FPGA-Based Hardware-In-the-Loop Co-Simulator Platform for SystemModeler

Miguel Acevedo Sanz
Master of Science Thesis in Electrical Engineering

FPGA-Based Hardware-In-the-Loop Co-Simulator Platform for SystemModeler

Miguel Acevedo Sanz

LiTH-ISY-EX--16/5013--SE

Supervisor: Mario Garrido
ISY, Linköpings universitet

Leonardo Laguna
Wolfram MathCore

Examiner: Kent Palmkvist
ISY, Linköpings universitet

Division of Computer Engineering
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Copyright © 2016 Miguel Acevedo Sanz
Abstract

This thesis proposes and implements a flexible platform to perform Hardware-In-the-Loop (HIL) co-simulation using a Field-Programmable-Gate-Array (FPGA). The HIL simulations are performed with SystemModeler working as a software simulator and the FPGA as the co-simulator platform for the digital hardware design. The work presented in this thesis consists of the creation of: A communication library in the host computer, a system in the FPGA that allows implementation of different digital designs with varying architectures, and an interface between the host computer and the FPGA to transmit the data. The efficiency of the proposed system is studied with the implementation of two common digital hardware designs, a PID controller and a filter. The results of the HIL simulations of those two hardware designs are used to verify the platform and measure the timing and area performance of the proposed HIL platform.

Furthermore, the possibility to use the proposed system as a co-processor of the software simulator in order to reduce the computation time of certain algorithms is also studied.

This thesis has been performed in collaboration with the company Wolfram Math-Core which is the developer of SystemModeler software simulator.
Acknowledgments

First I would like to thank Wolfram MathCore for the opportunity they gave me to do this master thesis with them and for the great work environment in their office in Linköping. From Wolfram I want to thank specially to my supervisor Leonardo Laguna and to Jan Brugård for their support during my thesis. I also want to give special thanks to my supervisor in ISY, Mario Garrido, for his help and inspiration.
And finally, I want to express my deepest gratitude to my family for all their support during my student years.

Linköping, September 2016
Miguel Acevedo
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>List of Figures</strong></td>
<td>ix</td>
</tr>
<tr>
<td><strong>List of Tables</strong></td>
<td>x</td>
</tr>
<tr>
<td><strong>Notation</strong></td>
<td>xi</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation and purpose</td>
<td>3</td>
</tr>
<tr>
<td>1.2 General requirements</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Problem statement</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Report overview</td>
<td>5</td>
</tr>
<tr>
<td><strong>2 Background and theory</strong></td>
<td>7</td>
</tr>
<tr>
<td>2.1 Background of the HIL simulations</td>
<td>7</td>
</tr>
<tr>
<td>2.2 On HIL simulations</td>
<td>9</td>
</tr>
<tr>
<td>2.2.1 General concepts</td>
<td>9</td>
</tr>
<tr>
<td>2.2.2 Synchronization</td>
<td>11</td>
</tr>
<tr>
<td><strong>3 System design</strong></td>
<td>15</td>
</tr>
<tr>
<td>3.1 General design</td>
<td>15</td>
</tr>
<tr>
<td>3.1.1 Host computer</td>
<td>16</td>
</tr>
<tr>
<td>3.1.2 Interface</td>
<td>17</td>
</tr>
<tr>
<td>3.1.3 FPGA HIL</td>
<td>17</td>
</tr>
<tr>
<td>3.2 Specific requirements for the design components</td>
<td>19</td>
</tr>
<tr>
<td><strong>4 Design implementation</strong></td>
<td>21</td>
</tr>
<tr>
<td>4.1 Host computer</td>
<td>21</td>
</tr>
<tr>
<td>4.1.1 Modifications in SystemModeler</td>
<td>22</td>
</tr>
<tr>
<td>4.1.2 ModelPlug</td>
<td>25</td>
</tr>
<tr>
<td>4.2 Interface</td>
<td>30</td>
</tr>
<tr>
<td>4.2.1 SPI</td>
<td>30</td>
</tr>
<tr>
<td>4.3 FPGA system</td>
<td>34</td>
</tr>
<tr>
<td>4.3.1 General parameters</td>
<td>34</td>
</tr>
</tbody>
</table>
4.3.2 Data handling ........................................... 35
4.3.3 Control ............................................. 37
4.3.4 Synchronization ....................................... 39
4.3.5 Serial-parallel converter ............................... 40
4.3.6 DUT ............................................... 42

5 Results and discussion ................................. 45
  5.1 PID ....................................................... 45
     5.1.1 PID theory .......................................... 45
     5.1.2 SystemModeler implementation .................. 47
     5.1.3 Results ............................................. 48
  5.2 Filter .................................................... 57
     5.2.1 Biquad filter theory ................................. 57
     5.2.2 SystemModeler implementation .................. 60
     5.2.3 Results ............................................. 61

6 Conclusions ............................................ 65
  6.1 Conclusions ............................................. 65
  6.2 Future work ........................................... 67

Bibliography .................................................. 69
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Summarized graph for the model-based-design methodology.</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Simplified diagram of the elements that constitute a HIL simulation.</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Diagram of the main components of a electric motor.</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Diagram for the time step synchronization mode.</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Diagram for the master-slave synchronization mode.</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Elements of the system</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>FPGA system</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Interaction between the host computer elements</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Diagram representation of the three SystemModeler classes.</td>
<td>22</td>
</tr>
<tr>
<td>4.3</td>
<td>Configuration parameters for the FPGA class.</td>
<td>23</td>
</tr>
<tr>
<td>4.4</td>
<td>Configuration parameters for the input and output classes.</td>
<td>24</td>
</tr>
<tr>
<td>4.5</td>
<td>Communication layers of ModelPlug.</td>
<td>25</td>
</tr>
<tr>
<td>4.6</td>
<td>Composition of the transmission packets.</td>
<td>28</td>
</tr>
<tr>
<td>4.7</td>
<td>SPI communication protocol for mode 0.</td>
<td>30</td>
</tr>
<tr>
<td>4.8</td>
<td>Interface set-up</td>
<td>31</td>
</tr>
<tr>
<td>4.9</td>
<td>Receiver FSM</td>
<td>32</td>
</tr>
<tr>
<td>4.10</td>
<td>Transmitter FSM</td>
<td>33</td>
</tr>
<tr>
<td>4.11</td>
<td>FPGA system</td>
<td>34</td>
</tr>
<tr>
<td>4.12</td>
<td>Data handler block</td>
<td>36</td>
</tr>
<tr>
<td>4.13</td>
<td>Control FSM</td>
<td>37</td>
</tr>
<tr>
<td>4.14</td>
<td>Synchronization block FSM</td>
<td>39</td>
</tr>
<tr>
<td>4.15</td>
<td>Serial-Parallel converter block</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>Digital PID</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>PID in SystemModeler</td>
<td>47</td>
</tr>
<tr>
<td>5.3</td>
<td>PID in FPGA</td>
<td>48</td>
</tr>
<tr>
<td>5.4</td>
<td>Virtual and FPGA PIDs comparison</td>
<td>49</td>
</tr>
<tr>
<td>5.5</td>
<td>Variation of configuration parameters in the model</td>
<td>49</td>
</tr>
<tr>
<td>5.6</td>
<td>Variation of configuration parameters in the FPGA</td>
<td>50</td>
</tr>
<tr>
<td>5.7</td>
<td>Effect of the variation of the $T$</td>
<td>51</td>
</tr>
<tr>
<td>5.8</td>
<td>Relation between the total computation time and the number of events.</td>
<td>53</td>
</tr>
<tr>
<td>5.9</td>
<td>Partial times communication</td>
<td>54</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Functions of ModelPlug .......................... 29

5.1 Sampling time effects .......................... 52
5.2 Sample size effects .......................... 53
5.3 Sampling size effects for 16 bits data packets .......................... 55
5.4 Area resources used in the Altera Cyclone II 2C20 FPGA ............... 56
5.5 Area resources used in the Xilinx Spartan 6 LX9 FPGA .................. 56
### Notation

#### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUT</td>
<td>Design Under Test</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In First-Out</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>GPIO</td>
<td>General-Purpose Input/Output</td>
</tr>
<tr>
<td>HIL</td>
<td>Hardware-In-the-Loop (simulation)</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Action Group</td>
</tr>
<tr>
<td>MAC</td>
<td>Media Access Control (sub-layer)</td>
</tr>
<tr>
<td>MISO</td>
<td>Master Input - Slave Output</td>
</tr>
<tr>
<td>MOSI</td>
<td>Master Output - Slave Input</td>
</tr>
<tr>
<td>PCI</td>
<td>Peripheral Component Interconnect</td>
</tr>
<tr>
<td>PID</td>
<td>Proportional, Integral, Differential (regulator)</td>
</tr>
<tr>
<td>RAM</td>
<td>Random-Access Memory</td>
</tr>
<tr>
<td>RX</td>
<td>Receiver</td>
</tr>
<tr>
<td>SCLK</td>
<td>Serial Clock</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial-Parallel Interface</td>
</tr>
<tr>
<td>SS</td>
<td>Slave Select</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>TX</td>
<td>Transmitter</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>UUT</td>
<td>Unit Under Test</td>
</tr>
</tbody>
</table>
Electronic devices form a big part of current technology nowadays. Since the invention of the transistor, the use of electronics is growing in an extraordinary manner. More specifically, digital electronics are becoming more and more popular among designers because of its flexibility, reliability and easiness to design. However, one of the barriers for these systems is how to test and verify them. The testing phase for digital electronics and electronics in general is costly and time consuming. This costly and time consuming process is a problem in modern day industry where increased competition between global companies create a need for short product development cycles. The most common solution is the simulation of electronic systems using virtual mathematical models. These simulation models are widely used for validation and verification of electronic systems during development phases. Nevertheless, simulations using virtual models produces two issues: First, it could be unfeasible for complex electronic designs, where it could be difficult to create a valid mathematical model for the electronic component developed. Second, the behavior of the virtual model could differ from the real implementation. This difference, even if is small, could cause a loss of accuracy during the verification process.

A method developed to solve these issues are the hardware-in-the-loop (HIL) simulations. These simulations consist of the addition of one more element to the simulation loop; a real hardware component. In other words, hardware-in-the-loop simulations are a co-simulation of two main elements: A physical hardware component and a mathematical model that represents the complete system which the hardware component is part of. The integration of a real component in a simulation model gives many advantages to the testing phase of a design. For example, this type of simulations increase the quality of the test by integrating the real hardware component. The results from a simulation are always more
realistic when using the real physical component than a virtual model. Additionally, HIL simulations give a safe platform to perform tests and verification of a system. This is especially useful for controllers of critically safe systems where a malfunction can cause the loss of the complete system. HIL simulations are the best solution for these cases, the real controller can be tested in a reliable manner without the need of risking the complete physical system by using a virtual model of that complete system. Other benefits of using HIL simulations during the design and testing phases is the improvement of the project development times. When developing a complex system that is composed of several components, HIL simulations allow the development of those sections in a parallel way. This is because the different sections can be developed and tested with models of their counterparts without the need of having them already finished. Then, using HIL simulations, the iteration time and costs between different models is highly reduced. All these reasons make the HIL simulations a suitable validation test-bench for digital designs.

Nonetheless, even with the use of hardware-in-the-loop simulations during the development period, the time to create an electronic digital design is still long. When creating and testing digital designs, specially for Application-Specific-Integrated-Circuits (ASICs), the main bottleneck is the fabrication of the prototype. The fabrication of an ASIC is costly and time-consuming, and costs and time are increased if several iterations are needed during the developing period. A solution for these time and cost problems during development are Field-Programmable-Gate-Arrays (FPGAs). A FPGA board is a configurable integrated circuit that can be reprogrammed with different digital designs. The FPGAs are formed by configurable logic elements that are connected with programmable interconnections. The rearrangement of these elements and the interconnections allows the implementation of different digital designs in a fast and simple way. To perform this configuration of the FPGAs are usually used Hardware Description Languages (HDL) such as VHDL or Verilog. FPGAs allow to test and implement the different iterations during the design phase without modifying the hardware, which reduces time and cost problems of having to implement many prototypes during the developing period. The FPGAs are, therefore, an ideal platform for the implementation and test of digital designs and are commonly used for HIL simulations of digital electronics.
1.1 Motivation and purpose

Although hardware-in-the-loop simulations are very useful for testing and are commonly used, they are usually created ad-hoc for the specific system that is going to be tested [1]. The creation of such a specific HIL simulation for a specific target system can be quite time consuming. Moreover, the creation of individual set ups for HIL simulations for these systems is not a simple task. This difficulty for the implementation makes HIL simulations a very hard task, specially for beginners and low-experienced designers, even regarding the most simple designs.

This thesis proposes a test platform to perform hardware-in-the-loop co-simulations using an FPGA board and SystemModeler software simulator. The solution is a generic HIL simulator platform that allows to take an existing design and connect it with the simulation environment in order to perform HIL simulations for verification. Then, the purpose of this thesis is to present a flexible and accessible test platform that allows to implement HIL co-simulations of different real digital components in a fast a relatively simple manner. This simplicity is also obtained through the use of the FPGAs, which admit to synthesize and test different digital designs with the same hardware.

SystemModeler, which is the software simulator used for this thesis, is a modelling and simulation environment for cyber-physical systems [2]. SystemModeler is based in Modelica language which is a object-oriented language based in equations used to create models of physical systems. This type of object-oriented simulator simplifies the representation of the FPGA based HIL platform, as a part of the complete simulation model. In addition, the model creation environment and the simulation environment of SystemModeler are quite straightforward to use even for novice users with little experience in digital design.

A secondary motivation for this thesis is to implement the FPGA as a co-processor for the software simulator. Certain algorithms with big computational loads could be very slow to calculate for normal computer processors. Therefore, the implementation of those algorithms in an external hardware accelerator could be helpful to reduce computation times. For this purpose, the good performance of FPGAs with highly parallelizable designs make them good candidates for running these algorithms.
1.2 General requirements

The main motivation of this thesis is to provide a platform that allows to easily implement HIL simulations. Such a platform needs to be very adaptable in many senses, even more when taking into account that the implementation of hardware designs in FPGAs are not as flexible as a software implementation. Additionally, Wolfram has a wide spectrum of users that ranges from novices (like students) to professionals. Therefore the usability and flexibility of the proposed platform are the main requirements for this thesis. The system should be simple to use and modify, even for users with little experience. However, as the HIL simulations are used for validation and verification, precision of the results is also a must.

Then, there are several aspects that need to be addressed to obtain this adaptability for the platform:

1. The proposed HIL system should have a low area usage. The reason behind this is to be able to fit in a wide range of FPGAs of the market and be as non-intrusive as possible to the user design.

2. The system should be able to accept several data types and sizes for the digital design that is being tested. Additionally, it should be simple to change the sizes or the types of the data samples without needing to modify the the proposed solution in depth.

3. Digital designs are not defined by a fixed number of inputs or outputs. Hence, both the simulation platform in the FPGA and the simulations models in SystemModeler need to adapt to those design variations.

4. As the design intends to broaden the use of the HIL simulator, the usability is very important. The interfaces supported should be available for most of the FPGAs on the market.

5. The system should be as independent of the interface as possible. The system should be designed in a modular fashion allowing the modification of the communication interface used. This modular design should also be obtained for other design elements outside the interface components.

The speed of the system is a second order requirement for this simulator, the main purpose, as previously commented, is the flexibility. However, the solution proposed needs to perform in acceptable values for the total simulation time.
1.3 Problem statement

Aside the implementation of a co-simulation platform to perform hardware-in-the-loop simulations, which is the main motivation for this thesis, the work will be used to answer five questions. These questions are:

1. Is the system flexible enough to perform different hardware-in-the-loop simulations without main changes?
2. More than the possibility to simulate the real system, does the platform give any other advantage with respect to software?
3. Which synchronization modes are found useful for the co-simulation?
4. Which speed rate and transfer rate are needed for this purpose?
5. Is it possible to use the system as a co-processor to optimize software simulations of heavy algorithms?

These questions will be discussed in the conclusions, chapter 6, of this report.

1.4 Report overview

The order of the chapters in this thesis and their content is briefly explained in the following section.

Chapter 2 contains a summary for the hardware-in-the-loop simulations. It starts giving a small view of the background found in literature for the HIL simulations and it continues with a short theoretical overview of the concepts for HIL simulations.

Chapter 3 gives a high level description of the elements that compose the system proposed in this thesis, with their uses and their individual requirements.

Chapter 4 contains the main explanation for the implementation of the proposed system. A thorough explanation of the design of all the elements that compose the finally implemented system is given in this chapter.

Chapter 5 presents the results obtained from testing the system. Two models are implemented to perform the testing. Theoretical explanations for those models together with a description of their implementation in SystemModeler are also given in this chapter.

Chapter 6 introduces the final conclusions obtained from the thesis and gives suggestions for future work within the proposed system.
2.1 Background of the HIL simulations

The simulation of electronic systems using hardware-in-the-loop co-simulation has been a common practice for many years. The origins of hardware-in-the-loop simulations are based in the aeronautic industry [3], where some of the first appearances consisted in flight simulators with simulated instruments. More early approaches to hardware-in-the-loop simulations were performed in the military industry and in aerospace research [1]. Modern applications of HIL simulations used in aeronautics are, for example, control systems for unmanned aerial vehicles [4] and in NASA's aerospace research [5].

Hardware-in-the-loop simulations have become very popular in the last decades with the integration of electronic systems in every day technology. The strong competition between companies' product development, in addition to the widespread use of electronics systems in almost every modern technological industry, has extended the use of HIL simulations to very different areas.

HIL are very popular in the automotive industry. One cause is that in modern cars, the electronic systems form a big part of the car components. Almost every electronic or electric main component of a vehicle is tested by using hardware-in-the-loop simulations. One of the first approaches of HIL for automotive systems is based on the design of diesel engine control systems [3]. Other automotive systems tested are, for example, Anti-lock Brake Systems (ABS) [6, 7, 8], suspension systems [9], or the general control system [10].
Other fields where HIL simulations are widely used are power electronics and electrical systems. Several works in this area used an FPGA board in the simulation loop [11, 12, 13, 14]. Power electronic systems are usually divided into a controller, a power converter, and the plant under control. There are different methods for applying HIL simulations to power electronics regarding how the sections of the system are implemented. On the one hand, for example in [13] the controller is implemented using a DSP meanwhile the FPGA is used to implement a model of the power converter. On the other hand, in [15] the controller is implemented in the FPGA and the model of the system runs in a PC. As a final case, [12] proposes a solution with both the controller, using a soft-processor, and the power converter implemented in the FPGA.

Further domains where HIL simulations are extensively used are the automatic control systems [16, 17], more specifically when digital controllers are involved. In [18] a procedure to design digital controllers using hardware-in-the-loop is proposed. In this case the implemented element is a Proportional-Integral (PI) controller.

Nonetheless, hardware-in-the-loop simulations are not only restricted to these areas. HIL simulations could be implemented in any type of electronic or mechanical system which contains sections that could be modeled in a mathematical simulator. There are more varied examples, as underwater vehicles [19], robotics [20, 21] or even traffic control systems [22], which show obvious potential for these type of simulations.

There is plenty of literature regarding hardware-in-the-loop; however, this literature is usually very specific to the field that is being tested. There are not many papers in literature that review the HIL problem from a more broad perspective. This problem is also commented in [23] where Harrison et al. propose an ontology for manufacturing systems in which HIL simulations are a part of. Their presented ontology integrates HIL in a more generalized Hybrid-Process-Simulation (HPS) with different levels of systems implemented. Another paper that presents a more general perspective is [1]. In this paper, Bacic identifies that there are no general guidelines for the measure of robustness when comparing HIL simulations. Bacic et al. have more research based in this focus of measuring the accuracy of different HIL simulations in [24].
2.2 On HIL simulations

2.2.1 General concepts

The common explanation for in-the-loop simulations is that they consist in the addition of one more element to the simulation loop. Then, for the case of Hardware-In-the-Loop (HIL) simulations a real hardware component, part from the complete physical system, is integrated to the simulation loop [1]. In other words, a HIL simulation consist in the co-simulation of a real hardware element together with a mathematical modeled system. Additionally, there are other in-the-loop simulations such as Software-In-the-Loop (SIL), Model-In-the-Loop (MIL), and Processor-In-the-Loop (PIL) simulations, but they are not considered in this thesis.

Regarding terminology, the term hardware-in-the-loop simulation often refers to the complete set up of both the modeled system and the real hardware system. Additionally, HIL system or HIL platform can refer [12, 20] to the test platform where the hardware component is tested. This HIL platform includes the mathematical model used in the simulation and the actual test-bed where the hardware is implemented. An additional name used for the simulated system is plant under control, this name is commonly used when the simulated system is a controller mostly for power electronic or automatic control systems. Moreover, the real hardware system is often referred as Device-Under-Test (DUT) or also as Unit-Under-Test (UUT). In control systems, the device-under-test is also named as Electronic-Control-Unit (ECU). In this thesis the term HIL system will refer to the test-bench that includes the mathematical model meanwhile HIL simulation will refer to the complete simulation system that includes the hardware component. To refer to the hardware component the term device-under-test or DUT will be used.

![Figure 2.1: Summarized graph for the model-based-design methodology.](image-url)
Hardware-in-the-loop simulations are usually one of the last steps of the Model-Based-Design methodology. Model-Based-Design is a methodology to design control systems [25]. A V diagram example is depicted in figure 2.1 where design phases are shown in the left and testing phases in the right side. In this type of methodology HIL simulations are regarded as a good platform to perform verification test of a hardware component during the integration testing, and they are especially useful when the hardware component is a digital controller. This formal verification and validation allows to test the response of the hardware algorithm to different sequences of inputs.

![Figure 2.2: Simplified diagram of the elements that constitute a HIL simulation.](image)

Figure 2.2 shows a simplified diagram of a common HIL simulation set up. As explained before, these set ups consist of two main parts: The mathematical model of the plant under control and the DUT. These two elements are connected in a close-loop where the outputs of one are the inputs for the other one and they provide feedback to one another. On the one hand, the mathematical model of the plant implemented in the HIL model generates the simulated sensor signals that feed the device-under-test through a physical I/O port. On the other hand, the device-under-test reacts to these sensor signals and provides a response that is sent back to the HIL model using the physical I/O port. This feedback provided from the DUT to the model is then applied to the simulated actuators in the HIL model which will simulate the response of the physical system. The last element in the HIL simulation environment is the test system, this test system provides the test cases for the HIL simulation environment and compares with the results of the simulation.

However, the set up presented in figure 2.2 is a simplified version. In existing literature, there are more complex implementations described depending on the application. The main problem regarding hardware-in-the-loop simulations is that these simulations are usually created ad-hoc for a specific application. Therefore, there is not much literature regarding HIL simulations from more broad points of view, the literature is rather more focused in the specific areas where the HIL
2.2 On HIL simulations

Simulations are used.

As a more specific example, in [26] Bouscayrol presents different HIL simulation types that exist when referring to HILs for power electronic systems and electrical machines. Moreover, he also suggests names for them. Electrical machine systems are usually composed by several sections: The controller, the power electronics system (that could be, for example, any kind of power converter), the electrical machine, and the load that is driven by that machine. These components are shown in figure 2.3. Then, according to Bouscayrol, the HIL simulation types could be defined by the number of real components that are implemented in the loop. The first type, the simplest HIL simulation, is the “Signal level HIL simulation” where only the controller is implemented as a DUT and the rest of the components are modelled in the simulation. The second type is the “Power level HIL simulation” where to the hardware controller is also added the hardware version of the power electronic system. The third and last one is the “Mechanical level HIL simulation” where the real electrical machine is added to the DUT and only the load of the system is a virtual model.

2.2.2 Synchronization

As hardware-in-the-loop simulations consist in two elements interacting, the synchronization of the communications is important. This is even more important since HIL simulators are very commonly used for the verification and validation test of real-time hardware controllers. The validation tests for critically safe physical systems have to be reliable in its operations over time. This means that the results need not only to be correct but also to be given correctly in the time domain. For these reasons the HIL simulations are often considered as real-time simulations. A real-time simulation is considered a simulation where the time spent in the simulation coincides with the real time that passes in a clock.

The synchronization between elements in the real-time type of simulation is often performed with a time stepped synchronization [27]. In this synchronization, the total simulation time is divided in smaller time steps. The duration of these time steps is defined by a sampling period, in other words, the time steps occur every certain sampling time. Then, if a real-time simulation is performed, the
sampling period and the operation of the different elements of the system need to be synchronized with the real time of the clock.

This time stepped synchronization for the real-time simulations is also used in HIL systems [3, 10, 15, 28, 29, 14, 30, 18]. Although the time stepped synchronization is of common use in HIL simulations, there is not a generically defined sampling time. This is due to the varying nature of the hardware elements tested in HIL simulations. In [3], which contains a deep study of HIL systems for engine control systems, an acceptable sampling time for controllers of about $T_{0c} = 1 - 10\, ms$, i.e., a sampling frequency of $f_{0c} = 100\, Hz - 1\, kHz$. However, these times are not valid for every application, in [18] an upper limit for their sample time is defined as $2.1\, ms$ and for their results a value of $250\, \mu s$ is used. Therefore, the value for the sampling periods needs to be determined specifically for each application.

The main synchronization model used for this thesis is based on this type of controlled time step simulation. A scheme for this synchronization is shown in figure 2.4. It consists of fixed synchronization times, or sampling times, when the information between the FPGA and the software simulator are interchanged. This fixed time works in a similar manner as the synchronization scheme 1 in [31]. This sampling time is configured at the beginning of the simulation in SystemModeler’s simulation model and remains the same for the rest of the simulation. In this model, both the simulator and the FPGA board are running in parallel.

![Diagram for the time step synchronization mode.](image)

The communication process observed in figure 2.4 is: In $T_0$ the initial communication is performed. The simulator sends the initial values for the first iteration to the FPGA and, as it is the first communication, the values coming from the FPGA are discarded. When the communication is finished, the simulator and the FPGA start running in parallel. As the FPGA is faster, it will enter in an idle state when it is finished. The time that the FPGA will run is notated as $T_{step}$ and can be configured. When $T_1$ is reached, the second communication takes part, the FPGA
returns the results from \( T_0 \) and the and the simulator sends new values. After the communication, the simulator and the FPGA start running again. This synchronization means that the results are delayed one sampling time. For example, the FPGA receives its input signals in time \( T_1 \) for the second iteration, but it will give the result for those inputs in the next synchronization time \( T_2 \).

![Diagram for the master-slave synchronization mode.](image)

**Figure 2.5:** Diagram for the master-slave synchronization mode.

A second synchronization model of interest for this thesis is when the FPGA is working as a co-processor. The graph for this communication is shown in figure 2.5. The synchronization for the co-processor mode is very similar to the synchronization for the co-simulator. This synchronization is based in a master-slave relation, with the FPGA working as a slave. The main difference with the co-simulation version is the order of the communications. In the previous case, there are fixed communication times (the sampling times) when the data is interchanged both ways. However for this case, the communications are separated in the transmission of samples to the FPGA and afterwards the transmission from the FPGA. In this synchronization model the software waits until the hardware have finished its operation, the synchronization is based in synchronization scheme 2 of [31]. The simulator decides when the FPGA have to start to work and sends the data samples for that. Afterwards, when the FPGA is finished it sends the result data samples back to the computer. During the time the FPGA is working, the simulator is in stand-by waiting for the data samples. The \( T_{step} \) for this synchronization model is still defined in the simulator side and configured at the beginning of the simulation.
This chapter introduces the proposed system, which includes an overall description of the general design, the different parts that compose it and the interconnections between these parts. Additionally, a justification for the use of these parts is given during their description. To finalize the chapter, the specific design requirements for system’s elements are explained. Further technical explanations regarding the design elements are given in chapter 4.

3.1 General design

The solution proposed in this thesis presents a platform based in an FPGA to perform hardware-in-the-loop simulations. The system has three main components: The host computer with the simulator, the FPGA board, and a communications interface. To clarify the purpose of these components, a comparison with the common HIL set up explained in chapter 2.2, and especially with image 2.2, can be done. The first element, the host computer that includes SystemModeler simulator, will contain the HIL model. Then, SystemModeler, with the help of the interface, provides the simulated sensors to the DUT which is implemented in the FPGA. Thereafter, when the DUT in the FPGA has produced a response, this response is sent back to SystemModeler using the interface. Additionally, if there are test cases for the simulation, these can be implemented in SystemModeler as well.
Figure 3.1: High level schematic of the elements of the system.

Figure 3.1 contains a more detailed explanation of the different elements of the proposed platform. The figure shows all the components of the platform, however, during this thesis there were only created the elements coloured in Grey. Additionally, two examples are presented in this thesis for the DUT, light blue on the picture. The following sub-sections contain a description of these elements and their purpose.

### 3.1.1 Host computer

The first part on figure 3.1 is the host computer. The main element in this side is the SystemModeler simulator; SystemModeler’s main purposes are to create the simulation model, control the simulation flow, and display the results. The problem then is how to represent the FPGA, or the DUT, in this simulation model. To solve this, a group of classes written in Modelica are used. These classes represent the FPGA, the inputs, and the outputs of the digital design. These classes proportionate additionally a good platform to modify the configuration parameters for the simulation. Once the DUT is represented in the simulation model, this representation has to be connected with the actual DUT implemented inside the FPGA. The ModelPlug library, the second biggest element on the host computer side, is in charge of this connection; it prepares the simulation data and creates the transmission data packets. Lastly, an issue arises with the use of different languages. SystemModeler uses Modelica language and ModelPlug is written in C++. Therefore, a group of extern C functions are used. Modelica supports to call extern C functions, so SystemModeler invokes these functions which in turn invoke their counterparts in ModelPlug to perform the final communication. The work during this thesis in this side of the platform regarded the creation of the Modelica classes, the ModelPlug library based on a previous existing model of ModelPlug, and the extern C functions.
3.1 General design

3.1.2 Interface

The interface is the second main block of the proposed system, it creates the connection between the FPGA HIL system and the host computer. It is represented in figure 3.1 between the ModelPlug library and the FPGA HIL system. A normal interface is composed by a physical connection and two protocol decoders, one on each side of the physical connection.

In this thesis the SPI protocol has been used for the interface. However, the host computer is not able to produce this SPI protocol, therefore an Arduino board is used. The software protocol decoder placed in the host computer transmits using the USB serial port of the computer to the Arduino. Thereafter, the Arduino transforms that transmission to the SPI protocol and sends it to the FPGA. In the FPGA side the second protocol decoder block is implemented and is used to read this SPI transmission.

The work in this side of the platform during this thesis consisted in the creation of the hardware protocol decoder in the FPGA and the programming of the Arduino board.

3.1.3 FPGA HIL

The third element shown in figure 3.1 is the FPGA HIL system, this element contains the DUT that is simulated. The purpose of the FPGA HIL system is to prepare the data signals for the DUT and to control the simulation run time in the FPGA. To be able to perform this, the FPGA HIL system is composed by five blocks: A data handling block, a control block, a synchronization block, a serial-parallel block and the DUT. A schematic with the composing blocks of the system is shown in figure 3.2.

In figure 3.2 all the components implemented in the FPGA are represented. It includes the protocol decoder, which is in fact part of the interface, the DUT, and the FPGA HIL system. In the following paragraphs the data flow through the FPGA will be followed to explain the different components.

The first block in the FPGA that receives the information from the interface is the hardware protocol decoder. As explained before this block is used to recover the incoming SPI signal and sends it to the data handler block.

The second block, the data handling block, has three main purposes: First, to store the data samples until they can be processed. Second, to make the FPGA HIL system more independent from the interface used. This is obtained by only allowing this block to interact with the interface and minimizing the connections of the data handling block with the interface. The third purpose is to perform data synchronization; the FPGA and the SPI decoder use different clock domains that need to be synchronized.
The next block in the data flow is the control block. The control block is in charge of reading all the commands sent by the software simulator and setting the start of the simulation inside the FPGA. When a new transmission arrives, the control block reads the first data value, which is an operation command, and sends the corresponding control signals to the rest of the system.

The following block is the synchronization block. To run the DUT each sampling period in the FPGA needs of three main phases: the load of the data, the run of the time step, and the acquisition of the results of the current sampling period. The synchronization block is in charge of processing these three phases at the correct moment.

The lasts two blocks in figure 3.2 are the serial-parallel block and the DUT itself. The purpose of the serial-parallel is to provide the inputs for the DUT to simulate. Most of HW designs are composed by several inputs or outputs, therefore, the need for this block is due to in the necessity of transforming the serial communications to this parallel nature. This block can be divided itself into two smaller sub-blocks: A serial-to-parallel block that performs that transformation at the input of the DUT and a parallel-to-serial block at the output of the DUT.

The whole FPGA side of the platform has been created during this thesis. For the DUT, two examples are provided in chapter 5.
3.2 Specific requirements for the design components

This section explains how the main requirements, explained in section 1.2, affects the different components of the proposed solution.

The major requirement regarding the host computer part of the thesis is the usability of the system because SystemModeler is the main element that the user will interact with. The process to create and use a new simulation model needs to be simple; simple to understand from a user point of view how the DUT in the FPGA is represented in the simulation model and how to configure it. Additionally, the general requirement concerning the data types is important in the host computer. The ModelPlug library is in charge of all the data transformations that are needed for the HIL platform. Moreover, the configuration of these data types needs to be straightforward for the user.

For the interface, requirement number five in section 1.2 is important: The interface should be available for most FPGAs in the market. Since the proposed platform targets a wide range of users, the interface needs to be highly compatible. On the host computer side, the interface should not have compatibility problems with different OS. This is also important taking into account that SystemModeler works for Windows, Linux, and Mac. Additionally, the interface should be relatively simple to implement in a new FPGA design. Another basic consideration for an interface is the communication speed. For use of the co-simulator, the transmission speed is important, but for uses as a co-processor the communication speed is critical. However, most high-speed interfaces are not available for low or medium end FPGAs, which enters in conflict with the first requirement for this part of the design.

The requirements that concern the FPGA side are based on the flexibility of the platform. First, the FPGA HIL system that controls the simulation of the DUT needs to occupy the smallest area possible. This is necessary to widen the range of FPGAs that can be used for the HIL platform. Second, the system has to be flexible with the DUT architecture. Digital designs that can be implemented as DUTs does not have a fixed architecture (they vary on degree of parallelism, number of inputs/outputs, size or type of those inputs/outputs). The platform needs to adapt to the different architectures that digital designs can have. Moreover, it should be simple, from the user perspective, to configure the FPGA platform for different DUT architectures. Third, the FPGA system needs to be designed in such a way that the implementation of new functionality is simplified. The proposed solution is a first version of a HIL platform for SystemModeler, therefore the platform should allow to implement new functionality in the future.
This chapter contains a more detailed explanation of the technical implementation of the different parts of the design. This chapter is divided in three main sections: The first one contains the explanations of the host computer side of the thesis, the second section on the interface, and the third and last section on the FPGA HIL system.

4.1 Host computer

The work in the host computer side involved mostly the modifications of the existing ModelPlug library and the connection of this library with SystemModeler through three Modelica classes. Figure 4.1 shows the schema of the interaction between the host computer elements. The two main elements shown in the figure are the simulator SystemModeler and the communication library ModelPlug. The third element shown in the figure are the extern C functions. As explained in the previous chapter, these functions are used to interconnect the Modelica code from SystemModeler with the C++ code of ModelPlug. The three Modelica classes in SystemModeler call their respective extern C functions which in turn call the ModelPlug C++ functions. In the figure the direction of the data transmissions between the functions is depicted with the arrows. Actual simulation data is sent to and from the FPGA with the SignalToFPGA and SignalFromFPGA groups of functions. The group of functions of updateBoard are used to transmit configuration data and actualize the sampling periods.
4.1.1 Modifications in SystemModeler

The main purpose of the SystemModeler side is to create the simulation environment and control the simulation. The simulation model in Modelica is performed interconnecting blocks that represent the different functionality. Therefore, some of these blocks have to be created to represent the DUT implemented inside the FPGA board in the simulation. These blocks are the three Modelica classes created in SystemModeler; these classes are shown in image 4.2 and represent the FPGA itself, the inputs, and the outputs of the DUT.

Additionally, these classes can be used for two additional purposes: To call the extern C functions previously mentioned and to define the configuration parameters for the host computer.
FPGA class

The first class created is the FPGA board. This FPGA board class is implemented inside a bigger class that comprehends the different boards that ModelPlug allows to connect. This class is additionally used to call the updates for the data samples at every sampling time. In other words, when a sampling point is reached the FPGA board class starts the `updateBoard` functions that will actualize the simulation values to send to the FPGA.

![Image of configuration parameters for the FPGA class.](image)

**Figure 4.3**: Configuration parameters for the FPGA class.

Regarding the configuration parameters for the FPGA class, an image of them is shown in 4.3. The configuration parameters for this class are:

1. The communication port that SystemModeler is using. During this thesis the communication ports in the host computer side are serial ports.

2. The `showPinCapabilities` parameter is not used for the FPGA communication. The version of ModelPlug for the FPGA created during this thesis does not take care of them.

3. The `useDTR` is not used either for the FPGA communication.

4. The sampling period specifies how often the communications with the FPGA board are performed using the `updateBoard` function.

5. The baud rate used for the communications. This baud rate is used for the serial communication in the host computer side.

6. The internal FPGA clock frequency in Hz.

7. The number of time steps to run inside the FPGA board. This number of time steps is specified in clock cycles. By define the value used is the sampling period.
8. The data width used inside the FPGA and in the interface. However, this parameter does not actually set the data width inside the FPGA (this is performed by other parameter specified in a VHDL package in the FPGA system code), this parameter is used only for configuration purposes in the host computer side.

**Input and output port classes**

The second group of classes created represent the inputs and outputs of the DUT. They are called `signalToFPGA` and `signalFromFPGA` and are homonym to their corresponding functions in ModelPlug and in extern C. They are used to send and receive data to ModelPlug every sampling interval after the stored values are updated by `updateBoard`.

![Figure 4.4: Configuration parameters for the input and output classes.](image)

These two classes have three different configuration parameters, shown in figure 4.4:

1. The pin number, which only function is to order the inputs and outputs in the software side of the platform. The input and output pins are sent through the interface from the smallest to the biggest. This means that the pin numbers have to be set accordingly with the architecture of the DUT.

2. The `signalType` is used to determine the data type that will be sent to the FPGA. The parameter specifies both the word length of the signal (it accepts 8, 16, 32, and 64 bits) and if it is signed or unsigned.

3. The number of fractional bits the number has. However, if the number is an integer, setting this parameter to 0 is enough.

The main purpose of these configuration parameters, especially the second and the third, is to define the data types used in the FPGA.
4.1.2 ModelPlug

Once the simulation model is created and the configuration parameters for the software have been defined in the SystemModeler side, the problem left is to organize these configuration parameters and the data samples so the FPGA HIL system can understand them. The communication from ModelPlug to the FPGA is performed with unsigned data packets of a fixed size, therefore the data received from the simulator needs to be translated in this library. Additionally, as commented in section 3.2 the FPGA side needs to receive the operation commands from the host computer, these operations commands are created in this library.

The communications performed by the ModelPlug library can be divided in three layers. A top layer that interacts with SystemModeler, a medium layer that operates with the data, and a low layer that interacts with the serial interface. A diagram of this layer implementation is shown in figure 4.5.

![Diagram of Communication layers of ModelPlug.](image)

The centre of the design of the ModelPlug library are two buffers: pin_in and pin_out. These buffers are used to store the values of the data signals for, respectively, the input and the output. Those buffers are accessed to store or read data by the top layer and the medium layer of the ModelPlug library.

In this subsection only the functions of the ModelPlug library that have been created or highly modified during this thesis are explained.
Top layer

The top layer of ModelPlug acts as an intermediary between SystemModeler and ModelPlug. These functions, that are called `signalToFPGA` and `signalFromFPGA`, are used to solve the data type differences between the simulator and the FPGA. The simulator side uses floating point data types meanwhile the most common in the FPGA designs are fixed points data types. This data transformation is performed in three phases:

1. The incoming signed data samples of type double are multiplied with $2^{\text{no. of decimals}}$, in this way the resulting value is the original scaled with the number of fractional bits desired.

2. The resulting sample is stored in a `uint64_t` data type, type that is used in the ModelPlug library. The conversion to `uint64_t` is done using pointer referencing $^1$ to store the information bits of the data samples. In this way, the sign is saved even if the data type used is an unsigned like `uint64_t`.

3. The converted `uint64_t` data sample, already with the information in fixed point values, is stored in the `pin_in` buffer for the next layer to use.

The three steps are very similar for the data received from the FPGA, they are however inverted:

1. The data received is already stored in the `pin_out` buffer, but this data is stored as `uint64_t` so the sign needs to be recovered. The samples received are stored in `uint64_t` types even though they contain actual signed data. To recover a signed value stored in an `uint64_t` data type the function needs to: First, know the real size of the sample. This is because the function needs to take the correct bits of the `uint64_t` sample, including the sign bit, and discard the rest. Second, perform pointer referencing, in a similar way that in the sent data, to recover the sign.

2. Once the data sign is recovered the sample is divided by the $2^{\text{no. of decimals}}$ to scale down the values. This scaled down value is stored in a double data type, hence recovering the floating point fractional values that SystemModeler uses.

3. The last step is to send these recovered data samples, using the external C functions, to the SystemModeler simulator.

$^1$First, the signed double value is stored in an `int64_t` sample. Second, an `int64_t` pointer is created pointing to the address of new `int64_t` sample. Third, a second pointer, in this case unsigned `uint64_t`, is created referencing to the first pointer. Fourth, the last step is to create a `uint64_t` variable were is stored, using the second unsigned pointer, the original double value. In this way the original signed value is stored in an unsigned variable.
Intermediate layer

The intermediate layer contains the pin buffers and the `updateBoard` function. This `updateBoard` function is invoked by the simulator every time the simulation reach a sampling point. The `updateBoard` function is used to: First, actualize the information contained in the pin buffers every sampling period. Second, prepare for the lowest layer the data contained in these buffers. An extra thing that needs to be taken into account is the size of the packets that the interface can handle (for example, the SPI interface uses packets of size either 8 or 16 bits). The `updateBoard` function divides the total sample to smaller packages of the interface size by shifting the original value. Afterwards, these values are stored in a temporal array, and sent to the lower layer.

The stages for sending data to the lower layer are:

1. Every time a sampling time point is reached in the simulation, `updateBoard` is invoked and it reads the information stored in `pin_in`.
2. The values read from `pin_in` are divided into smaller packets and stored in a temporal array.
3. This temporal array with all the packets for all the inputs is sent to the low layer of ModelPlug.

The stages for receiving data are the same but inverted in order:

1. A temporal array is received from the low layer with all the packets for all the FPGA outputs.
2. The packets are joined together by shifting.
3. The buffer `pin_out` is actualized with the new values for the output pins of the FPGA.
**Low layer**

The last layer, the lowest one, includes the functions that interact directly with the interface. They are called by `updateBoard` every sampling point to send and receive the signals from the FPGA. Their main function is to build the transmission packet. This transmission packet is shown in figure 4.6.

<table>
<thead>
<tr>
<th>Operation (OP)</th>
<th>Size (S)</th>
<th>Data Packet (D)</th>
<th>Data Packet (D)</th>
<th>Data Packet (D)</th>
<th>Data Packet (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 4.6: Composition of the transmission packets.*

The transmission packet consists in a header and the data. The data is composed by the data packets provided by the intermediate layer. The header is constituted by the operation commands and the size of the transmission. The operation commands are used to tell the FPGA which functionality to perform, the current operations are: `writeData`, `readData`, `writeTstep`, and `resetFPGA`.

**writeData** Operation command used to send data to the FPGA. The operation code used is 0x5A for transmission packets of one byte length. If the transmission packets are longer than one byte, the operation code is extended. After sending this operation code are sent the size and the data packets received from `updateBoard`.

**readData** Operation command used to receive data from the FPGA. The operation code in this case is 0x55. The host computer, that is the master in the synchronization mode used, sends the operation command and the size of the transmission to the FPGA. Thereafter, the FPGA answers by sending the data packets.

**writeTstep** This is a configuration command with a operation code of 0x56. It is used at the beginning of the simulation to configure the $T_{step}$, or in other words, to configure how many clock cycles has to run the DUT during each simulation iteration.

**resetFPGA** Operation command used to reset the FPGA from the host computer. The operation code for this command is 0x51.

These operation commands are created with different functions, one for each of them, in the low layer. These functions are called by `updateBoard` and use the pin buffers to store or take the information.
A summary of the functions used in the ModelPlug library is shown in table 4.1. The functions in this table are ordered by communication layers.

**Table 4.1: Summary of the functions used in ModelPlug, ordered by communication layers.**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Top layer</strong></td>
</tr>
<tr>
<td>signalToFPGA</td>
<td>Gets data from the simulator, transform it to unsigned, and write the value to the pin_in buffer.</td>
</tr>
<tr>
<td>signalFromFPGA</td>
<td>Recovers data from the pin_out buffer, transform it to signed and sends it to the simulator.</td>
</tr>
<tr>
<td></td>
<td><strong>Intermediate layer</strong></td>
</tr>
<tr>
<td>updateBoard</td>
<td>At each sample period updates the pin buffers. It first reads the incoming data from the FPGA and stores it in pin_out, then it takes the data from the simulator stored in pin_in and sends it to the low layer. For both, incoming and outgoing, transmissions the data has to be processed as the interface use small data packets.</td>
</tr>
<tr>
<td></td>
<td><strong>Low layer</strong></td>
</tr>
<tr>
<td>FPGA::writeData</td>
<td>Takes the data provided by updateBoard, adds the header to it and sends it to the FPGA.</td>
</tr>
<tr>
<td>FPGA::readData</td>
<td>Sends a command to the FPGA requesting the data results. When the FPGA sends them, this function reads these results and stores them in a temporal buffer for updateBoard to read.</td>
</tr>
<tr>
<td>FPGA::writeTstep</td>
<td>Called before starting the simulation this function sends to the FPGA board the time step configured in the simulation model.</td>
</tr>
<tr>
<td>FPGA::ResetFPGA</td>
<td>This function is also called before the simulation and is used to reset the FPGA.</td>
</tr>
</tbody>
</table>
4.2 Interface

The interface comprises the physical connection between the FPGA and the host computer and the modules that decode the signals. Three main possible connections have been studied for this thesis: The USB JTAG, Ethernet and SPI. The last one, SPI, has been finally chosen for this thesis. This decision has been done because the SPI is a simple interface that could be connected to the GPIO port of the FPGA and could average a similar speed that the USB JTAG. An additional benefit of using this transmission in the host computer side is that they do not require additional drivers, so it is possible to use them in any computer with any operative system. The design on the FPGA board would allow to implement a different interface without many changes, although, some interfaces like Ethernet may need a soft-processor to be able work. The library ModelPlug is designed to use serial communications such as USB, so the implementation of some interfaces like Ethernet would need to modify the library.

4.2.1 SPI

Theory

The chosen interface for the proposed solution is the Serial-Parallel-Interface (SPI). This is an interface that uses four wires and sends the data bits in series. The wires used are a SPI clock wire (SCLK), a chip select wire (SS), and two data wires one for each direction: Master-Output Slave-Input (MOSI), and Master-Input Slave-Output (MISO).

![SPI communication protocol for mode 0.](image)

An example of a SPI communication is shown in figure 4.7. The bits are sent in series, one by one, through MOSI or MISO depending of the direction of the transmission. The bit rate of the transmission is controlled by the SCLK and each bit is sent in one pulse of this clock. The SPI connection is a master-slave connection, in which one master can control several slaves at the same time. The chip select wire controls which of those slaves is being targeted at the moment. The SPI communication has three modes depending on which polarity, phase and edge of the SPI clock are used. For the case implemented in this thesis, the mode used is mode 0 with transmission of the most significant bit first. This is the mode...
represented in figure 4.7. In it the SPI clock has a polarity of zero, which means that the active state is one and the inactive is zero. Moreover, the clock phase for this mode is also zero which means that the sampling of the data is done in the rising edges of the clock and the data has to be outputted on the falling edges of the clock.

Implementation

A normal computer lacks the output interface needed to generate the four wire protocol used by the SPI, therefore to solve this issue an Arduino board is used [32], more specifically a Teensy board [33]. The host computer sends the data signals to transmit to the Teensy board through USB and this board translates it to the SPI protocol. A schematic for this set up is shown in figure 4.8. The physical connection is performed using the dedicated SPI pin interface of the Teensy board together with the GPIO port of the FPGA board.

In the Teensy side, the transmission packets with the array of data samples created by the ModelPlug library are received. Then, the function of this board is to transform this information to the SPI protocol shown in figure 4.7 and transmit it through the four SPI wires. For this purpose the Teensy SPI functions, based on the Arduino SPI functions [34], are used. The voltages used are in the 3.3V range for both the FPGA and the Teensy board. However, in case the FPGA had a output voltage of 5V, the Teensy boards have an input tolerance for that. As commented before, the Arduino SPI libraries allow to send data samples in sizes of 8 or 16 bits. For this reason the transmission packet is divided in smaller packets of that size by ModelPlug.

Inside the FPGA a decoder is used to read the SPI protocol and send the data signals recovered to the rest of the FPGA system. This block is divided in two main sections, one for the receiver and another one for the transmitter. The incoming and outgoing bits are stored in two shift registers during the transmissions. The control of the transmission has been implemented using finite state machines, one FSM for each of them. Both FSMs are designed with a non-predefined number of states. The number of states is defined instead by the parameter N_SPI, which is the number of bits of the data packets. Additionally, the FSMs are controlled by the SPI clock not with the FPGA system clock.
Figure 4.9: Schematic diagram for the receiver FSM. The state is defined inside the circle. The change of state conditions and the outputs are defined over the arrows.

**Receiver FSM**  The receiver state machine is N_SPI states long, from state 0 to state N_SPI - 1. Figure 4.9 shows a diagram for this FSM. The states are used as a counter for the number of bits that arrive. In each of the states a bit is shifted through the less significant bit position of the shift register. The first state, the state 0, is used as an idle state where the FSM waits until the bit transfer starts. This start is signalled by the first SPI SCLK cycle arrival. Then from state 1 to N_SPI-2 the FSM keeps registering the incoming data bits in the shift register. When it is turn for the last state, state N_SPI-1, it means that the last bit is received so the FSM gives an strobe signal to indicate that the reception is finished. Then, this strobe or ready signal alongside with the data stored in the shift register are sent to the rest of the FPGA system.

**Transmitter FSM**  The transmitter state machine needs more functionality than the receiver one. This is because the data sample to transmit needs to be loaded in beforehand. A diagram with this FSM is shown in figure 4.10. The total number of states is N_SPI+1. Similarly than with the receiver, the states are used as a counter for the bits to transmit. The state 0 is, similarly to the receiver FSM, an idle state that waits to the first transmission. This first transmission is controlled from the FPGA system and is set when the command code for transmitting (readData) is received from the host computer. When this code is received the FSM changes to the state 1, which is used to get the data from the FPGA and to wait until the transmission starts. The transmission start is marked when the SPI SS is low and the SPI SCLK starts working. The states from 2 to N_SPI-1 work in a similar way than the receiver, they shift the most significant bit from the shift register to the SPI MISO wire. Then, the last state, N_SPI, shifts the last bit and decides either going to idle or to the state 1 to wait to send more data.
4.2 Interface

![State diagram]

**Figure 4.10:** Schematic diagram for the transmitter FSM. The state is defined inside the circle. The change of state conditions and the outputs are defined over the arrows.

Two points more to clarify about the interaction of the SPI block with the rest of the system are:

1. The only iterations needed between the SPI block and the FPGA system are the data buses (both for transmit and receive), a request data and a data is ready signals, respectively, for the transmitter and the receiver. This attempt to limit the interactions between the interface block and the rest of the FPGA system is done to reduce the amount of work to change the SPI interface block to another one.

2. The data length used for the interface transmission will define the size of the data width for the FPGA system. The purpose of this, is to simplify the control block in the FPGA. Thus, for the SPI interface case the data lengths of one or two bytes used by the interface would force the FPGA system to use the same ones.
4.3 FPGA system

The FPGA HIL system is constituted by four elements: Data handling, control, synchronization, and serial-to-parallel conversion, plus the DUT to simulate. Additionally, in the source code is given a VHDL file that contains the configuration packages and parameters needed to build the system. Figure 4.11 (showed also in the previous chapter) depicts the design inside the FPGA. The following subsections will follow the data flow inside the FPGA HIL system to explain the implementation and configuration of the constituent elements.

![Figure 4.11: Schematic of the FPGA system. White arrows represent data flow and black arrows represent control signals](image)

4.3.1 General parameters

The configuration of the DUT file is unknown when the FPGA HIL system is designed. Therefore, the design of the HIL system needs to be created in a generic manner that can be parameterized. For this purpose there are three main groups of parameters used to build the system. Those parameters are set as constants in a package called `config_pkg` located in the same working library as the rest of the files. This package needs to be invoked in every VHDL file for successful synthesis.
The parameters defined in this package that are used for the system configuration are:

1. The interface word length, its generic name is data_width. This parameter is used to build the data word length of all the system.

2. The number of inputs and the number of outputs, called respectively N_in and N_out. These two numbers are used to generate the input and output registers in the serial-to-parallel block.

3. The data lengths for each input and output. These parameters are configured as an array with the sizes of each individual input or output. The array names are size_array_in and size_array_out. These values are used as well in the serial-to-parallel block to generate the input and output registers.

Each individual element of the FPGA uses these configuration parameters during synthesis to create the system for each individual DUT simulation.

### 4.3.2 Data handling

The data handling block is the first element of the FPGA HIL system seen from the interface side. As commented in section 3.1.3, the two main purposes for this element are to store the incoming and outgoing data from the FPGA and to make the FPGA system more independent from the interface. An additional issue that this block needs to solve is the crossing of clock domains, the interface works with a different clock domain that the rest of the FPGA.

To address these purposes are used asynchronous FIFO memories. The use of asynchronous FIFO memories allows to maintain reliable communication between, two clock domains, while maintaining a faster data rate than the managed with other techniques such as handshakes. The final design consists in two FIFO, each of them with one reading port and one writing port. Those ports works independently, therefore there are needed two address registers for each memory. The control of these FIFOs’ addresses is internal in the data handling block, which simplifies the implementation of a new interface decoder without modifying this block. The implementation of the FIFOs is done using dual port RAMS together with the write and read logic. A diagram for the data handler block design is shown in figure 4.12, in the figure the FIFO on top is the FIFO for the interface receiver and the FIFO under that is for the interface transmitter.

The FIFOs behaviour is controlled mostly inside the data handling block itself. For the writing of both FIFOs are only needed two external signals: A data_ready signal high during one clock cycle and the data signal itself. In a similar manner, for the reading of the FIFOs, two signals are needed: A read_data one clock cycle signal and the data signal. When read_data is asserted the FIFOs will read and set the new value at the data output. An additional signal, named
**Figure 4.12**: Schematic diagram for the FIFOs design inside the data handler block.

FIFO\_empty, is given when the memories are empty. This signal is used by the control block later on to know when to read from the FIFOs.

Generally, for FIFO design there is the possibility of the FIFO getting full and causing data loss. However, the maximum amount of data received by the FIFO before the control block starts reading can only be one transmission packet, so sizing the RAMs accordingly solves this data loss possibility. Additionally, the FPGA reading rate is much faster than the writing rate from the interface (even if the interface has a faster clock than the FPGA this clock is used for receiving each bit, so the rate of writing complete packets to the FIFO is still slower than the FPGA read rate). Therefore there is no need of a complex circuitry to regulate if the FIFOs gets saturated.

Normally, FIFOs designs for clock synchronization uses a synchronization scheme with Gray coding for the FIFOs full and empty signals. However, the reading and writing characteristics of this circuit makes this synchronization unnecessary. On the one hand, the control block only reads when there is information already on the FIFO. Moreover, this reading rate is faster than the writing rate. On the other hand, the writing from the FPGA HIL side is performed without interruption when the simulation is finished, so when the interface reads the FIFO all the information is already there.
### 4.3.3 Control

The FPGA HIL system needs to be able to read the operation commands sent by the host computer and respond to them. The control block is in charge of these functions; it reads the commands that arrive to the input FIFO and set the control signals accordingly. This block is designed as a finite state machine (FSM), this design is chosen because of its simplicity, effectiveness and low area use.

![State graph sketch of the control state machine.](image)

The FSM is designed in a branched style using one branch for each of the operation command. In this way it is easier to implement new functionality if needed by just adding a new branch to the FSM. The behaviour is as follows: When a communication is received at the input FIFO, the FSM reads the operation command and choose the corresponding branch. In figure 4.13 a state graph sketch with the main different states and branches is shown. In the image, from left to right are shown the branches for: receiving data, sending data, force reset, and set $T_{step}$.

The change of states in the way down of the FSM tree shown in figure 4.13 is performed when the Rx FIFO in the data handling section receives information from the interface. This data reception is marked with the de-assertion of the signal $FIFO\_empty$. The read_OP state reads the operation command received and if the value is one control command sent by the simulator, the FSM continues to the corresponding branch.
Receiving data branch  The first data branch in figure 4.13 is the receive data branch. This branch, as it name states, is used to receive data from the host computer. The receiving data branch has two iterative states: One to wait until new data arrives from the interface and another one to read that data and send it to the serial-to-parallel block. This first state is \texttt{wait\_data}; when the communication is started (defined by the receive data operation command) the FSM enters in this waiting state until a data sample arrives to the FIFO register. The second state is \texttt{read\_data}, the FSM changes to this state after the arrival of a new sample. This state gives a control signal, \texttt{enable\_shifting}, to the serial-parallel block ordering it to take the sample from the data bus. This signal enables the shifting of data in the registers of the serial-parallel block (more on this will be explained in section 4.3.5). Additionally, a counter keeps track of the number of samples that have arrived. When the total number of the transmission packets is received, the FSM restarts to the state \texttt{idle}.

Sending data branch  The second data branch in figure 4.13 is the send data branch, branch designed also with two iterative states. The first state is called \texttt{prepare\_data}. It purpose during the first iteration is to give the start transmission signal to the data handling block. After the first iteration, this state function is to keep track of how many data samples have been sent using a counter. The second state, \texttt{send\_data}, gives a signal to the FIFOs to send data to the interface decoder. Thereafter, when one data sample is transmitted, the FSM moves back to the \texttt{prepare\_data} state to count again or to \texttt{idle} if the final count is reached.

Force reset branch  Next branch is a force reset branch. This branch restarts the FPGA HIL system and it is used at the beginning of each simulation to reset the FPGA system by software from the host computer. This is done to make sure that there is no old data from previous simulations.

Set time step branch  The last branch in the state graph sketch is the time step data branch. This branch is used to set the time step for the DUT. The FSM reads the new time step value sent by SystemModeler and stores it in a control register. This register with the new time step value will be used by the synchronization block to calculate how many clock cycles the FPGA has to run each simulation step.
4.3.4 Synchronization

The synchronization block main purpose is to control the run of the DUT during each simulation step. Additionally, this block is used for loading data to the DUT and acquiring back data from it.

The synchronization block counts with two main inputs: A signal that marks the start of the simulation, \texttt{start\_sim}, and a register that contains the value for the \texttt{T\_step}. With these inputs the synchronization block needs to be able to start the simulation; stop it when the \texttt{T\_step} value has been reached; and give the commands to recover the result of the DUT.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{synchronization_state_machine}
\caption{State graph sketch of the synchronization state machine.}
\end{figure}

The synchronization is designed using another FSM. The decision for this design is to allow and simplify further modifications of the synchronization modes that are used. A state graph sketch for this FSM is shown in figure 4.14. The state machine consists of four states:

- **Idle** The first state is an idle state where the FSM stays until the control block gives the start simulation signal. At that moment, the FSM moves to the load state.

- **Load** This state is an one cycle intermediate state to load the registers at the input of the DUT file (more on this in the next subsection) and immediately changes to the next state.
Simulation This state controls the run of the HW file with an enable signal, `enable_sim`. The end of the simulation step is determined by the $T_{step}$, when the $T_{step}$ is reached the run of the DUT is stopped.

Acquiring When the $T_{step}$ stipulated for the FPGA is finished, this acquisition state gives the signal to the serial-parallel block to store the data outputs of the DUT. This enable acquisition signal is maintained high until all the data outputs from the DUT have been shifted out. The duration of this state is also controlled by a counter together with the number of output registers of the serial-parallel block.

The activation of the DUT during the simulation steps can be controlled in two ways: First, an enable signal `enable_sim` available can be used if the DUT is designed with an enable signal; this enable signal can be used to activate or deactivate the elements of the DUT design. Second, a clock signal that can be enabled. This clock is activated by the `enable_sim` for the number of $T_{steps}$ stipulated and then deactivated again.

### 4.3.5 Serial-parallel converter

The serial-parallel block is used to perform the transformation from the serial transmission of the interface to the parallel nature of most digital designs. The serial-parallel block consist in a serial-to-parallel converter for the received data and a parallel-to-serial converter for the data to be sent. The first one is used at the input of the DUT and the second one is used at the output.

The design for this block is based on shift registers. This implementation has been chosen for being the most simple to control and generalize. The incoming data samples are shifted into position with respect to the data inputs of the DUT. Opposite to this, for the output, at the end of the simulation step the outputs of the DUT are stored in shift registers and shifted out in series.

The main issue that appears when designing this block is that the number of inputs and outputs of the DUT file is not known. Therefore the design needs to be created in a generic manner. For this purpose, one level of `if ... generate` and `for ... generate` statements are used to replicate the set up of one register and apply it to every input or output. The number of replications depends on the number of inputs or outputs and is defined by a configuration parameter.

However, another issue arise with the use of different data sizes. The FPGA HIL system has a word length of 8 or 16 bits that are defined by the interface, but the sizes of the inputs and outputs can vary for the digital design implemented. This is solved using a second level of `if ... generate` and `for ... generate` statements. The shift registers do not contain the value for one input or output, but the values for the small packets of 8 or 16 bits. The final number of shift registers used for each of the inputs is given by the sample width/system width relation.
Figure 4.15 shows an example with two inputs, one of 32 bits long and the other one with 64 bits long. The data packets enter through the top register of the picture and are stored in the register when the signal `enable_shift` is active. On the other case, when this control signal is not active, the registers maintain the value they held. Thus, the control block, which is in charge of the generation of that control signal, needs to give as many pulses as total number of packets compose all the data inputs. These pulses are, as well, given synchronized with the arrival of the data packets. When all the data packets have arrived and are in place, the values of those registers are concatenated to form the inputs to the core of the DUT. It can be observed from the figure that the packets of the data have to be received from less significant to most significant so the inputs to the DUT are ordered properly. Additionally, due to the shifting design the data sample that arrives in the first place will be set to the last input of the DUT.

For the output the process is performed in the opposite way. The outputs of the DUT are divided into packets with the system data width and stored in the output registers. Then, the data packets are shifted through the registers and outputted one by one to the data handling unit that will store them. This acquisition of the data and the posterior shifting is controlled by the synchronization block.
Additionally, in the case when there is not $T_{step}$ defined and the DUT design is running continuously in the FPGA, the incoming data that is being received will not be steady during the shifting process. For this case an additional register is placed in each of the input ports of the DUT to keep the input values steady until the shifting is finished. The load of this register is controlled by the load state of the synchronization FSM explained in 4.3.4.

4.3.6 DUT

The last block of the FPGA HIL system is the DUT. However, as this block will be implemented by the user the comments about the DUT file in this subsection are regarded as how the implementation of a new file can be performed. Additionally, some examples of VHDL files that have been used during the testing phase to obtain results for this thesis will be explained in the results chapter, chapter 5.

Algorithm 4.1: Instantiation of the DUT file inside the system main file

```vhdl
u5: component HW_core
    port map(
        reset_n => core_reset_n,
        sys_clk => sim_clk,
        core_in_1 => core_in(0),
        core_in_2 => core_in(1),
        core_in_3 => core_in(2),
        core_in_4 => core_in(3),
        core_out_1 => core_out(0));
```

The first step to link a file is to modify the `config_pkg` package, mentioned in section 4.3.1, with the values for the new file. This configuration parameters from this package will build the system according to the specifications for the new digital design. Apart from that, the only other step needed is to link the DUT in the main file `HIL_main.vhd`. An example is shown in algorithm 4.1. In this code a design with four inputs and one output is linked. The generic names for the inputs and outputs inside the file are `core_in_x` and `core_out_x`, being $x$ the number of input or output. Those names could be changed without needing to perform big modifications. However, the signals that connect these ports with the rest of the design should not be modified, otherwise, the serial-parallel and main VHDL files need to be modified as well. These signals are named `core_in` and `core_out` and are used respectively for the inputs and outputs of the DUT. These signals are created as an array of `std_logic_vector` where the first index of the array is used to access the corresponding input or output.
An additional comment is that, in the design, the generic name used for the HW file to simulate is \texttt{HW\_core} and the file is \texttt{HW\_core.vhd}. These names could be changed but in that case the instantiation of the components have to be also modified. Moreover, the file implemented should be placed in the same work library that the rest of the design.
Results and discussion

In this chapter are explained the testing results performed on the HIL platform. To obtain these results two designs were implemented and tested as DUTs on the HIL simulator.

5.1 PID

The first complex design implemented is a Proportional-Integral-Derivative (PID) controller used to regulate a DC electrical motor. The PID has been chosen because is a commonly used controller design and it does not have a big complexity. To perform the testing, first a model of the PID has been created in SystemModeler. This model is used to perform the comparisons with the hardware PID controller implemented in the FPGA board to test its functional behaviour.

5.1.1 PID theory

The PID is a very popular controller in digital systems due to its simplicity and its easiness to implement [35, 36]. The PID is based on the multiplication by a constant, the integration, and the derivation of an input error signal. This error signal is obtained by the subtraction of the reference signal with the output measurement from the controlled plant. The conventional equation is given by:
\[ u(t) = K_p \left\{ e(t) + \frac{1}{T_i} \int_0^t e(\tau) d\tau + T_d \frac{de(t)}{dt} \right\} \quad (5.1) \]

In this equation the coefficients \( K_p \), \( T_i \), \( T_d \), and \( e(t) \) represent, respectively, the proportional gain, the integral time constant, the derivative time constant, and the error signal. This equation can be modified to be implemented in a digital design using a sample time, and its computation complexity can be reduced with a recursive algorithm [35]. The equation for this implementation is:

\[ u(k) = u(k - 1) + q_0 e(k) + q_1 e(k - 1) + q_2 e(k - 2) \quad (5.2) \]

The coefficients for this equation are given by the following terms:

\[ q_0 = K_p \left( 1 + \frac{T_d}{T} \right) \]
\[ q_1 = -K_p \left( 1 - \frac{T}{T_i} + 2 \frac{T_d}{T} \right) \quad (5.3) \]
\[ q_2 = K_p \frac{T_d}{T} \]

A digital implementation [36] for this equation is shown in figure 5.1. First, the input error signal needs to be delayed two clock cycles. Then, the error, the error delayed one cycle and the error delayed two cycles are multiplied by the coefficients of equations 5.3. The final step is to add the results of those multiplications with one delayed sample of the output.

![Diagram of the digital PID architecture.](image-url)
5.1 PID

5.1.2 SystemModeler implementation

A modelled digital implementation for the PID has been created in SystemModeler. This model is used to compare with the real digital implementation in the hardware. An image of this model in SystemModeler is shown in figure 5.2. The purpose of this model is to control the turning on and off of a DC motor using the PID. The PID coefficients are implemented as constants which are calculated with the proportional, integrative, and derivative terms that are defined as global parameters for the model. To match the result of this model with the hardware version implemented in the FPGA, an extra delay is implemented at the input of the PID (refer to section 2.2.2 for the one delay of the FPGA simulation results).

![Figure 5.2: Representation of the PID digital model in SystemModeler.](image)

The hardware implementation of the digital PID controller is made following the design shown in figure 5.1. The SystemModeler simulation model that uses the FPGA instead of a virtual PID is shown in figure 5.3. In this design the \( q_0 \), \( q_1 \), and \( q_2 \) coefficients are given as inputs to make easier the simulations of the PID, in this manner, the digital design in the FPGA does not need to be re-synthesized every time those coefficients are modified. The data types used for this simulation are signed fixed point data types of 32 bits with 16 bits fractional.
5.1.3 Results

Functional testing

The SystemModeler models explained in the previous section are used together to perform functionality testing. In figure 5.4 the plots of the outputs from both models are shown in the same graph. In that graph it can be observed that the functionality of the real controller implemented in the FPGA is the same as that obtained from the simulated model in SystemModeler. The results obtained from both models, mathematical and real, are practically coincident. The small variation is due to the changes from floating point data in the model and fixed point in the FPGA. It is important to emphasize this difference, even if is very small. The simulation approach using the FPGA uses the real data formats of the hardware controller, in contrast with the data types of the modelled version. This means that the results of the HIL simulation are more realistic. The results are obtained with values of 6, 2, and 0.5 for, respectively, $K_p$, $T_i$, and $T_d$. A sampling time $T$ of 0.001 seconds is used for this case.
Figure 5.4: Functionality testing comparing the results from the modelled PID and the digital PID in the FPGA.

Additionally, two graphs with variations of the PID’s configuration coefficients are shown in figure 5.5 for the SystemModeler model and in 5.6 for the FPGA HIL model. These figures are used to demonstrate the correct functionality of the digital PID implemented in the FPGA and how the HIL platform works properly to perform the tuning of the PID.

Figure 5.5: Variation of configuration parameters in the modelled PID implementation.
Another interesting test to measure the performance of the HIL simulator is to modify the sampling times to see how they affect the simulation. The size of the sampling intervals affects the total computation time of the simulation. Shorter time intervals mean more frequent data interconnections, which are the bottleneck of the design, between the computer and the HIL system. However, increasing too much the sampling interval could affect the performance of the digital controller simulation. In figure 5.7 are shown different plots with different sampling intervals for the same coefficients of the PID. The sampling times from 0.01 to 0.001 seconds show a correct behaviour for this PID; the sampling times bigger than those can cause the HIL simulation to become unrealistic. The PID needs to be reconfigured for big sampling times, however, those sampling times would need to correspond with the ones in the real system that is being modelled.
Performance tests

The next step is to verify the speed performance of the HIL system with this set-up. The timing measurements are divided in two: The measurements in the FPGA board and the measurements in the host computer side. On the one hand, the measurements in the FPGA board are taken using SignalTap II Logic Analyser software from Altera. These measurements contain the times of the SPI transmission seen from the FPGA GPIO port and the internal times for the FPGA system. On the other hand, the measurements in the host computer regard the total times of the functions: updateBoard, FPGA::readData, and FPGA::writeData. The time measurement in updateBoard includes the total time spent for the low and medium layers communication. The times in FPGA::readData and FPGA::writeData only contains the read from FPGA and write to FPGA times respectively. However, measuring the time for FPGA::writeData is problematic. The function in the host computer side finishes when the data is sent to the serial interface of the computer, therefore it does not include the time spent in the Arduino transforming to the SPI communication and in the SPI communication itself. Hence, in the transmission to the FPGA only are measured the time spent in the computer and the time spent in the SPI transmission, but the intermediate time spent in the Arduino is unknown. This is not a problem for FPGA::readData because it waits until the Arduino responds sending back the information recovered from the FPGA, so it does include all the receive transmission communication time.

There are three configuration values that could affect the total transmission times: The sampling time, the sample data width, and the interface data width. Different simulations varying these values are performed to observe if these parameters really affect the transmission time and how they do it.

Figure 5.7: Effect of the variation of the $T$
The first parameter studied is the sampling time. Theoretically, this parameter should not affect the time spent on each communication but it should affect to the total computation time of the simulation. This is the parameter that has to be adequately configured if real-time simulations are to be run. Several simulations are done varying this sampling time value and its results are shown in Table 5.1. For these simulations the configuration for the other parameters are: 32 bits for the sample data length and 8 bits for the interface data length. The total simulation time is set to 10s for all the performed tests.

<table>
<thead>
<tr>
<th>Sample Time (s)</th>
<th>Individual Communication</th>
<th>Total Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>readData (µs)</td>
<td>writeData (µs)</td>
</tr>
<tr>
<td>0.001</td>
<td>157.8</td>
<td>17.64</td>
</tr>
<tr>
<td>0.002</td>
<td>154.4</td>
<td>17.93</td>
</tr>
<tr>
<td>0.0025</td>
<td>152.5</td>
<td>17.54</td>
</tr>
<tr>
<td>0.005</td>
<td>155.4</td>
<td>17.84</td>
</tr>
<tr>
<td>0.01</td>
<td>161.4</td>
<td>17.91</td>
</tr>
<tr>
<td>0.05</td>
<td>149.4</td>
<td>17.39</td>
</tr>
<tr>
<td>0.1</td>
<td>169.2</td>
<td>17.48</td>
</tr>
</tbody>
</table>

The values of updateBoard, FPGA::readData, and FPGA::writeData in Table 5.1 shows that the assumption of the sampling time not affecting each individual communication is true. The simulations shows an average of 0.18 ms, 0.157 ms, and 0.018 ms for respectively: updateBoard, FPGA::readData, and FPGA::writeData. This is a normal result if is taken into account that a modification of the sampling time only affects to the number of iterations between hardware and software that are performed, but it does not affect to the individual iterations.

The most interesting values from Table 5.1 are the computation times for the simulation and the number of events. The results for these simulations shows a proportional increment in the total computation time of the simulation with respect to the number of events (the number of events are defined by the sampling time, each sampling period results in one event. For example, the 10 seconds simulation divided in sampling intervals of 0.001 seconds results in 10000 events). The division of the total computation time of the simulation between the number of events results in an average value about 15 ms for each event, which is a much bigger value than the time spent by updateBoard in the communication. This relation between the total computation time and the number of events can be observed better in figure 5.8, were it can be observed that if follows a totally linear progression.
The second configuration parameter to take into account is the data length of the samples. Four simulations are run for sample sizes of 64, 32, 16, and 8 bits, meanwhile the sampling time and the interface data length are maintained, respectively, in 1 ms and 8 bits. The results for these simulations are shown in table 5.2. In this table the values for the timings of each communication in the host computer and in the FPGA are represented. The computation times for the simulation are nearly the same for all simulations so they are skipped in this table.

The table 5.2 shows that the sample sizes indeed affect the total timings. Meanwhile the significant time increment in the SPI communication is expected (the number of bits to send are doubled with each step); the processing time in the host computer should not be affected in a considerable manner.
In figure 5.9 a bar chart with the different values of table 5.2 is shown. From this chart it can be seen that the time increment of the SPI communications is proportional to the number of packets sent. Moreover, looking to the `FPGA::writeData`, that as explained before only measures processing time in the host computer, it can be observed that the processing time does not get incremented significantly. The increments in `updateBoard` can be explained with the SPI increments in both the sending and receiving data to the FPGA board. Additionally, the bar chart shows that the main bottleneck is the receive communication from the FPGA (performed by the function `FPGA::readData`). It is normal that this function takes more time than `FPGA::writeData`, as it is sending a command to the FPGA and waiting for the response. However, it is still more than twice the time spent by the send communication to the FPGA.

The last configurable parameter to test is the interface data length. The two data lengths used by the Arduino SPI libraries are 8 and 16 bits. By using the 16 bits packets instead of the 8 bits packets in the interface, the total number of packets that needs to be processed and sent by the computer reduces in half. This reduction should be noticed in the total times for each individual communication. Additionally, the SPI communication generated by the Arduino board is not continuous, between packets there is an idle time in the communication. Therefore, the increment of the packet size would reduce the number of packets sent and thus the amount of idle times. Table 5.3 contains the results for the same simulation than table 5.2 but using an interface of 16 bits instead of 8 bits.
Table 5.3: Values timings of the communication function and the SPI communication with variations of the data sample size for 16 bits interface data width.

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>Host Computer</th>
<th>SPI to FPGA</th>
<th>SPI from FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>readData (µs)</td>
<td>writeData (µs)</td>
<td>updateBoard (µs)</td>
</tr>
<tr>
<td>64 bits</td>
<td>208.4</td>
<td>17.41</td>
<td>230.6</td>
</tr>
<tr>
<td>32 bits</td>
<td>192.6</td>
<td>17.68</td>
<td>215.7</td>
</tr>
<tr>
<td>16 bits</td>
<td>168.9</td>
<td>15.9</td>
<td>189.8</td>
</tr>
</tbody>
</table>

The results from this table shows:

- First, that the interface data length does affect the timings in the host computer, however, the effect is not always a reduction as expected. For the `writeData` function, it does cause a small time reduction for bigger data sizes. However, for `readData` the time consumed is increased by using 16 bits packets in the interface. A possible explanation for this is that the increase of time to transform to the SPI protocol from 8 bit packets to 16 bits packets done by the Arduino functions is not proportional. Then, the reduction of execution time gained by reducing the number of packets does not compensate the increase of time for these SPI functions.

- Second, the use of bigger packets in the interface does reduce the total SPI transmission time. However, this is only useful for transmissions with many packets. Then, this interface has a clear reduction of time for data sizes of 64 and 32 bits in the transmission to the FPGA, which are transmissions of, respectively, 18 and 10 packages for the 16 bits interface and of 34 and 18 packages for the 8 bits interface.
Area results

The area resources used by the HIL system, without the PID controller, are shown in tables 5.4 and 5.5. In table 5.4 are shown the values obtained with the Altera Quartus program, the board used is a Terasic DE1 development board with a Cyclone II 2C20 FPGA. In table 5.5 are shown the values obtained with the Xilinx ISE program, the board used is a Papilio Duo with a Spartan 6 LX9 FPGA. The clock speed goal is 50Mhz and 32Mhz for the Altera and Xilinx boards respectively. Moreover, the operating settings of the synthesis tools were set to typical without any special settings for the synthesis tools.

Table 5.4: Area resources used in the Altera Cyclone II 2C20 FPGA.

<table>
<thead>
<tr>
<th>Resources</th>
<th>Used</th>
<th>Total</th>
<th>% used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Elements</td>
<td>472</td>
<td>18752</td>
<td>3</td>
</tr>
<tr>
<td>Registers</td>
<td>331</td>
<td>19649</td>
<td>2</td>
</tr>
<tr>
<td>Pins</td>
<td>67</td>
<td>315</td>
<td>21</td>
</tr>
<tr>
<td>M4Ks RAMs</td>
<td>2</td>
<td>52</td>
<td>4</td>
</tr>
<tr>
<td>Memory Bits</td>
<td>9216</td>
<td>239616</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.5: Area resources used in the Xilinx Spartan 6 LX9 FPGA.

<table>
<thead>
<tr>
<th>Resources</th>
<th>Used</th>
<th>Total</th>
<th>% used</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>291</td>
<td>5720</td>
<td>5</td>
</tr>
<tr>
<td>Slice Registers</td>
<td>338</td>
<td>11440</td>
<td>2</td>
</tr>
<tr>
<td>Pins</td>
<td>14</td>
<td>102</td>
<td>13</td>
</tr>
<tr>
<td>B8BWER RAMs</td>
<td>2</td>
<td>64</td>
<td>3</td>
</tr>
</tbody>
</table>

Tables 5.4 and 5.5 shows an area utilization of about 2 to 5 % for the main resources. These are good area results if it is taken into account that both FPGAs used for the thesis are low end FPGAs with a reduced amount of resources.
5.2 Filter

The second digital design used to perform verification tests of the HIL system is a digital filter. This decision has been made because filters are common designs to implement in a digital manner and they are fairly simple to design. Moreover, a digital filter could need smaller sample times if high frequencies have to be used, which is an interesting point to test.

5.2.1 Biquad filter theory

The filter design implemented is a digital biquad filter for audio. The biquad is an IIR filter of second order. The name biquad comes from bi-quadratic, the reason for this name is that the transfer function for this filter in the Z-domain is composed by two quadratic functions. This transfer function in Z-domain for the biquad filter is:

\[
H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}} \tag{5.4}
\]

However, it is common to normalize \(a_0\) to 1 for some architectures. Then, the transfer function would be:

\[
H(z) = \frac{b_0 + b_1 \frac{a_1}{a_0} z^{-1} + b_2 \frac{a_2}{a_0} z^{-2}}{1 + \frac{a_1}{a_0} z^{-1} + \frac{a_2}{a_0} z^{-2}} \tag{5.5}
\]

This normalised transfer function implemented in a difference equation is:

\[
y[n] = \left(\frac{b_0}{a_0}\right) x[n] + \left(\frac{b_1}{a_0}\right) x[n-1] + \left(\frac{b_2}{a_0}\right) x[n-2] - \left(\frac{a_1}{a_0}\right) y[n-1] - \left(\frac{a_2}{a_0}\right) y[n-2] \tag{5.6}
\]

The direct implementation of the previous equation is called direct form 1. The digital design for this implementation is shown in figure 5.10. This design is composed by four delays, five multipliers and four two-input adders.
A second form for the filter equation is obtained by rearranging the design elements. This equivalent design for the biquad filter is called direct form 2, Image 5.11 represents this design. Using the direct form 2, the number of delay elements gets reduced to two.
The difference equation for this form of the biquad filter is:

\[
v[n] = x[n] - \left( \frac{a_1}{a_0} \right) \times v[n - 1] - \left( \frac{a_2}{a_0} \right) \times v[n - 2]
\]

\[
y[n] = \left( \frac{b_0}{a_0} \right) \times v[n] + \left( \frac{b_1}{a_0} \right) \times v[n - 1] + \left( \frac{b_2}{a_0} \right) \times v[n - 2]
\]  (5.7)

The coefficients for the multiplications can be calculated with the following equations:

\[
b_0 = \frac{1 - \cos (W_0)}{2}
\]

\[
b_1 = 1 - \cos (W_0)
\]

\[
b_1 = \frac{1 - \cos (W_0)}{2}
\]

\[
a_0 = 1 + \alpha
\]

\[
a_1 = -2 \times \cos (W_0)
\]

\[
a_2 = 1 - \alpha
\]  (5.8)

To calculate these coefficients the values for \( \alpha \) and \( W_0 \) are needed. These values can be obtained with:

\[
W_0 = 2 \times \pi \times \frac{f_0}{f_s}
\]

\[
\alpha = \frac{\sin (W_0)}{2 \times Q}
\]  (5.9)

Where: \( f_0 \) is the cut off frequency, \( f_s \) is the sampling frequency, and \( Q \) is the Q-factor. The Q-factor is the quality factor which gives how damped is the filter, it gives the ratio between dissipated energy with the stored energy.
5.2.2 SystemModeler implementation

For the design implemented in this thesis the direct form 1 is used. The simulation method is the same that was used for the PID controller. First, a mathematical model for the filter is created in SystemModeler and simulated. Afterwards, the digital filter is implemented in the FPGA board using VHDL and simulated with the same set up than the modelled filter. Then, the results of both filters are compared.

To perform the simulation the input is set to a square pulse of frequency 100 Hz. This pulse due to its square form will cause harmonics at higher frequencies. Then, the purpose of the filter in this simulation is to eliminate these harmonics. The filter coefficients are defined so the biquad functions as a low pass filter with a cut off frequency \( f_c \) of 200 Hz.

![SystemModeler implementation of the filter.](image)

The biquad model implemented in SystemModeler is shown in figure 5.12. The values for alpha and \( W_0 \) are generated with general parameters in the simulation model. In figure 5.13 is shown the SystemModeler model representation with the FPGA board. The values for the multiplication constants are sent as inputs to the FPGA instead of being parameterized inside the FPGA. In this case, the parameters for alpha and \( W_0 \) are also defined as general parameters in the simulation model.
Figure 5.13: FPGA implementation of the filter.

5.2.3 Results

The frequency plot for the square input pulse is shown in figure 5.14, in this figure it can be observed the main tone at the frequency of 100 Hz and then several harmonics at higher frequencies. The results for the simulation are shown in figures 5.15 and 5.16. The first figure, figure 5.15, shows the frequency plot of the modelled filter response. This filter has attenuated the harmonics, but they are not totally removed. On the other figure, figure 5.16, the frequency plot of the digital filter in the FPGA is shown. The response of the filter is not the exact same that the modelled one, the results are better than in the modelled version. The harmonic tones are more attenuated in the FPGA design. However, such a substantial difference between the modelled filter and the filter simulated in the HIL is unexpected. This difference could be explained by the fractional data differences between the floating point types from the simulated model and the fixed points from the FPGA model. The rest of the values for both models have been verified and are the same for both filters.
Results and discussion

Figure 5.14: Frequency and phase plots for the square pulse. The main tone is located at 100 Hz and the harmonics at higher frequencies.

Figure 5.15: Frequency and phase plots for the modelled filter response. The harmonics are highly attenuated.
The actual responses in time domain for both filters are shown in figure 5.17, for the modelled filter, and in figure 5.18, for the real filter in the FPGA. To eliminate the harmonics, the filters try to transform the square signal to the sinusoidal equivalent. Comparing image 5.17 with image 5.18 it can be observed that the output generated by the filter in the FPGA is more smooth than the output of the modelled filter. However the filter in the FPGA adds more phase shift to the signal, this extra phase shift could be caused by the delay of the synchronization model.
Results and discussion

Figure 5.17: Plot of the pulse signal and the model filter output in time domain.

Figure 5.18: Plot of the pulse signal and the FPGA filter output in time domain.
The conclusions of the thesis and the future work are presented in this chapter.

### 6.1 Conclusions

The first and main conclusion is that the solution proposed in this thesis is a feasible co-simulator platform for digital designs implemented in FPGAs and physical models created with Modelica. The results presented in the previous chapter demonstrates that the proposed system between SystemModeler and the FPGA allows to perform co-simulations with good reliability of the results. Mostly because the simulation results obtained from the HIL simulation are more realistic since the real hardware component is tested. In addition, the final resource utilization of the FPGA part of the system is very low, it ranges from a 3 to 5% of the resources of both FPGA boards used, which is a good result taking into account that the FPGAs used are not big in size compared with others in the market. The need to have a low area usage system is one of the main requirements for this thesis. Moreover, another interesting feature of the system is that the combination of the simulation environment in SystemModeler and the FPGA system architecture allow to implement several FPGAs for one HIL simulation.

The thesis, in addition to propose a feasible system to perform HIL co-simulation, intended to answer a few questions about the system.

The first question to answer is if the system is flexible enough to perform different hardware-in-the-loop simulations without many important changes. The answer

---

1. 472/18752 Logic elements for the Altera Cyclone II and 291/5720 LUTs for the Xilinx Spartan 6
is that the system is quite simple and flexible to modify. In the host computer side, the simulation model in SystemModeler is easily configurable with only the modification of the configuration parameters in three classes created during this thesis. This modification of the configuration parameters can be performed with a graphical interface which simplifies the process. On the other side, in the FPGA, all the configuration parameters are implemented in the configuration package. Therefore, the only needed modifications in this side are to change these parameters and afterwards link the HW file of the DUT in the main VHDL file of the system.

A second question is if the co-simulation model gives more advantages other than testing a digital hardware design. This question could be answered with the explanations proposed in chapter 2 of why using a HIL simulation, i.e.: Reducing the developing times and costs, and the testing of safe-critical systems without putting in danger the complete physical system. However, one of the most important purposes, in addition to perform functional testing of the DUT, is that using the system to perform HIL simulations allows to create more realistic simulations. The response obtained from a real controller is always more accurate and realistic for the real system than a modeled version of it, which enhances the quality of the verification process. Regarding more specifically the system implemented in this thesis, it can be useful to perform fine tuning of the control coefficients of a system. For example, in the PID controller presented in chapter 5, the system proved useful to tune the PID coefficients in the FPGA. The coefficients were modified in the simulation model for each simulation, this option allows to reduce much more the developing time than modifying these values in the FPGA, which means that the file in the FPGA has to be re-synthesized every time.

The lasts questions for the co-simulation model are regarded to the synchronization modes and the transfer speed rates. The synchronization mode presented in this thesis has been considered the most useful for this system. It allows to co-simulate with a limited time step in the FPGA, but it additionally allows to perform a parallel running of the computer and the FPGA if this time step is not set. The main idea for the timings in HIL simulations is to be real-time and this synchronization model allows that. The speed for the transfer rates is not as critical as it needs to be for co-processing. The interface presented in this system would allow to perform real-time simulations for real controllers that does not need a really fast sampling rate, a sampling rate of 1 to 10 ms as described in [3] would be fine. However, the problem of each event in the simulator lasting around 15 ms without mattering the interface transfer time will impede the real-time simulations for the range described in [3].

The rest of the questions for this system are regarding the possibility of using it as a co-processor. The main problem of using this system to perform these co-processing is the transmission rates of the interface. The SPI transmission implemented is fast enough to perform co-simulation in real-time for some controller applications. Nevertheless a co-processing use would need to perform as
fast as possible, which would even need to be faster than real-time. Therefore, the
timing results presented in chapter 5 are insufficient for this task. To allow these
transfer rates a fast communication interface, like an FPGA with PCI, would be
needed. However, this would require of a specific board and a specific host com-
puter, so the flexibility factor is lost. The USB-SPI connection presented allows
to connect any computer with almost any FPGA board.

6.2 Future work

The focus for the future work should aim to improve performance aspects of the
system. The foremost aspects to improve are regarded to the transfer speed of
the system, as one of the main points for hardware-in-the-loop simulations is to
be performed in real-time. This is not very feasible with the actual configuration,
so some improvements are needed in that sense. However, the simulation model
will always be a top limiter for the speed, for very complex simulations it may
not be possible to achieve real-time for small sample times even if a very fast and
optimised interface is used.

The first improvement in the system should be regarding the interface. The whole
communication process should be improved. The SPI communication speed is
not very high, however in chapter 5 is shown that the majority of the time taken
by the communication functions is not this SPI transference but the whole pro-
cess around it. The function FPGA::readData, for example, is specially ineffi-
cient, it takes most of the communication time. Therefore not only would be
interesting to implement a new interface with higher transmission rates such as,
for example, Ethernet, but also to improve the communication functions in the
host computer side. Moreover, with the SPI communication the process of com-
municating from the computer with the Arduino and then with the SPI transfer
to the FPGA could be less efficient than a straight interface from the host com-
puter to the FPGA. For this reason, to implement a more simple interface, even
if that interface has not a faster transfer rate than the SPI, that does not use an
intermediate element could be useful to reduce these extra inefficient times. This
other interface would also be useful to eliminate the need of having to use an
external element such as the Arduino. Another needed improvement to be able
to perform real-time simulations is to determine from where the more or less
fixed 15 ms that the simulator spend in every simulation even. This time has to
be reduced if real-time simulations are to be performed.

Once the speed performance aspects are improved, the future work could fo-
cus towards implementing more features for the system. More synchronization
modes should be added to the system, mainly some synchronization mode where
the host computer and the FPGA does not have a master-slave relationship. Addi-
tionally, the co-processor use for an FPGA board is still interesting. If an interface
with a faster transfer rate is implemented the co-processor set up would be an in-
teresting feature for the system.


