USER INTERFACE DEVELOPMENT IN EMBEDDED DUAL-CORE SYSTEM FOR POWER CONVERTER CONTROL APPLICATIONS

Santiago Antón Area

Master’s Thesis presented to the Telecommunications Engineering School
Master’s Degree in Telecommunications Engineering

Supervisors:
Francisco Poza González
Jesús Doval-Gandoy

2018
Acknowledgements

First of all, I would like to thank my family and my couple for the support received during this academic stage, both emotionally and economically.

Second, to those teachers who have tried to transfer me their knowledge and those who have tried to criticize my mistakes, since both are part of the evolution at this stage.

In addition, I would like to thank my supervisors, Francisco Poza González and Jesús Doval-Gandoy, for understanding the difficulties of completing this Master Thesis while advancing my professional career.

Finally, I would like to thank the University of Vigo and the Electronics Technology Department for giving up the necessary hardware for the accomplishment of this Master Thesis.

Santiago Antón Area

Vigo, July 16, 2018
Preface

This project was born from the Electronics Technology Department of the Higher Technical School of Industrial Engineering of the University of Vigo and from the need to establish its own platform for the study of applications related to the power control of photovoltaic inverters.

As a result of this, the possibility of implementing a solution based on the development of a platform focused on the interface and management of a control application is proposed. Said platform must be able to implement any type of power control of those studied in said department.

In addition, the Final Degree Project in Bachelor’s Degree in Telecommunication Technologies Engineering at UVigo was based on a signal acquisition card in power control applications. Carrying out a project compatible with a previous project motivates to get it implemented.

Santiago Antón Area

Vigo, July 16, 2018
Abstract

Throughout this Master Thesis, a debugging interface based on an Asymmetric Multi-Processing (AMP) configuration on a System-On-Chip (SoC) from Xilinx is designed and implemented for tasks related to the control of converters for photovoltaic panels.

By way of introduction, the application that has this Thesis objective is defined and the different elements that serve as a basis are specified, such as the development board to be used and different technologies such as Grid-Tied Converters and Real Time Operating Systems.

Within the bulk of the thesis, first, an analysis is made of the different AMP configurations in which this application can be implemented, selecting the most appropriate one based on the requisites.

Secondly, the internal architecture of the Zynq SoC is detailed to understand the operation and communication of the different modules that make it up. This allows to optimally manage the hardware of said SoC.

Next, the steps taken to configure and start both the hardware and software part of the SoC, as well as the necessary configuration of the operating system, are detailed.

Later, a control of a power converter is carried out in Simulink with the aim of implementing it in a core of the CPU that will allow the validation of its operation.

Once the necessary elements are in place, it is designed and implemented an application software focused on the implementation of a web interface, the control implementation of a power converter and communication of each of the parts.

Finally, the results obtained are presented and studied for the elaboration of a conclusion.
Contents

Acknowledgements ........................................................................................................ iii
Preface ............................................................................................................................... v
Abstract ........................................................................................................................... vii
List of figures .................................................................................................................... xiv
List of tables ...................................................................................................................... xviii
1. Introduction ............................................................................................................... 1
   1.1. Definition of initial specifications ........................................................................ 1
   1.2. Hardware platform .............................................................................................. 1
   1.3. Target application ............................................................................................... 3
2. State of the art ............................................................................................................. 5
   2.1. Power converters ............................................................................................... 5
   2.2. Embedded systems ............................................................................................. 6
   2.3. Real time operating systems .............................................................................. 7
   2.4. Asymmetric Multi-Processing and Symmetric Multi-Processing ....................... 8
3. Asymmetric Multi-Processing analysis ..................................................................... 10
   3.1. Configuration types ........................................................................................... 10
      3.1.1. Bare-metal / Bare-metal AMP ...................................................................... 10
      3.1.2. Bare-Metal (Cortex A9) / Bare-Metal (MicroBlaze) AMP ......................... 11
3.1.3. Linux / Bare-metal AMP ................................................................. 11
3.1.4. Linux / FreeRTOS AMP ................................................................. 12
3.1.5. TOPPERS SafeG ................................................................. 13
3.1.6. Xenomai ................................................................................. 14
3.2. Choice ..................................................................................... 15

4. Architecture analysis of the used SoC .......................................................... 16
4.1. Processing System ........................................................................... 17
4.1.1. Application Processor Unit ............................................................. 17
4.1.2. Memory interfaces ..................................................................... 19
4.1.3. I/O peripherals (IOP) ................................................................. 19
4.1.4. Interconnect .............................................................................. 20
4.2. Programmable Logic ....................................................................... 21

5. Software and hardware programming of the SoC ........................................ 24
5.1. Hardware setup ............................................................................. 24
5.2. Software Setup ............................................................................... 26
5.2.1. Boot ROM .............................................................................. 26
5.2.2. First Stage Boot Loader ............................................................... 26
5.2.3. U-Boot .................................................................................... 27
5.2.4. Kernel .................................................................................... 29
5.2.5. Device tree ............................................................................. 30
5.2.6. Fyle system ........................................................................... 31
List of figures

Figure 1: ZedBoard ........................................................................................................ 2
Figure 2: Target application ......................................................................................... 3
Figure 3: PV grid-connected generation system ............................................................ 4
Figure 4: Implemented application ............................................................................. 4
Figure 5: Bare-metal / Bare-metal AMP configurations ............................................ 10
Figure 6: Bare-Metal (Cortex A9) / Bare-Metal (Microblaze) AMP configuration ...... 11
Figure 7: Linux / Bare-metal AMP configuration ....................................................... 12
Figure 8: Linux / FreeRTOS AMP configuration ......................................................... 12
Figure 9: TOPPERS SafeG AMP configuration ............................................................ 13
Figure 10: Nagoya University TOPPERS SafeG ......................................................... 14
Figure 11: Xenomai configuration ............................................................................... 14
Figure 12: Zynq-7000 SoC overview ....................................................................... 16
Figure 13: Application Processor Unit ..................................................................... 17
Figure 14: Processing System Interconnect ............................................................... 20
Figure 15: Arrangement PL basic resources ............................................................... 21
Figure 16: Arrangement PL column resources ........................................................... 22
Figure 17: Default Zynq® SoC ............................................................................... 24
Figure 18: Custom Zynq® SoC ................................................................................ 25
Figure 19: Hardware implementation FPGA ............................................................. 25
Figure 20: “Create Zynq Boot Image” tool ................................................................. 32

Figure 21: Power control structure .............................................................................. 39

Figure 22: Power control simulation structure ............................................................... 39

Figure 23: PLL structure ............................................................................................... 40

Figure 24: Continuous PLL .......................................................................................... 42

Figure 25: Vin (up) and Vout (down) - PLL ................................................................. 42

Figure 26: Angular frequency estimation - PLL ............................................................. 43

Figure 27: Discrete PLL ............................................................................................... 43

Figure 28: Angular frequency estimation with Notch filter - PLL ............................... 44

Figure 29: Notch filter magnitude response ................................................................. 45

Figure 30: PR controller simulation structure ............................................................... 45

Figure 31: Continuous PR ............................................................................................ 46

Figure 32: Input (up) and Output (down) - Continuous PR ......................................... 46

Figure 33: Proportional part (up) and Resonant part (down) - Continuous PR ............ 47

Figure 34: Discrete PR ................................................................................................. 47

Figure 35: Input (up) and Output (down) - Discrete PR .............................................. 48

Figure 36: Proportional part (up) and Resonant part (down) - Discrete PR ............... 48

Figure 37: Discrete Order II PR .................................................................................. 49

Figure 38: Input (up) and output (down) - Discrete Order II PR .................................. 50

Figure 39: Proportional (up) and Resonant (down) parts - Discrete Order II PR ....... 50
Figure 40: Continuous Plant ................................................................. 51
Figure 41: Iref (up) and Iout (down) – Power Control........................................ 52
Figure 42: Vin (up) and Iout (down) – Power Control ....................................... 52
Figure 43: User interface Host - CPU0 .......................................................... 53
Figure 44: AMP interface CPU0 - CPU1 ......................................................... 56
Figure 45: Linux application SDK configuration............................................... 58
Figure 46: FCGI library configuration ............................................................. 59
Figure 47: CGI application flowchart ............................................................. 60
Figure 48: Monitor application flowchart ......................................................... 62
Figure 49: Power control-application flowchart.................................................. 63
Figure 50: System debugger .......................................................................... 65
Figure 51: GET request interface .................................................................... 67
Figure 52: POST request interface ................................................................... 69
Figure 53: Control period (µs) ........................................................................ 71
Figure 54: IRQ elapsed time (µs) ..................................................................... 71
Figure 55: Iref (up) and Iout (down) - Power Control (same as Figure 41) .......... 72
Figure 56: Vin (up) and Iout (down) - Power Control (same as Figure 42) .......... 73
Figure 57: Iref and Iout - Power Control ............................................................ 73
Figure 58: Plant Vin and Iout - Power Control .................................................... 74
List of tables

Table 1: FPGA resources ............................................................................................................. 23
Table 2: Direct Form II Notch filter coefficients ........................................................................ 44
Table 3: Direct form II PR coefficients ...................................................................................... 49
Table 4: OCM configuration registers for the power control.................................................... 55
Table 5: OCM pages for circular buffer ..................................................................................... 55
1. Introduction

In this report, a SoC is used to model an electronic power converter connected to the electrical grid, implementing the control of the converter and the user interface to generate the references and monitor the control variables.

1.1. Definition of initial specifications

The System must implement deterministically a power control at a frequency of 10 \( kHz \). For this, all control operations must be performed every 100 \( \mu s \), offering a result based on the updated value of each of the input variables.

In addition to the control of the power converter, it is necessary to store the values of a set of control variables. These variables are selected from an interface to be able to study their response in different scenarios.

The user interface has to facilitate the visualization of these variables and give the possibility of downloading all the data to a Comma Separated Values (CSV) file for a more specific analysis of the variables in environments such as MatLab.

1.2. Hardware platform

This project has been implemented on the ZedBoard [1] platform designed by AVNET and made by Digilent. The ZedBoard is a development kit based on the Xilinx Zynq®-7000 All Programmable SoC, which contains many interfaces to implement a wide range of applications.
The Figure 1 shows the interfaces available on the board, as well as other modules such as the OLED screen, the DDR3 memory and the Zynq SoC. Next, the elements used are specified:

- **Zynq**: It is the central element of the Zedboard that allows managing all the resources that are included. It consists of two ARM Cortex-A9 and a Field Programmable Gate Array (FPGA) and its architecture will be described later.
- **DDR3**: Memory in which the applications will be executed and where the operating system runs.
- **Gb Ethernet**: Communication interface between the user and the developed system.
- **FMC or Pmods interface**: System output ports.
- **SD card**: Non-volatile memory of the system, located on the backside of board.
- **XADC Header**: Analog input ports of the system.
- **USB UART**: Serial port to access the system.
- **JTAG / Debug**: SoC debug interface.
1.3. Target application

The Figure 2 shows the target application, which can be divided into four main functions:

- **Grid Tied Converter:** Located in the lower left of the Figure 2, the converter is a hardware element that is responsible for switching the DC voltage obtained by the renewable energy source to obtain a sinusoidal signal that is coupled to the electricity grid through discrete components. This switching is controlled by the signals connected to the transistors gates. A scheme of this can be seen in Figure 3 [2].
• Power Control: The power control is performed by software implemented in the CPU1. This software monitors the voltage of the grid and the current of the converter to manage the activation of the transistors gates, managing to control the current injected into said grid.

• Communication: The communication between the CPUs and between the CPU0 and the user is managed in order to obtain the data of each cycle carried out by the power control.

• Web interface: The front-end used for the application is a web page, focused on offering a simple interface to visualize and save the different variables managed by the power control.

This target application is an objective in the future but the scope of this work does not include the connection with a converter. The converter has been modeled in CPU1 to replicate the operation of the presented application. The implemented application is shown in the Figure 4.

![Figure 3: PV grid-connected generation system](image)

![Figure 4: Implemented application](image)
2. State of the art

In this section it are briefly reviewed several concepts that serve as the basis for the development of the System.

2.1. Power converters

A power converter is a system or electronic equipment that aims to convert electrical energy between two different formats. For example, obtain direct current from alternating current. The initial concept of converter can be extended to include aspects such as: efficiency, reversibility, degree of ideality, reliability, volume or technology to name the most important.

The converters can be classified according to different criteria. One of the most commonly used is to group them according to the format of the input and output energies. Basically and according to this criterion, four large groups can be established:

- **AC/DC converters, or rectifiers:** This type of converters convert alternating current, single-phase or three-phase, in continuous.
- **DC/DC converters:** This type of converters transforms a certain value of input direct current into a different value of output, with the possibility of including, in addition, galvanic isolation between input and output.
- **DC/AC converters, or inverters:** Basically, they carry out a conversion of direct current into alternating current, with the possibility of being able to control both the frequency and the effective value of the voltage or output current.
- **AC/AC converters:** In its most basic control structure, its function is to modify the effective value of the input voltage, conserving its frequency, although it can also achieve an output voltage with a frequency submultiple of the input. In the latter case they are called cycloconverters.
2.2. Embedded systems

An embedded system is an electronic system designed to perform few functions in real time, as the case may be. Embedded systems are designed to meet specific needs. In general, embedded systems can be programmed directly in the assembly language of the microcontroller or microprocessor incorporated on it, or also, using specific compilers that use languages such as C or C++. In some cases, when the response time of the application is not a critical factor, it can also be used languages interpreted as Java.

The main characteristics of an embedded system are the low cost and power consumption. An embedded system is made up of a microprocessor and software that runs on itself. However, this software needs a place where it can be saved and then executed by the processor. This could take the form of RAM or ROM, which a certain amount is used by the embedded system.

Some examples of embedded systems applications are:

- Machines and other response and automatic answering devices.
- Systems for the automatic control of moving objects.
- Mobile phones and GPS systems
- Devices of a computer network, such as routers, hubs, firewalls etc...
- Printers for computers, copiers and multifunction devices.
- Control devices for HDD and FDD.
- Drivers of a car engine, brake systems, systems closing, etc...
- Household appliances such as refrigerators, conditioners of air, security systems, microwave ovens, washing machines, TV and DVD systems, etc...
- Medical equipment.
- Personal assistants such as laptops, notebooks, etc...
- Programmable Logic Controllers (PLC) for industrial applications.
- Devices for videogames.
- Digital cameras and video cameras, etc...
2.3. Real time operating systems

A real-time system responds in a (timely) predictable way to all individual unpredictable external stimuli arrivals. The system should respond to each individual external event in a predictable way, this means before the deadline defined in the system requirement.

In a well-designed RT system, each individual deadline should be met. With the actual state of the art, it is sometimes hard and also costly to achieve this requirement.

According to [3], the deadlines to respect in RT systems might range from picoseconds to hours. Today's processors might be fast and deal with interrupts in the microsecond range, but power consumption may then become a serious problem. Portable embedded systems will have limited processor power for that reason. There exists a spectrum of technologies to deal with the different deadline requirements. The shortest deadlines (<1µs) will have to be handled by hardware without software being involved. Deadlines from 1 to 10µs can be dealt with one processor and just one program and some interrupt routines. Using a real-time operating system will help a lot in designing complex embedded system but only deadlines in the range of 10µs to 100ms or more can be reasonably handled.

A real-time operating system (RTOS) is not simply a real-time system. It is the core part of any real-time system. A real-time system includes all the system elements such as hardware, middleware, applications, communications, and I/O devices. All the elements are needed to meet the system requirements. However, RTOS provides sufficient functionality to enable a real-time application to meet its requirements. It is also important to distinguish between a fast operating system and a RTOS. Speed, although useful for meeting the overall requirements, by itself is not sufficient to determine whether a system meets the requirements for an RTOS.

As detailed in [4], real-time operating systems often support very limited virtual memory functionality. For example, “RT-Linux” keeps all real-time applications in a dedicated address space; this memory is never paged out of main memory. RTOS normally support several scheduling methods and applications have the flexibility of specifying which scheduling to use. Most RTOS systems support POSIX-defined communication and
synchronization mechanisms (e.g., semaphores, mutexes, condition variables, spin locks, signals, pipes, and message queues).

Some RTOS examples are listed below [4]:

- VxWorks
- QNX
- eCos
- RT-Linux
- Windows CE
- Symbian OS
- Free RTOS

2.4. Asymmetric Multi-Processing and Symmetric Multi-Processing

From a software perspective, there are broadly two types of multicore design: Asymmetric Multi-Processing (AMP) and Symmetric Multi-Processing (SMP).

An AMP system has multiple CPUs, each of which may be a different architecture. Each has its own address space, and each may or may not run an OS. Some kind of communication facility between the CPUs is provided for each. AMP is most likely to be used when different CPU architectures are optimal for specific activities, like a DSP and an MCU. In an AMP system, there is the opportunity to deploy a different OS on each core, for example, an RTOS and Android/Linux.

On the other hand, an SMP system has multiple CPUs like an AMP, but in this case each has exactly the same architecture. Normally, a single OS is used that runs on all the CPUs, dividing work between them (one significant difference from AMP). Some kind of communication facility between the CPUs is provided. This is normally through shared memory, but accessed through the API of the OS. Typically, SMP is used when an embedded application simply needs more CPU power to manage its workload, in much the way that multicore CPUs are used in desktop computers.
Although there is a clear distinction between AMP and SMP architectures, there is not necessarily a simple choice between the two. For example, there may be a number of cores of identical architecture which are configured to be an SMP sub-system. Logically, this sub-system looks like a single core and can, therefore, be included in an AMP system.
3. Asymmetric Multi-Processing analysis

The Zynq SoC can be configured to run independent software in each processor core. This allows having a specific core to perform temporary critical tasks where response time and execution time must be minimal [5].

These configurations are called Asymmetric Multi-Processing and can be implemented through two different approaches. The first of these, unsupervised AMP, uses a core processor to perform the tasks of coordination and management of the independent core. The other option is an AMP configuration created using the ARM TrustZone architecture, which offers different tools for the independent execution of a system in each core.

3.1. Configuration types

3.1.1. Bare-metal / Bare-metal AMP

The first of the configurations available to implement an AMP system is to execute a specific application on each CPU [6].

In this configuration, presented in Figure 5, the First Stage Boot Loader (FSBL) is responsible for loading both applications and launching the execution of CPU0, while CPU1 is initialized by the application of CPU0.
3.1.2. Bare-Metal (Cortex A9) / Bare-Metal (MicroBlaze) AMP

This is a case similar to the previous one, where two cores are arranged executing an independent application. The main difference is the execution of one of the applications in an embedded microprocessor in the programmable logic of the SoC [7]. This configuration can be observed in the Figure 6.

![Figure 6: Bare-Metal (Cortex A9) / Bare-Metal (MicroBlaze) AMP configuration](image)

3.1.3. Linux / Bare-metal AMP

Another configuration, represented in Figure 7, is the one that implements a Linux-based operating system for the main core of the CPU, leaving CPU1 to execute a specific application. The CPU1 application is loaded by the FSBL and Linux is responsible for starting its execution [8].
The communication between both CPUs can be done through the on-chip memory (OCM) by disabling the cache to improve the determinism.

3.1.4. Linux / FreeRTOS AMP

In this AMP configuration the main core of the CPU runs the Linux operating system and the CPU1 uses an RTOS, in this case, FreeRTOS. This configuration allows Linux to access in read or write mode the memory addresses used by the RTOS [9], as seen in Figure 8.
Other operating systems in real time can be used under the same configuration, with FreeRTOS being an implementation example of one of them.

3.1.5. TOPPERS SafeG

TOPPERS SafeG is an RTOS / GPOS that implements the ARM TrustZone architecture. To do this, the system is divided into two "worlds": the secure “world” and the non-secure “world”.

![TOPPERS SafeG AMP configuration](image)

*Figure 9: TOPPERS SafeG AMP configuration*

As it can be observed in Figure 9, the secure “world” can access all the resources of the system and its operation cannot be affected by the non-secure “world”. The non-secure “world” can only access unsecured resources as shown in the Figure 10.
3.1.6. Xenomai

Xenomai is a dual-kernel, open source real time solution for Linux. It uses a kernel for real-time applications and a Linux kernel for non-critical applications. It implements the so-called RTOS "Skin": an API that includes implementations seen in different RTOS, facilitating the applications development in the Xenomai Kernel. The Figure 11 represents this configuration.

Figure 11: Xenomai configuration
3.2. Choice

Due to the initial specifications of the System, where the power control must be carried out deterministically in 100 µs, the configuration implemented is the Linux / Bare-metal AMP.

This configuration is the one that best adapts to the requirements of the project. It has a kernel with Linux for communication and presentation of the interface, while the other core performs the tasks of power control independently.

By not having a layer below the application of power control, a higher response speed is ensured. Achieving a deterministic system with this configuration depends on disabling the caches and resources that favour the amount of operations that a CPU can perform.

By disabling the OCM memory cache, the power control and data dump to the interface without compromising the temporary system requirements can be performed.
4. Architecture analysis of the used SoC

As mentioned in “Introduction” section, the Zedboard used contains a Zynq-7000 SoC (xc7z020clg484). This SoC integrates a feature-rich dual core ARM® Cortex™-A9 MPCore™ based processing system (PS) and the Xilinx programmable logic (PL) in a single device. The ARM CPUs are the heart of the PS which also includes on-chip memory (OCM), external memory interfaces and a large set of I/O peripherals. The SoC architecture [10] can be seen in Figure 12.

![Zynq-7000 SoC diagram](image)

*Figure 12: Zynq-7000 SoC overview*
The Zynq-7000 SoC is composed of the following major functional blocks:

- **Processing System (PS)**
  - Application processor unit (APU)
  - Memory interfaces
  - I/O peripherals (IOP)
  - Interconnect
- **Programmable Logic (PL)**

4.1. Processing System

4.1.1. Application Processor Unit

The application processor unit (APU) contains different modules that offer high performance processing characteristics.

*Figure 13: Application Processor Unit*
As seen in Figure 13, in it are located the two ARM Cortex-A9 processors implemented in the SoC sharing 512 KB of L2 cache. These processors implement SIMD (Single Instruction Multiple Data) co-processors called NEON designed for audio, video and 3D graphics processing tasks. It is necessary to mention the Memory Management Unit (MMU) module that resides in each of the processors. The MMU performs the function of memory protection and address translation. This module is close to the L1 and L2 caches, translating the virtual addresses with which the processor works in the physical addresses.

The accelerator coherency port (ACP) is responsible for facilitating communication between the APU and the Programmable Logic (PL). This allows to communicate the PL with the OCM memory and the L2 cache without losing coherence with the data of the L1 cache of the cores.

The processors are organized in a multi-process configuration. The snoop control unit (SCU) is responsible for maintaining consistency between the L1 caches of both processors.

The APU also contains an on-chip memory (OCM) of 256 KB shared by both processors. This memory contains a non-volatile read-only zone that stores the bootloader with which the CPUs are initialized.

Another module is the generic interrupt controller (GIC) in charge of managing the interrupts received by the PS and the PL. This controller allows enabling, disabling and prioritizing interrupt sources and sending them to the selected CPU.

Finally, the APU contains a 32-bit watchdog timer and a 64-bit global timer.
4.1.2. Memory interfaces

The memory interfaces support several memory technologies, including:

- DDR Controller: Supports DDR3 and DDR2 variations.
- Quad-SPI Controller: This interface can be a primary boot device.
- Static Memory Controller (SMC): Supports NAND and SRAM / NOR memories.

4.1.3. I/O peripherals (IOP)

The input and output peripherals of the SoC correspond to different interface standards. These interfaces are designed for communication with modules external to the PS. The available controllers are the following:

- GPIO (x54)
- Gigabit Ethernet Controllers (x2)
- USB Controllers (x2)
- SD/SDIO Controllers (x2)
- SPI Controllers (x2)
- CAN Controllers (x2)
- UART Controllers (x2)
- I2C Controllers (x2)
- PS MIO I/O
4.1.4. Interconnect

The processing system implements different interconnect technologies which are shown in the Figure 14.

Figure 14: Processing System Interconnect
4.2. Programmable Logic

The Programmable Logic (PL) is based on the Artix®-7 FPGA logic and has multiple interfaces for connection to the Processing System (PS). This allows to implement in the logic different hardware elements, in order to accelerate the processing of different applications.

The PL can be configured by the PS at the beginning or at any moment. In addition, the PL can be partially reconfigured, allowing to fix a part of the hardware and reconfigure the remaining logic to allow different implementations.

The Figure 15 [11] shows the basic FPGA resources such as Input/Output Blocks (IOBs), Configurable Logic Block (CLBs) or programmable interconnection lines.

An IOB is the input and output interface for the signals outside the FPGA. It allows to configure pin parameters such as voltage (1.2 to 3.3 V), impedance, delay or different types of output.
A CLB contains a pair of slices, and each slice is composed of four 6-input LUTs, four flip-flops and the two carry lines. These slices have no connection between them, as can be seen in the upper left area of Figure 15 [11].

To connect the different elements, an interconnection matrix is available throughout the FPGA. This allows to communicate any element with another as long as the delay between the two points does not exceed the period of the clock used. In addition, different clock lines are available to address the different clocking requirements, for example, high fanout, short propagation delay or extremely low skew.

![Figure 16: Arrangement PL column resources](image)

Figure 16 [11] shows that, throughout the FPGA, blocks of RAM (Block RAM) are vertically arranged to implement different hardware resources with a high memory consumption, for example, FIFO between two clock domains or a large LUT. Each RAM block has 36KB of dual port memory with a configurable width of up to 72 bits.
Like the Block RAM, the digital signal processor (DSP) are arranged vertically. These allow performing different arithmetic operations in addition to configuring them in a cascade structure to implement complex functions without consuming FPGA logic fabric resources. Each DSP consist of a 25×18 bit two’s complement multiplier and a 48-bit accumulator.

Table 1 shows the resources available in the Programmable Logic of the used SoC:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Number of elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Slice</td>
<td>13300</td>
</tr>
<tr>
<td>LUT</td>
<td>53200</td>
</tr>
<tr>
<td>Block RAM</td>
<td>140</td>
</tr>
<tr>
<td>MMCM/PLL</td>
<td>4</td>
</tr>
<tr>
<td>DSP</td>
<td>220</td>
</tr>
<tr>
<td>I/O</td>
<td>200</td>
</tr>
</tbody>
</table>
5. Software and hardware programming of the SoC

In this section, the necessary configuration to obtain an AMP system in the Zynq SoC is detailed. First, it is necessary to configure the hardware associated with the system, configuring the SoC parameters and defining the hardware implemented in the FPGA. Second, the different elements that make up the software are explained and compiled. Third, it is presented the initialization of the previous hardware and software through an SD card. Finally, the basic configuration of the Linux operating system is performed, installing the necessary tools for the system.

5.1. Hardware setup

The hardware part of the system is configured using the Vivado® [12] software from Xilinx® [13]. After creating a new project by selecting the Zedboard as target hardware, the SoC configuration block is introduced into the system.

![ZYNQ Processing System](image)

*Figure 17: Default Zynq® SoC*

This component (Figure 17) is modified to add two communication interfaces (SPI and IIC) for read converters in a real test environment connected to the electrical grid. The aforementioned interfaces are part of the SoC peripherals. Their pins are redirected to the FPGA so that they can be routed to the output pins of the ZedBoard (for example, to the Mezzanine connector).

The private FIQ interrupt line for CPU1 is also enabled from the FPGA, which enables the priority interrupt from the CPU1 to be enabled. The resulting SoC of the previous modifications are shown in Figure 18.
The only component implemented with the logical resources of the FPGA is a Timer managed through the AXI bus. This Timer can generate an interrupt once an interval has been defined to set the system frequency of the CPU1 through its private interrupt line. The XADC configuration wizard is also added.

Figure 18: Custom Zynq® SoC

Figure 19 shows the interconnection of the different system components, where the IO pins corresponding to the DDR memory and the SPI and IIC interfaces are observed. The
system reset management module and the interconnection module of the AXI bus are also visible.

For the connection to a power controller, it is necessary too to implement a PWM module for the management of power transistor gates and to use the XADC interface.

5.2. Software Setup

5.2.1. Boot ROM

The Zynq® boot starts by running the code that resides in the ROM memory that is part of the OCM. Its main function is to load the FSBL from the *BOOT.BIN* boot binary.

5.2.2. First Stage Boot Loader

The first stage boot loader (FSBL) is the first external application that executes the SoC after its power up. This software provided by Xilinx is responsible for dumping the user application to memory, which is done in order to be executed in addition to programming the FPGA. This FPGA is programmed through the *.bit* file generated in the build through Vivado. Once the execution is complete, the FSBL executes the application loaded in RAM. All these operations are executed under the CPU0 using the OCM memory.

The main objective of this bootloader is to introduce the execution of U-Boot, which will be responsible for initializing the AMP configuration.

This application is built using the Xilinx SDK software, and it is necessary to create a new project of the type "Xilinx-> Application Project", selecting as target hardware the design created by Vivado. Then, it must be selected as the template: "Zynq FSBL".
5.2.3. U-Boot

U-Boot or Universal Boot Loader is an open source bootloader used in systems of different architectures like PowerPC, ARM, XScale, MIPS, Coldfire, NIOS, Microblaze and x86. Allows to configure basic system parameters, load applications into memory or boot the kernel of an operating system.

In order to build this application, the following steps have been followed:

1. Download the source code, available in the Xilinx repository of GitHub.

   ```
   git clone https://github.com/Xilinx/u-boot-xl
   git checkout master
   ```

2. Build U-Boot for the ARM architecture and for the configuration of the Zedboard.

   ```
   cd u-boot-xl
   make CROSS_COMPILE=arm-xilinx-linux-gnueabi- zynq_zed_config
   make
   ```

3. Add to the binary name the extension .ELF (Executable and Linkable File) supported by Xilinx.

   ```
   cp u-boot u-boot.elf
   ```

In addition to obtaining the application that will be executed after the FSBL, it is necessary to execute several commands to: launch the system, configure the Linux kernel and load the power control application for the CPU1. These commands are expected by U-Boot in the file uEnv.txt, whose content is shown below:
First of all, the boot parameters of the Linux kernel are set by using the `bootargs` variable of U-Boot. The simplest way to configure the kernel for use an AMP configuration is to set the parameter of the number of CPUs available (`maxcpus`) to 1. In this way the kernel will only have access to the CPU0 of the SoC. Once the access to a single CPU is limited, the memory sections that the kernel can manage are defined, setting `mem` to half of the DDR memory available in the hardware. Another necessary parameter is to select the Linux serial access interface device, in this case, `/dev/ttyPS0` at a rate of 115.2 kbps. This allows access to the Linux terminal through the serial port of the Zedboard once the system is initialized. The rest of the `bootargs` parameters indicate to the kernel the file system used. The root of the system is set in the second partition of the MMC `/dev/mmcblk0p2` memory in `ext4` format.

Secondly, different applications are loaded into memory using the `fatload` command. The kernel and the DTB (Device Tree Blob) are dumped in the first half of the DDR memory, and the application of the CPU1 in the second half.

Finally, using the `bootm` command, the Linux kernel is loaded, passing the dump addresses of the kernel and the DTB as parameters.
5.2.4. Kernel

The Linux kernel is the main part of the operating system. It is responsible for three tasks: to manage the applications access to the different hardware resources, to temporarily multiplex the execution of the applications and to serve as the basis for different distributions.

In the same way as U-Boot, the Linux kernel must be configured and built for the ARM architecture of the Zynq® SoC:

1. Install `ncurses` library required for the compilation.

```
sudo apt-get install libncurses5-dev
```

2. Obtain the source code, available in the Xilinx repository of GitHub.

```
git clone https://github.com/Xilinx/linux-xlnx.git
```

3. Add to `PATH` the `mkimage` U-Boot application, in order to generate the compatible Kernel. In this case, a symbolic link is created in `/usr/local/bin`.

```
ln -s u-boot-xilinx/tools/mkimage /usr/local/bin
```

4. Build the kernel for the ARM architecture and for the configuration of the Zedboard.

```
cd linux-xlnx

make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-xilinx_zynq_defconfig UIMAGE_LOADADDR=0x8000 uImage
```
5.2.5. Device tree

The kernel has been compiled to be compatible with the Zynq® SoC architecture used, but it needs to know the hardware available on the Zedboard. This hardware is defined by a structure called device tree blob (DTB), which can be generated from a .dts file.

In the source code of the kernel there are different .dts files for different supported boards, from which we obtain the definition of the basic components of the Zedboard. Among them are the USB serial interface, the physical layer of the Ethernet protocol and the memory controller of the MMC and the DDR.

Here are the steps to obtain the .dtb binary:

1. Add to PATH the kernel DTC application, in order to generate the compatible .dtb file. In this case, a symbolic link is created in/usr/local/bin.

   ```bash
   ln -s linux-xlnx/scripts/dtc/dtc /usr/local/bin
   ```

2. Obtain the .dts file that defines the SoC hardware.

   ```bash
   cp linux-xlnx/arch/arm/boot/dts/zynq-7000.dtsi .
   ```

3. Obtain the .dts file that defines the Zedboard hardware.

   ```bash
   cp linux-xlnx/arch/arm/boot/dts/zynq-zed.dts .
   ```

4. Build the .dts file to obtain the .dtb binary using the DTC compiler.

   ```bash
   dtc -I dts -O dtb -o devicetree.dtb zynq-zed.dts
   ```
5.2.6. File system

As a last step to get a functional OS, a file system is needed. To implement it, there are two possibilities.

The first one is to use a RAMDisk as a file system, keeping in RAM all the system modifications that are made. It has the advantage of obtaining higher transfer speeds than a non-volatile memory, but any modification is lost when the system is restarted.

The second of the implementations consists of using a non-volatile memory as a file system, such as a hard disk or an SD card.

For the system of this project it has been decided to use the SD card allowing to save the different configurations that are made to the system, as well as being able to permanently save the power control data. The file system used is based on the release of Ubuntu 14.04 LTS (adapted by linaro) for a minimum configuration of the operating system. The use of a Long Term Support (LTS) version ensures the possibility of installing any program from the updated Ubuntu repositories.

It can be downloaded using the following command:

```
```

5.3. Boot on SD card configuration

In this section the SD card in which the system resides is configured. This consists on set two partitions. The first of these is in charge of save all the files necessary for initialization, while the second one is the file system.
5.3.1. Wrap bootloader

In order for the Boot ROM initializes correctly the FSBL, it is necessary to generate the binary BOOT.BIN package. Within the SDK software, it is possible to generate said binary package through "Xilinx Tools -> Create Zynq Boot Image", as seen in Figure 20.

![Create Zynq Boot Image](image)

*Figure 20: “Create Zynq Boot Image” tool*

The image is generated from 3 files and the configuration is saved in the indicated .BIF file. The first file is FSBL.elf, which must be added as a bootloader type partition. The
two remaining files must be added as datafile partition, and correspond with the programming .bit files of the FPGA and U-Boot, that will be executed after the completion of the FSBL.

The resulting image must be renamed to BOOT.BIN.

5.3.2. Format SD card

As we have seen previously, two partitions are necessary for the operation of the boot system.

```bash
sudo fdisk /dev/sdb
sudo mkfs -t vfat -n ZED_BOOT /dev/sdb1
```

The first one must be formatted as a FAT32 file system with a size of at least 64 MB under the name ZED_BOOT. The size of this partition has been set to 256 MB through fdisk to be able to load heavy applications without having to reformat the card.

```bash
sudo mkfs -t ext4 -L ROOT_FS /dev/sdb2
```

The second one must be formatted as an EXT4 file system under the name ROOT_FS. The size of this partition is the remainder of the SD card.

5.3.3. Load boot files

The boot files that are saved in the ZED_BOOT partition are the following:

- BOOT.BIN (FSBL, HW.bit, U-Boot)
- uEnv.txt (U-Boot commands)
- ulmage (Kernel)
- devicetree.dtb (Device Tree)
- app_cpu1.bin (CPU1 app)

These files can be copied by hand from the file browser.
5.3.4. Load file system

The process for loading the file system into the \textit{ROOT\_FS} partition is as follows:

1. Unmount SD card partitions.

\begin{verbatim}
sudo umount /media/ZED\_BOOT
sudo umount /media/ROOT\_FS
\end{verbatim}

2. Create a temporary folder for the file system.

\begin{verbatim}
mkdir -p /tmp/linaro
\end{verbatim}

3. Copy the file system to temporary folder.

\begin{verbatim}
sudo cp linaro-saucy-developer-20140410-652.tar.gz /tmp/linaro/fs.tar.gz
\end{verbatim}

4. Unzip the file system.

\begin{verbatim}
sudo tar -xvf fs.tar.gz
\end{verbatim}

5. Mount the second SD card partition in a new folder.

\begin{verbatim}
sudo mkdir -p /tmp/sd\_ext4
sudo mount /dev/sdb2 /tmp/sd\_ext4
\end{verbatim}

6. Synchronize the unzipped file system with the SD card.

\begin{verbatim}
sudo rsync -a /tmp/linaro/binary /tmp/sd\_ext4
\end{verbatim}
7. Unmount the SD card partition.

```bash
sudo umount /tmp/sd_ext4
```

5.4. Linux configuration

In this configuration section, the necessary system tools are installed and configured. Due to the need to have a web interface for system management, these tools focus on connectivity.

5.4.1. Ethernet

In order to obtain an Internet connection through the Ethernet port, it is necessary to configure the `/etc/network/interfaces` file.

```bash
sudo su

echo auto eth0 >> /etc/network/interfaces

echo iface eth0 inet dhcp >> /etc/network/interfaces
```

This configuration is done by executing the above commands through a serial terminal. These commands add two new lines to the file. The first enables the automatic start of the Ethernet interface through the kernel and the second configures the DHCP protocol to request the router for an available IP.

```bash
ifup eth0
```

Finally, the interface is started manually using the previous command to avoid restarting the system.
5.4.2. SSH server

Once the Internet connection is established, it is necessary to install an SSH server to enable the remote connection to the system.

```
apt-get install openssh-server
```

Through the Ubuntu command `apt-get install` it is installed an open source server.

```
sshpass -p linaro ssh linaro@192.168.1.2

sudo su
```

Once the installation is complete, the system is accessible from any SSH client, indicating the user "linaro" and the current IP address (with password: "linaro").

5.4.3. HTTP server

Analogously to the SSH server, the web server is installed through the Ubuntu repositories.

```
apt-get install nginx
```

Nginx is an open source web server optimized to offer low memory consumption. The `/etc/nginx/sites-available/default` file contains the server configuration, where resources are selected in function of the requests.

```
mkdir /media/data/

mkdir /media/data/images

mkdir /media/data/www
```

The first step to configure the server is to create the folders where the HTML documents are served.
The second step is to edit the configuration file to establish access to the created folders.

```
server {
    location / {
        root /media/data/www;
    }
    location /images/ {
        root /media/data/images;
    }
}
```

```
nginx -s reload
```

Finally, the Nginx configuration is reloaded. In this way, after an HTTP request from a web browser to the system IP address, the requested file is returned if it exists in `/media/data/www`.

5.4.4. FCGI server

A FCGI (Fast Common Gateway Interface) server is a protocol for interconnecting interactive programs with a web server, in this case, Nginx.
There are three requirements to implement an FCGI application. The first one is to build an application over a library that supports FCGI, which will be done in the “8. Software development” section. The second is to have a program capable of launching applications compiled for FCGI, so it is necessary to install the following application from the Ubuntu repository:

```
apt-get install spawn-fcgi
```

The remaining requirement is to configure the web server to act as a proxy between the FCGI application and the host PC. To do this, the following entry is added to the Nginx configuration file:

```
location ~ \.cgi$ {

    root /media/data/cgi;

    fastcgi_pass 127.0.0.1:8088;

    #fastcgi_index index.php;

    include fastcgi_params;

}
```
6. Control power design

To be able to test the system is necessary to define a power control to be executed in the CPU self-dependent core. The control structure of the converter is shown in Figure 21. In the lower part, the structure of the converter is observed. This converter is responsible for transforming the continuous voltage to an alternating voltage, in order to inject current into the network.

![Figure 21: Power control structure](image)

This control is implemented by idealizing the behaviour of the converter and the PWM modulator through a continuous converter model, which allows to simulate the performance of the interface without having a converter. This model is implemented below.

![Figure 22: Power control simulation structure](image)
In Figure 22 it can be seen the Simulink® [14] diagram with which the simulations are performed, both continuous and discrete.

This control is divided in three main blocks, which are detailed below:

1. Phase-locked loop (PLL)
2. Proportional-resonant controller (PR)
3. Plant

6.1. Phase-locked loop

A phase locked loop (PLL) [15] is a closed loop system in charge of generating a sinusoid with the same phase of the input signal. This allows the synchronization of the power control with the phase of the electrical grid. A PLL structure is shown in Figure 23.

![Figure 23: PLL structure](image)

The first element of a PLL is the phase detector (PD), which function is to compare the input sinusoid with the voltage controlled oscillator (VCO) output, obtaining a signal proportional to the phase error. For this, the signal measured at the input and the VCO signal output are multiplied, obtaining:

\[ v = v_{grid} \cdot \sin(\omega_{grid} \cdot t + \theta_{grid}) \]
\[ v' = \cos(w_{PLL} \cdot t + \theta_{PLL}) \]

\[ \epsilon = 0.5 \cdot v_{grid} \cdot \sin((w_{grid} - w_{PLL}) \cdot t + (\theta_{grid} - \theta_{PLL})) + 0.5 \cdot v_{grid} \]
\[ \cdot \sin((w_{grid} + w_{PLL}) \cdot t + (\theta_{grid} + \theta_{PLL})) \]

In steady state operation, it can be assumed that \( w_{grid} = w_{PLL} \), and, for small angles, \( \sin(\theta) \simeq \theta \). Additionally, the frequency component \( w_{grid} + w_{PLL} \) can be minimized with a Notch filter centered in \( 2 \cdot w_{grid} \), obtaining:

\[ \epsilon \simeq 0.5 \cdot v_{grid} \cdot (\theta_{grid} - \theta_{PLL}) \]

The error obtained is the loop filter input, which in this case is an integral proportional controller (PI), defined as:

\[ y_{PI}(s) = K_p + \frac{K_i}{s} \]

By last, the PI filtered output feeds the VCO in charge of generate a sinusoid signal from this error and the target frequency \( w_o = 2 \cdot \pi \cdot f_c \):

\[ v' = \cos((w_{error} + w_o) \cdot t + \theta_{PLL}) \]
6.1.1. Continuous PLL

In Figure 24 it can be seen the continuous model of the PLL implemented in Simulink®, using a multiplier without Notch filter as phase detector. The input and output signals of the PLL are shown in Figure 25.

Both signals have an approximately similar phase, so it can be seen the correct operation of the PLL. However, the estimated output frequency (the input of the "Int2" block in
the diagram of the Figure 24, ideally \( w_\phi = 2 \cdot \pi \cdot 50 \approx 314 \) has a ripple with a frequency component of \( 2 \cdot f_{\text{grid}} \), shown in Figure 26.

![Figure 26: Angular frequency estimation - PLL](image)

6.1.2. Discrete PLL

To simulate the PLL in the discrete domain the structure in Figure 27 is created in Simulink®, replacing the integrators by discrete integrators on a frequency of \( f_s = 10 \, kHz \).

![Figure 27: Discrete PLL](image)
With the objective of minimize the ripple of the frequency estimated by the PLL it has been added a Notch filter centered in 100 Hz and with a band width of 10 Hz.

![Figure 28: Angular frequency estimation with Notch filter - PLL](image)

As it can be seen in Figure 28, the frequency error is minimized as it advances into the steady state.

The implementation of this filter requires the discrete components calculation, in an Order II stage in this case. For this, with the aid of FDATool® [16], the coefficients for $f_s = 10 \text{ kHz}$, $f_c = 100 \text{ Hz}$ and $BW = 10 \text{ Hz}$ are obtained, as shown in Table 2.

<table>
<thead>
<tr>
<th>Numerator</th>
<th>Denominator</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_0 = 0.99840395573378138$</td>
<td>$a_0 = 1$</td>
</tr>
<tr>
<td>$b_1 = -1.9928676671816614$</td>
<td>$a_1 = -1.9928676671816614$</td>
</tr>
<tr>
<td>$b_2 = 0.99840395573378138$</td>
<td>$a_2 = 0.99680791146756276$</td>
</tr>
</tbody>
</table>
With those parameters a filter with an attenuation of 28 dB is obtained for the wanted frequency, as it can be seen in Figure 29.

![Figure 29: Notch filter magnitude response](image)

6.2. Proportional-resonant controller

A proportional-resonant controller (PR) has ideally an infinite gain on frequency \( w_1 \), at which it is designed, \( w_1 = 2 \cdot \pi \cdot 50 \) in this case. This allows a resonant controller to follow a sinusoidal signal without error in steady state.

The continuous mode of this controller is defined by the structure in Figure 30.

![Figure 30: PR controller simulation structure](image)
6.2.1. Continuous PR

In the diagram in Figure 31 is implemented in Simulink® the continuous mode of the converter to check its performance.

![Figure 31: Continuous PR](image)

Figure 32: Input (up) and Output (down) - Continuous PR
As it can be shown in Figure 32 and Figure 33, the controller follows correctly the input signal, so the proportional part grows as the controller entries in steady state, minimizing the proportional part effect.

6.2.2. Discrete PR

It is needed to discretize the resonant proportional controller model for the working frequency of the power control to be implemented, in this case $f_s = 10kHz$. 

![Figure 34: Discrete PR](image)
To do this, it is established in Simulink® a discrete simulation environment at said frequency, implementing the scheme of Figure 34.

Figure 35: Input (up) and Output (down) - Discrete PR

Figure 36: Proportional part (up) and Resonant part (down) - Discrete PR

Figure 35 and Figure 36 show the correct performance of the controller, which is able to follow the input signal.
6.2.3. Discrete Order II PR

To implement the resonant branch of the PR in an Order II structure, it is necessary to unify both integrators.

Given $G_{PR}(s)$, it is discretized through the impulsive invariant method obtaining as a result $G_{PR}(z)$, and their coefficients are shown in Table 3.

$$G_{PR}(z) = K_p + K_i \cdot T_s \cdot \frac{1 - z^{-1} \cdot \cos(w_1 \cdot T_s)}{1 - 2 \cdot z^{-1} \cdot \cos(w_1 \cdot T_s) + z^{-2}}$$

<table>
<thead>
<tr>
<th>Numerator</th>
<th>Denominator</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_0 = 1$</td>
<td>$a_0 = 1$</td>
</tr>
<tr>
<td>$b_1 = -0.9995$</td>
<td>$a_1 = -1.999$</td>
</tr>
<tr>
<td>$b_2 = 0$</td>
<td>$a_2 = 1$</td>
</tr>
</tbody>
</table>

In Figure 37, said structure of Order II is implemented to check if the performance is correct, once both integrators are unified inside of the same structure.
As seen in the section “6.2.2. Discrete PR”, the results of the simulations shown in Figure 38 and Figure 39 are satisfactory, as the error in steady state is null.
6.3. Plant

Due to the nature of the project, centered on the design of the control interface, the model of the plant has been simplified, managing the input voltage to set at its output a current proportional to the selected resistance and inductance values:

\[ G_p(s) = \frac{1}{R + s \cdot L} \]

The Figure 40 shows the Simulink diagram that implements \( G_p(s) \):

6.4. Power control

Once the different elements of the power control (PC) have been seen separately, the complete system of Figure 22 is simulated.
The Figure 41 shows how the output current reaches the 20 $A_{\text{peak}}$ requested by the reference current, this allows to arbitrarily modify the requested value, being able to control the current destined to the electrical grid.

Finally, is seen the operation of the plant in Figure 42, which receives an input voltage of about 33 $V_{\text{peak}}$, resulting in a current of 20 $A_{\text{peak}}$ for the selected values of $R = 0.5 \Omega$ and $L = 5 \, mH$. 
7. Software design

The software of this project is divided in two main parts: the communication between the Host and Linux, and the communication between Linux and the power control CPU.

7.1. User interface Host – CPU0

As seen previously, HTTP and FCGI servers have been installed on Linux. These servers act as a gateway to the GET and POST commands that a browser sends to the system IP address.

![Diagram of User interface Host - CPU0]

*Figure 43: User interface Host - CPU0*
The Figure 43 shows the steps covered during the request of the page `amp.cgi` under a POST request. The GET requests follow the same route but without reading data in memory, offering the initial configuration of the web page.

The HTTP request arrives to the Nginx server, which redirects the request to the FCGI application once the `.cgi` extension is recognized. For the `.cgi` files, Nginx acts as a proxy.

The CGI application must process the request, saving the parameters sent in the request. Based on the parameters received, the application identifies the variables requested. Next, the data is read in memory and are associated by columns to the different variables requested. This allows to define an array of data with which a variable in JavaScript is used to represent that generated data. Since the web page only accesses the CGI application, it is the one that must include the necessary HTML for the visualization of the contents.

At the end of the CGI application response, the Nginx server completes the transfer. This allows the host to view the implemented web page.

7.2. AMP interface CPU0 – CPU1

The communication between the CPUs of the SoC is carried out through the memory integrated in the SoC itself (OCM). This memory is used to dump the data from CPU1 to CPU0, transferring the variables from the power control to Linux.

A small section of the OCM memory serves as configuration registers for the CPU1. These registers are reflected in Table 4, in which the position of the beginning of the memory is the address 0xFFFF8000 of the MMU.

Four pages of this memory are used as a circular buffer for data dump (Table 5).
Table 4: OCM configuration registers for the power control

<table>
<thead>
<tr>
<th>Name</th>
<th>Offset</th>
<th>Description</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEART_BEAT</td>
<td>0x0000</td>
<td>Interrupt counter</td>
<td>1</td>
</tr>
<tr>
<td>BUFFER_INDEX</td>
<td>0x0004</td>
<td>Write index in the OCM buffer</td>
<td>1</td>
</tr>
<tr>
<td>START_CMD</td>
<td>0x0008</td>
<td>Initializes CPU1</td>
<td>0</td>
</tr>
<tr>
<td>STOP_CMD</td>
<td>0x000C</td>
<td>Pause CPU1</td>
<td>0</td>
</tr>
<tr>
<td>SHUTDOWN_CMD</td>
<td>0x0010</td>
<td>Turn off CPU1</td>
<td>0</td>
</tr>
<tr>
<td>NUM_VARS</td>
<td>0x0100</td>
<td>Number of variables to be transferred from the control</td>
<td>0</td>
</tr>
<tr>
<td>SEL_VARS1</td>
<td>0x0104</td>
<td>32 bits that activate the first 32 variables individually</td>
<td>0</td>
</tr>
<tr>
<td>SEL_VARS2</td>
<td>0x0108</td>
<td>Not used</td>
<td>0</td>
</tr>
<tr>
<td>SEL_VARS3</td>
<td>0x010C</td>
<td>Not used</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0x0110</td>
<td>Floating point value of the inductance</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0x0114</td>
<td>Floating point value of the capacity</td>
<td>0</td>
</tr>
<tr>
<td>R</td>
<td>0x0118</td>
<td>Floating point value of resistance</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>0x011C</td>
<td>Floating point value of the current</td>
<td>0</td>
</tr>
<tr>
<td>V</td>
<td>0x0120</td>
<td>Floating point value of voltage</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5: OCM pages for circular buffer

<table>
<thead>
<tr>
<th>OCM Buffer</th>
<th>Direction MMU</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAGE0</td>
<td>0xFFFF1000</td>
</tr>
<tr>
<td>PAGE1</td>
<td>0xFFFF2000</td>
</tr>
<tr>
<td>PAGE2</td>
<td>0xFFFF3000</td>
</tr>
<tr>
<td>PAGE3</td>
<td>0xFFFF4000</td>
</tr>
</tbody>
</table>

As seen in the previous section, web requests are processed in the CGI application. This application configures the registers of Table 4 according to the data received by the web form. This is the first step in the communication between both CPUs, which is represented in Figure 44.
The CPU1 monitors the configuration of the OCM registers in its main loop. When the start command is activated, the interrupts of a timer that serves as the time base for the control are enabled. As the interrupt handler generates new data to be downloaded, these are written to the OCM memory. The write index of the buffer is adjusted to the data that has been dumped.
In parallel, CPU0 queries the data write address. In case the address is increased, the new data is read. Once the reading is done, the reading pointer is equal to the writing pointer. Finally, the data stored in the RAM is transferred to a file in the Linux file system.

CPU1 remains running while the main loop does not detect a stop command.
8. Software development

In the previous section we have explained the functions that each of the CPUs must attend. To implement these functions, three applications are programmed in C. This language has been chosen for compatibility with the functions provided by Xilinx for the management of resources within the SoC.

Two of these three applications are designed for Linux under the CPU0: one in charge of managing the communication with the Host and another one in charge of managing the OCM buffer for reading data.

The last application corresponds to a baremetal design, where CPU1 will have the only function of implementing this code. This application manages the power control and dumps the data to the OCM memory through the buffer.

8.1. Linux applications

To be able to build compatible applications with Linux it is enough to create a new project in the SDK indicating “Linux” in “OS Platform” field, as shown in Figure 45.

![Figure 45: Linux application SDK configuration](image-url)
8.1.1. CGI application

The first of the applications developed is the CGI application. This application runs under the FCGI server process within Linux, which allows the Host to communicate with Linux.

To develop in C an application that can be executed as FCGI, it is necessary to implement this library. To do this, the C library is downloaded in the system where the build is made, and \
*-lfcgi* is added as a parameter to the compiler. This library is available in the “libs” folder of the project files. Figure 46 shows the configuration of the SDK to be able to build with this library.

![Figure 46: FCGI library configuration](image)

The rest of the tools used by the application are defined in the reference functions provided by Xilinx. This means that no further modifications are necessary to implement the source code of the application.
In Figure 47, the flow chart of the CGI application is shown, which will be detailed below.

As a first step, variables and OCM records are initialized to a predefined state. If the OCM memory has valid values in the configuration registers, this configuration is loaded into the software variables. This allows to not affect the operation of the CPU1 in case of re-launching the application.

After initialization, the application implements an infinite loop that handles requests by calling the function `FCGI_Accept()` from the FCGI library. When a request arrives at the server this function is responsible for accepting it.
The next step is to process the received request. The request method and the size of its content are received as environment variables. Consulting the value of the environment variable `REQUEST_METHOD` identifies whether it is a GET or POST request. GET requests do not modify any variable of the application, keeping the value of these variables to the last value configured by a POST request.

POST requests are attended to configure both the web page to respond and to manage the OCM records for the configuration of the CPU1. As an example, the string received in a POST request is displayed:

```
L=0.005&R=0.5&iref=20&vref=220
```

This string is processed to obtain the values of inductance, resistance and reference current requested. Once obtained, these values are saved into the application variables so that they are written in the corresponding OCM registers and the web is updated.

Once the configuration is done, the file in memory that contains the power control data is read and this data is added to the JavaScript. The variables to be downloaded are selected through the POST request of the web form.

As a final step, it is returned the JavaScript and HTML, resulting in the web. This web is configured dynamically according to the values of the application variables.

8.1.2. Monitor application

This application is executed in parallel to the one explained in the previous section. Its main function is to dump the data from the OCM buffer to a binary file with each of the values written in the circular buffer by the CPU1. This application monitors three OCM records: `START_CMD`, `STOP_CMD` and `BUFFER_INDEX`.

First, as shown in Figure 48, the application waits for the command to execute the power control through the \textit{START\_CMD} register. Once the initialization order has been received, the program deletes the data contained in the output file.

Second, the application enters the main loop, where \textit{STOP\_CMD} and \textit{BUFFER\_INDEX} are monitored. In case the value of the \textit{STOP\_CMD} register is active, the application returns to the initial state, waiting for a new start command and with the available data in the output file. Otherwise, the value of the write index in the OCM buffer is read. If the value of the write index is different from the value of the read index, the buffer is read until the read and write indexes have the same value.

\textit{Figure 48: Monitor application flowchart}
Finally, the read variables are written in the output file of the application.

Through this configuration, the data file dumped by the power control is always available on the SD card. This file is read by the CGI application to display the data on the web.

8.2. Baremetal application

The application executed by CPU1 is responsible for executing the power control and for dumping into OCM the control variables selected through the OCM configuration register `SEL_VARS1`.

![Flowchart](image)

*Figure 49: Power control-application flowchart*

In Figure 49, the flowchart of the main routine and the flowchart of the interrupt routine of the application are shown.
On the one hand, first of all the main routine is in charge of initializing the hardware components to be used, in this case, the timer and the interrupt manager. Secondly, the value of the OCM register \texttt{START\_CMD} is monitored. In case \texttt{START\_CMD} is active, the configuration of the OCM records is loaded into the global variables of the program. These variables are used by the interrupt routine for power control. In addition, interrupts of the configured timer are enabled to obtain an execution frequency of 10 kHz. Finally, the OCM \texttt{STOP\_CMD} register is monitored to stop the interrupts in case it is activated.

On the other hand, the interrupt routine is in charge of erase the interrupt flag in order to receive the next one as soon as it is available. Then, the power control is executed and divided into the following sub-processes:

- Update input
- PLL process
- PR process
- Plant process

Each of these processes implement the power control discretized in previous sections in this memory. Once the execution of the power control is finished, the requested data is transferred to the circular buffer in the OCM memory. The selection of data to be downloaded is done through the 32-bit register \texttt{SEL\_VARS1}. This register enables 32 signals that are fixed to an array of pointers that point to the control variables, making this selection at compile time. Once assigned the variables that can be dumped, the \texttt{SEL\_VARS1} register is used to choose which of these 32 variables will be written in the OCM buffer at run time.

8.2.1. Debug methods

In order to debug an application that runs independently, a debug environment is required. The debugging of the CPU1 application has been done using the SDK’s tool.
“System Debugger”. This tool communicates directly with the CPU using a Zedboard’s USB cable.

In Figure 50, there are three information windows of the current state of CPU1.

In the window on the left, the assembly code that is currently executing the CPU1 is shown, in this case, the WFE loop (Wait For Event) at the memory location 0xFFFFFFFF0. This loop is the process that the CPU1 traces after its initialization. CPU1 will remain in that loop until an event occurs and the execution pointer jumps to the power control address.

The internal registers of the CPU are shown in the upper right window, which allows to see how different data is loaded in the execution of the C code, instruction by instruction.
In the lower right window the system is observed, identifying the available CPUs and the status of each of them. In addition, it allows to pause and re-launch any CPU or load a new application into a CPU. This functionality has been used to load and debug the application of power control. It also allows to enable different breakpoints to observe the state of the CPU when passing through a line of C code.
9. Results

The resulting System implements a control power converter in CPU1, leaving CPU0 available for the user interface. This interface, implemented on an HTTP server, performs the configuration of the converter and shows different variables of the power control.

There are two different access modes to the interface corresponding to the HTTP request type: GET and POST.

9.1. GET request

In the GET type requests, the previous configuration parameters are loaded, showing the variables selected in the last POST request. In case no POST request has been made, the interface shown is based on the default configuration. It is shown in Figure 51.

Figure 51: GET request interface
This web page allows to configure the power control through a form divided into three sections:

- **Control Configuration**: This section allows to start or to stop the execution of the power control through the OCM commands `START_CMD` and `STOP_CMD`.
- **Datalog Configuration**: This section allows to select the variables to be dumped through the circular buffer in the OCM. For this, a list of 32 variables that are specified at compile time is used, allowing to choose which of them are selected. These variables can be modified by changing their name in the header file and the pointer to the desired value in the power control application.
- **Plant Configuration**: The last section allows to configure different parameters related to the power control, including different values of discrete components and the amplitudes of the reference voltage and current signals, that is, the current that is injected into the grid and the voltage of it. These last two parameters must be monitored by an ADC according to the real values of the converter and the grid, although for the demonstration of the system they have been selected by software.

The “Launch” button is added, which will be the actuator of the interface to make a POST request to the server with the configured parameters.

### 9.2. POST request

In POST requests, the interface is responsible for processing the request chain to configure the application based on the received attributes. In case the power control has been started through the “Restart CPU1” checkbox, the interface generates a new section designed to configure the signals to be displayed and a graph to represent them. These new sections are shown in Figure 52.
In the Plot Configuration section, the variable or variables to be represented in the lower graph can be configured. This graphic is generated through the Google Charts API [17]. In addition to the variables, the number of points represented and the starting point can be indicated.
10. Conclusions

In this last section, the future lines of the Project will be proposed in the first place, to later expose the technical and personal conclusions.

10.1. Future lines

This project has focused on making the interface for an AMP System, managing a control process in a core and monitoring it through the other core. As it have seen in the section of the introduction "1.3. Target application", the objective is to control an inverter to inject electrical current into a grid coming from an array of photovoltaic panels.

In order to control an actual converter, the input and output signals of the Zedboard must be managed:

- Analog Inputs: The XADC interface of the board has 4 analog inputs to monitor the voltage and current of a balanced three-phase grid, making measurements for two of the phases and estimating the values of the third. To be able to add a larger number of analog inputs, the XADC connector includes 4 digital pins to be used as selection signals of an external analog multiplexer. This would allow working with an unbalanced three-phase grid or monitoring other variables of interest.

- Digital Outputs: The necessary outputs correspond to the activation signals of the different gates of the inverter switching elements. These signals are managed by a Pulse Width Modulator (PWM) that will perform the commutations based on the outputs of the power control. The PWM must be implemented in a hardware module of the FPGA or through the Triple Timer Counters of the Cortex A9. The external interface can be made both by the Pmod connectors and by the FMC connector on the board.
10.2. Technical conclusions

In order to study the performance of the System and verify the design, the interface itself is used to show variables that determine its operation for one second (10,000 samples at 10 kHz).

Figure 53: Control period (µs)

In the first place, the period of execution of the power control is shown in Figure 53. As can be seen, a period of 100 µs on average is achieved with maximum values of 100.05 µs and minimum of 99.95 µs approximately.

Figure 54: IRQ elapsed time (µs)
Secondly, in Figure 54 can be observed the processing time of each of the interrupts in which the power control and the data dump to the interface are performed. For the control shown in the section "6.4. Power control" the result is obtained in less than 2 µs. Due to the margin of time with respect to the 100 µs of the sampling period, it is possible to execute much more complex controls than the one implemented in this project without affecting the performance of the system.

These times are measured by the internal CPU timer, so that the measuring instrument itself can introduce small errors in the measured quantity. Despite the error, these values can be considered adequate for a control system.

The initial specification of obtaining a deterministic system is fulfilled in view of the times obtained for an AMP configuration. In this configuration the CPU of the control has deactivated the caches to the memories and has a code without optimization that reads the value of each variable each time it is used.

Finally, it is necessary to verify the correct operation of the power control. Said verification is based on simulations carried out in Simulink documented in "6.4. Power control". The final simulations of the control variables are then retrieved, shown in Figure 55 and Figure 56.
To compare the results, the same variables are configured in the interface by visualizing the graphs shown in Figure 57 and Figure 58.
From the Figure 57 and Figure 58, the correct functioning of the implemented power control can be determined. The output current obtained follows the selected reference current, in this case, 20 A. In addition, the input voltage to the plant and the output current coincide with their values obtained in Simulink.

10.3. Personal conclusions

This Master Thesis has been developed over two years in which has been reconciled the difficult task of serving the academic world and the professional world that has opened for me in this master’s internship in Televes. Despite the difficulties, I have acquired skills related to the division of tasks and the management of one of the most important resources in the labour market: time.

Carrying out a system that serves as a platform for a future project related to the Electronics Technology Department of the Higher Technical School of Industrial Engineering of the University of Vigo has served as motivation and the result obtained is in accordance with the personal objectives prior to its realization.

The knowledge obtained regarding power control, the ARM architecture and its inclusion with an FPGA are very useful for the professional career, where the implementations of similar SoCs are fundamental.
11. Bibliography


