to migrate to another development board. Xilinx (2019c) provides some documentation on migrating from older Zynq7000 series to the Zynq Ultrascale+ MPSoC devices.

6.2 Graphical User Interface Platform

6.2.1 Access from User Space

While we were successfully able to access the memory mapped AXI-Registers through the use of `/dev/mem`, this should not be used as a permanent solution. This method is only for prototyping and testing. The use of `/dev/mem` is dangerous and has a high chance of causing kernel panics if memory outside the AXI-Registers are written to. The proper way to access the AXI-Registers is to use a device driver that can be loaded which will restrict access to only the AXI-Regiters and no

other portion of memory. The device driver can be custom made or the linux `generic-uio` driver can be used. But to get the AXI-Registers recognized as a device, it requires some setup in the kernel as well as configuration in the linux device tree. We learned about this method towards the end of our testing and decided to leave this to the future revisions of this Electroporator design.

6.3 Petalinux with Qt

The Petalinux tool was overall a bit disappointing. The initial experiments with it resulted in bugs and corrupt project files that required a restart. There were several modules listed in the options that triggered errors that failed builds because they were no longer supported or refused to work. The level of fine configurability was lacking, but this seems to a general consensus regarding this tool on Xilinx's support Forum. But the final image that it created worked as intended. However, the results of the software platform were not all successful.

The ability to be able to deploy applications based on the Qt framework was chosen as a requirement for this system because it excels in embedded environments. It is easy to use, prototype, and implement with good source of documentation. We aimed to get Qt Quick working with all of its modules on this embedded linux system. Qt Quick was chosen for its rapid development capabilities as well as its embedded system application features. The first test run was to see if the test application can be deployed. The results were that it failed automatic deployment, but it was able to run manually. Now this is not a fault of Qt, but Petalinux's driver implementation of the xorg server. The first test application launched and everything was working from it. The second test application's deployment failed like the first one, but this application couldn't run. This was because not all of the Qt Quick modules were included in the binary. We tried to resolve this by finding the module in the petalinux tool and adding it, but that module did not exist in the list. The virtual keyboard module test application was chosen because it would inevitably be one of the modules included if the GUI was implemented with touchscreen capabilities in mind.

The Qt binaries from the petalinux tool are outdated. Due to the fixed release schedule of the tool, the binaries are never up to date. We tried several method of getting the QtQuick Virtualkeyboard module working. First was to somehow get the petalinux tool to compile it, but it ended with us just fighting with the tool. Second was to compile it from source, and this failed as well because we couldn't build it due to the qmake binary throwing errors that led us down a rabbit hole to a conclusion saying that the qt binary made by the petalinux tool had holes in it. Petalinux contains package managers, rpm and dnf, but they have no repositories available from which we could install packages. This resulted in us having to manually compile every package that we wanted from source code. We were unable to compile the latest Qt5 from source as we did not know the correct arguments to use during the compilation and some of the build tools were not available in Petalinux, such as the `maven` build system.

We were unable to get the SPI controller working from the embedded linux side. After revisiting this part at a later date towards the end of the project, we found out that we had missed some configurations in order for us to be able to access the spi as a device. The spi controller had to be added to the linux device tree and set that the spidev driver can be utilized for it. A full comprehensive guide on how to accomplish this and getting a working spi controller through linux can be found on `https://www.hackster.io/news/microzed-chronicles-using-spidev-in-petalinux-18ff937807c5`.

6.4 Arch Linux Arm instead of Petalinx

Due to the issues we encountered with the Petalinux Image, we tried using a different system for the root filesystem. Since our host machine was running Arch Linux, we attempted to get the ARM version of the system running on the Ultra96v2. According to ArchLinuxArm (nd), the Ultra96v2 is not officially supported, but there exists a generic filesystem for AArch64 systems. We attempted to get that working by using an unorthodox method of a Petalinux kernel image with the root filesystem from Arch Linux Arm. For Arch linux systems you would typically be building the kernel using their Arch Build System (ABS), but since we already had a working kernel from Petalinux we used that instead. The system was able to boot, fig. 6.1, and after some tweaking to the wireless kernel module of the Ultra96v2 we were able to get internet connectivity. Arch linux is known for its large package repository as well as its community-driven repository, the Arch User Repository (AUR). Arch Linux Arm community was no different, they had a large repository of packages that are up-to-date with the latest relases. Among these packages were the Qt5 libraries, which contained all the latest modules including the Qt Quick Virtualkeyboard module.

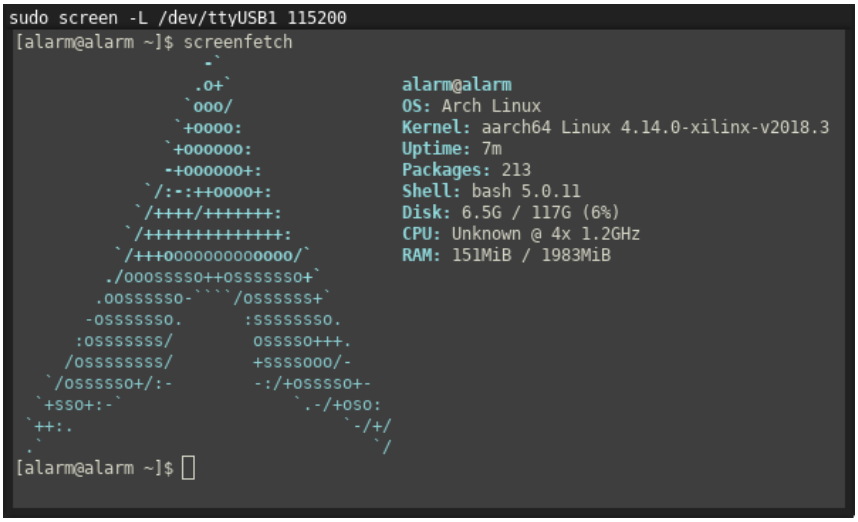

Figure 6.1: Arch Linux Arm running on the Ultra96v2

The reason that we did not include this exploration with Arch Linux Arm was because it yielded no usable results at the end. The issue that we had was getting the xorg server running. Petalinux had some precompiled binary graphics driver, `ARMSOC`, that worked only with Petalinux root filesystem, but trying to use the latest version from the Arch repository did not work and we could not get xserver started. We attempted to compile this driver from source, but no successful results came out of it. We attempted alternative drivers, such as `fbdev` and `vesa` but none seem to work. The resulting error logs can be found on `https://archlinuxarm.org/forum/viewtopic.php?f=65&t=14117`. If we had managed to get this working it would have made developing on the embedded linux of the Ultra96v2 a lot more streamlined due to the abundance of available packages from their repository.

6.4.1 Future Features

6.4.2 Fan Reduction

In comparison to the Zynq7000 series, the Zynq UltraScale+ series has higher power consumption due to it having extra hardware such as the realtime cores, a GPU, and a larger FPGA fabric. As a result, the ultra96v2 comes with a fan that's constantly running. The Zynq Ultrascale+ BGA flip-chip on the Ultra96v2 has no lid, and is exposed to the air with a fan mounted on the bottom of the chassis. The fan is very audible is constantly at max power. By utilizing the Sytem Management IP we can make an automated system that regulates the speed of the fan, `https://www.hackster.io/andycap/ultra96-fan-control-21fb8b`.

6.4.2.1 Data Logging

Figure 6.2 shows a block design of the data logger. Incoming data is received through the SPI and with the help of some custom logic, the raw data from the ADC is forwarded to the Direct Memory Access (DMA) block where it is written to system memory. The User space then collects this data and displays it on the screen or writes it to a file.

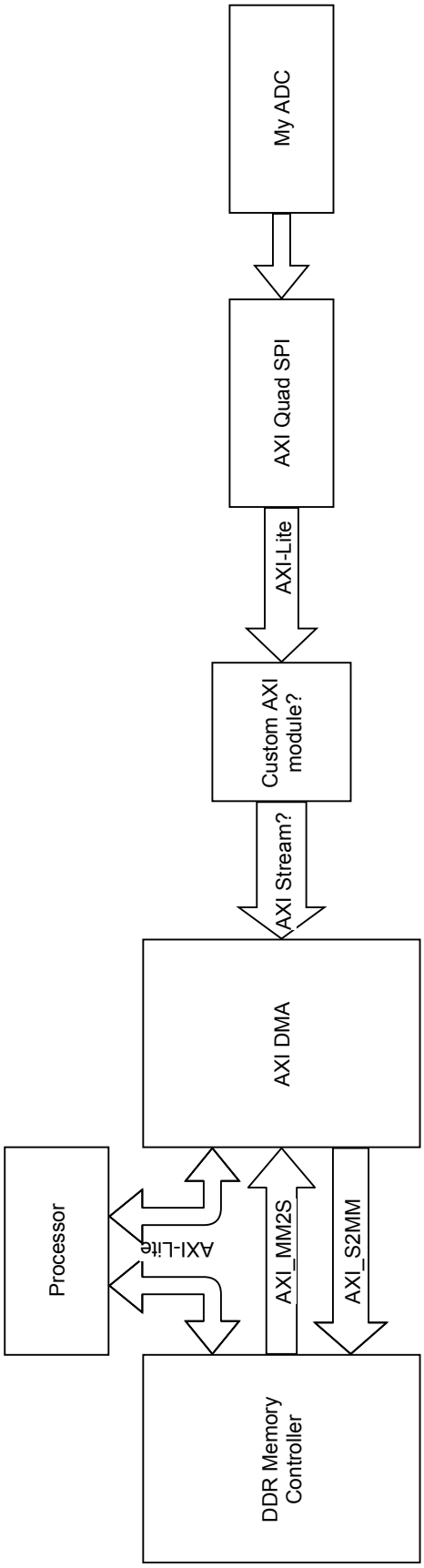

Figure 6.2: Proposed feedback mechanism and data logging

Chapter 7

Conclusion

The purpose of this project was to design a versatile and compact control unit and graphical user interface platform that is aimed towards laboratorial and clinical applications. The commercially available electroporators on the market lack the the ability for fine-tuning parameters and have a limited spectrum of pulses that can be generated. Commercial electroporators are heavy, immobile, and some require permanent installations which creates high costs for maintenance, setup, and operation of the device. To mitigate these shortcomings, a new design was proposed that offers a high degree of configurability and a wide spectrum of pulse generation types in a small form factor.

The control unit of the electroporator was implemented in an FPGA and thanks to the utilization of the structured design method, we were able to achieve 10 nanosecond pulse resolutions. Along with being able to generate mono-polar pulses, we were also able to generate symmetric and asymmetric bi-polar pulses. By utilizing a System on Chip (SoC), we were able to combined two sub-modules of an electroporator that is normally separated, the control unit and the graphical user interface platform. With system modularity being a big requirement, the final outcome of the electroporator is a highly modular and compact system with four different types of generators incorporated into one.

While the core functionality of the electroporator is complete, there are some features that, if added, would greatly increase the quality and usability of the system. These include, but are not limited to, getting a professionally designed intuitive graphical user interface, an automatic safety mechanism that monitors the system's generated pulses, and a logging system that can function as a built in oscilloscope.

Bibliography

ArchLinuxArm (n.d.). *mmap(2) Linux Programmer's Manual*. [Online; accessed 13-December-2019].

ARM (2014). ARM Cortex-A53 MPCore Processor Technical Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0500d/CHDCGHEA.html. [Online; accessed 10-December-2019].

AVNET (2018). Ultra96 USB-to-JTAG/UART Pod. https://www.element14.com/community/docs/DOC-91244/l/ultra96-usb-to-jtaguart-pod. [Online; accessed 10-December-2019].

AVNET (2019a). Board Definitoin File. https://www.element14.com/community/servlet/JiveServlet/downloadBody/92692-102-1-381948/Installing-Board-Definition-Files_v1_0_0.pdf. [Online; accessed 10-December-2019].

AVNET (2019b). Board Support Package. http://downloads.element14.com/downloads/zedboard/ultra96/ultra96v2_oob_2018_3.zip. [Online; accessed 10-December-2019].

AVNET (2019c). Ultra96-V2. https://www.element14.com/community/community/designcenter/zedboardcommunity/ultra96. [Online; accessed 10-December-2019].

Bianconi, E., Piovesan, A., Facchin, F., Beraudi, A., Casadei, R., Frabetti, F., Vitale, L., Pelleri, M. C., Tassani, S., Piva, F., Perez-Amodio, S., Strippoli, P., and Canaider, S. (2013). An estimation of the number of cells in the human body. *Annals of Human Biology*, 40(6):463–471. PMID: 23829164.

Bienkowski, P. and Trzaska, H. (2017). Quantifying in bioelectromagnetics. In *Dosimetry in Bioelectromagnetics*, pages 269–284. CRC Press.

Docker (2019). Package Software into Standardized Units for Development, Shipment and Deployment. https://www.docker.com/resources/what-container. [Online; accessed 10-December-2019].

Ellinger, I. and Ellinger, A. (2014). *Smallest Unit of Life: Cell Biology*, pages 19–33. Springer Vienna, Vienna.

ESA (2019). VHDL. https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Microelectronics/VHDL. [Online; accessed 10-December-2019].

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 171–172. IEEE.

Gaisler, J. (2011). A structured VHDL design method. https://www.gaisler.com/doc/vhdl2proc.pdf, https://www.gaisler.com/doc/structdes.pdf. [Online; accessed 10-December-2019].

Haberl, S., Miklavcic, D., Sersa, G., Frey, W., and Rubinsky, B. (2013). Cell membrane electroporation-part 2: the applications. *IEEE Electrical Insulation Magazine*, 29(1):29–37.

Kotnik, T., Kramar, P., Pucihar, G., Miklavcic, D., and Tarek, M. (2012). Cell membrane electroporation- part 1: The phenomenon. *IEEE Electrical Insulation Magazine*, 28(5):14–23.

Lewis, J. and Training, S. V. (n.d.). VHDL-2008: Why It Matters. https://s3.amazonaws.com/verificationhorizons.verificationacademy.com/volume-8_issue-3/articles/stream/vhdl-2008-why-it-matters_vh-v8-i3.pdf. [Online; accessed 10-December-2019].

Linux (2015). *mem(4) Linux Programmer's Manual*. [Online; accessed 13-December-2019].

Linux (2019). *mmap(2) Linux Programmer's Manual*. [Online; accessed 13-December-2019].

Miklavčič, D., editor (2017). *Handbook of Electroporation*. Springer International Publishing.

Miklavčič, M. (2010). *Advanced Electroporation Techniques in Biology and Medicine*. CRC Press.

noasic GmbH (2019). AirHDL register management done right. https://airhdl.com/. [Online; accessed 10-December-2019].

Pirc, E., Reberšek, M., and Miklavčič, D. (2017). Dosimetry in electroporation-based technologies and treatments. In *Dosimetry in Bioelectromagnetics*, pages 233–268. CRC Press.

Rebersek, M., Miklavcic, D., Bertacchini, C., and Sack, M. (2014). Cell membrane electroporation-part 3: the equipment. *IEEE Electrical Insulation Magazine*, 30(3):8–18.

Xilinx (2014). PetaLinux - Building and Deploying Applications With Qt 4.8.5, QWT 6.1.0, Qt Creator 3.0. https://www.xilinx.com/support/answers/59172.html. [Online; accessed 10-December-2019].

Xilinx (2017). Vivado Design SuiteAXI Reference Guide UG1037. https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf. [Online; accessed 10-December-2019].

Xilinx (2018a). Downloads. https://www.xilinx.com/member/forms/download/xef-vivado.html?filename=Xilinx_Vivado_SDK_2018.3_1207_2324.tar.gz. [Online; accessed 10-December-2019].

Xilinx (2018b). PetaLinux Tools Documentation, Reference Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1144-petalinux-tools-reference-guide.pdf. [Online; accessed 10-December-2019].

Xilinx (2018c). Vivado Design Suite UserGuide (UG973). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug973-vivado-release-notes-install-license.pdf. [Online; accessed 10-December-2019].

Xilinx (2018d). Zynq-7000 SoC Data Sheet: Overview (DS190). https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. [Online; accessed 10-December-2019].

Xilinx (2018e). Zynq UltraScale+ MPSoC DisplayPort Controller - Does the DisplayPort Controller support active or passive adaptors to HDMI, DVI or VGA? https://www.xilinx.com/support/answers/71774.html. [Online; accessed 10-December-2019].

Xilinx (2018f). Zynq UltraScale+ MPSoC Product Tables and Product Selection Guide. https://www.xilinx.com/support/documentation/selection-guides/zynq-ultrascale-plus-product-selection-guide.pdf. [Online; accessed 10-December-2019].

Xilinx (2019a). Memory Recommendations. https://www.xilinx.com/products/design-tools/vivado/memory.html. [Online; accessed 10-December-2019].

Xilinx (2019b). What is an FPGA? https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html. [Online; accessed 10-December-2019].

Xilinx (2019c). *Zynq Migration GuideZynq-7000 SoC to Zynq UltraScale+ MPSoC Devices (UG1213)*. [Online; accessed 13-December-2019].

Xilinx (n.d.). Risk Management for Medical Device Embedded Systems. https://www.xilinx.com/support/documentation/white_papers/wp511-risk-mgmt.pdf. [Online; accessed 10-December-2019].

Chapter A

APPENDICES

Apendix A: Generics.vhd
Apendix B: Generator_Wrapper.vhd
Apendix C: One_shot_latch.vhd
Apendix D: Monopolar_simple.vhd
Apendix E: Monopolar_full.vhd
Apendix F: Bipolar_simple.vhd
Apendix G: Bipolar_full.vhd
Apendix H: State_Mux.vhd
Apendix I: State_Decoder.vhd

FACULTY OF ENGINEERING TECHNOLOGY
GROUP T LEUVEN CAMPUS
Andreas Vesaliusstraat 13
3000 LEUVEN, België
tel. + 32 16 30 10 30
fet.groupt@kuleuven.be
www.fet.kuleuven.be