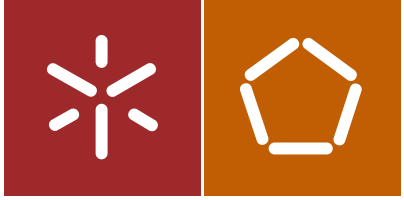
Universidade do Minho
Escola de Engenharia

José Carlos Vale de Mesquita

Hybrid Linux System Modeling with Mixed-Level Simulation

October, 2016

Hybrid Linux System Modeling with Mixed-Level Simulation
José Carlos Vale de Mesquita
UMinho | 2016



Universidade do Minho
Escola de Engenharia

José Carlos Vale de Mesquita

Hybrid Linux System Modeling with Mixed-Level Simulation

Master's Thesis
Master's Degree in Industrial Electronics Engineering and Computers

Carried out under guidance of
PhD Adriano José da Conceição Tavares

October, 2016

Declaração

Nome:
José Carlos Vale de Mesquita

Endereço Eletrónico: carlosvmesquita@gmail.com

Telefone/Telemóvel: 91 226 26 65

Número do Bilhete de Identidade: 14394958

Título da Dissertação: Hybrid Linux System Modeling with Mixed-Level Simulation

Orientador: Professor Doutor Adriano José da Conceição Tavares

Designação do Mestrado:
Mestrado Integrado em Engenharia Eletrónica Industrial e Computadores

Universidade do Minho, ____/_____/_____
Assinatura:_____

Acknowledgments

”I would like to thank my supervisor, PhD Adriano Tavares, for all the insight and solicitude provided throughout the project that helped me reach my objectives for this dissertation. I would also like to thank Engineer Vítor Silva for the time spent sharing his knowledge and leading me throughout the project. Someone I could not forget is Engineer Nelson Naia, who helped me in critical moments during this work, and also for his friendship. Wish him the best of lucks in his new career.

Another great appreciation goes to Professor Jorge Cabral, for giving me the chance to learn from him and his sheer presence. A special thank you goes to Engineers César Monteiro, Marcelo Sousa and Luís Novais, my everyday partners and workspace companions, who helped me through the toughest days.

To my family, who always gave me all i needed to complete this life goal and always walked by my side, a much sincere thank you. To my mother, for putting up with me and never letting go; to my father, for always being there for me when i needed him. To my brother Mário, for being the one who took me into the fascinating world of electronics in the first place!

To you Inês, for being the greatest friend i could ask for. Your patience, caring and courage impress me.

To everyone who contributed to this dissertation, direct or indirectly, and to everyone who has ever caused an impact on my life, my sincerest gratitude.

Lastly, thank you God, for giving me so much reasons to be thankful for.”

Abstract

Keywords: hardware-software co-design, co-simulation, QEMU, PSIM, Linux.

We live in a world where the need for computer-based systems with better performances is growing fast, and part of these systems are embedded systems. This kind of systems are everywhere around us, and we use them everyday even without noticing. Nevertheless, there are issues related to embedded systems in what comes to real-time requirements, because the failure of such systems can be harmful to the user or its environment.

For this reason, a common technique to meet real-time requirements in difficult scenarios is accelerating software applications by using parallelization techniques and dedicated hardware components. This dissertations' goal is to adopt a methodology of hardware-software co-design aided by co-simulation, making the design flow more efficient and reliable. An isolated validation does not guarantee integral system functionality, but the use of an integrated co-simulation environment allows detecting system problems before moving to the physical implementation.

In this dissertation, an integrated co-simulation environment will be developed, using the **Q**uick **EMU**lator (QEMU) as a tool for emulating embedded software platforms in a Linux-based environment. A SystemVerilog **D**irect **P**rogramming **I**nterface (DPI) Library was developed in order to allow SystemVerilog simulators that support DPI to perform co-simulation with QEMU. A library for DLL blocks was also developed in order to allow PSIM® to communicate with QEMU. Together with QEMU, these libraries open up the possibility to co-simulate several parts of a system that includes power electronics and hardware acceleration together with an emulated embedded platform.

In order to validate the functionality of the developed co-simulation environment, a demonstration application scenario was developed following a design flow that takes advantage of the mentioned simulation environment capabilities.

Resumo

Palavras-chave: *HW-SW co-design*, co-simulação, QEMU, PSIM, Linux.

Vivemos num mundo em que a procura por sistemas *computer-based* com desempenhos cada vez melhores domina o mercado. Estamos rodeados por este tipo de sistemas, usando-os todos os dias sem nos apercebermos disso, sendo grande parte deles sistemas embebidos. Ainda assim, existem problemas relacionados com os sistemas embebidos no que toca aos requisitos de tempo-real, porque uma falha destes sistemas pode ser perigosa para o utilizador ou o ambiente que o rodeia.

Devido a isto, uma técnica comum para se conseguir cumprir os requisitos de tempo-real em aplicações críticas é a aceleração de aplicações de *software*, utilizando técnicas de paralelização e o uso de componentes de hardware dedicados. O objetivo desta dissertação é adotar uma metodologia de *co-design* de *hardware-software* apoiada em co-simulação, tornando o *design flow* mais eficiente e fiável. Uma validação isolada não garante a funcionalidade do sistema completo, mas a utilização de um ambiente de co-simulação permite detetar problemas no sistema antes deste ser implementado na plataforma alvo.

Nesta dissertação será desenvolvido um ambiente de co-simulação usando o QEMU como emulador para as plataformas de *software* "embebido" baseadas em Linux. Uma biblioteca para *SystemVerilog DPI* foi desenvolvida, que permite a co-simulação entre o QEMU e simuladores de **R**egister-**T**ransfer **L**evel (RTL) que suportem SystemVerilog. Foi também desenvolvida uma biblioteca para os blocos **D**ynamic **L**ink **L**ibrary (DLL) do PSIM®, de modo a permitir a ligação ao QEMU. Em conjunto, as bibliotecas desenvolvidas permitem a co-simulação de diversas partes do sistema, nomeadamente do hardware de eletrónica de potência e dos aceleradores de *hardware*, juntamente com a plataforma embebida emulada no QEMU.

Para validar as funcionalidades do ambiente de co-simulação desenvolvido, foi explorado um cenário de aplicação que tem por base esse mesmo ambiente.

Contents

List of Figures

List of Tables

Acronyms List

ACC **ACC**ess

ADC **A**nalog to **D**igital **C**onverter

AMBA **A**dvanced-**M**icrocontroller **B**us **A**rchitecture

API **A**pplication **P**rogram **I**nterface

ARM **A**dvanced **R**ISC **M**achine

ASIC **A**pplication-**S**pecific **I**ntegrated **C**ircuit

ASIP **A**pplication-**S**pecific **I**nstruction Set **P**rocessor

AVR32 **A**dvanced **V**irtual **R**ISC 32

AXI **A**dvanced e**X**tensible **I**nterface

BDM **B**ackground **D**ebug **M**ode

BSD **B**erkeley **S**oftware **D**istribution

CLB **C**onfigurable **L**ogic **B**lock

CPU **C**entral **P**roccessing **U**nit

COTS **C**ommercial **O**ff-**T**he-**S**helf

DDR **D**ouble **D**ata **R**ate

DMA **D**irect **M**emory **A**ccess

DPI **D**irect **P**rogramming **I**nterface

DSP **D**igital **S**ignal **P**rocessor

DLL **D**ynamic **L**ink **L**ibrary

DUT **D**evice **U**nder **T**est

eCos **e**mbedded **C**onfigurable **o**perating **S**ystem

EDA **E**lectronic **D**esign **A**utomation

ESRG **E**mbedded **S**ystems **R**esearch **G**roup

EEPROM **E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory

FIFO **F**irst **I**n **F**irst **O**ut

FMI **F**unctional **M**ock-up **I**nterface

FMIL **F**unctional **M**ock-up **I**nterface **L**ibrary

FMU **F**unctional **M**ock-up **U**nit

FPGA **F**ield **P**rogrammable **G**ate **A**rray

FSF **F**ree **S**oftware **F**oundation

GPU **G**raphics **P**roccessing **U**nit

GCC **G**NU **C**ompiler **C**ollection

GPL **G**eneral **P**ublic **L**icense

GPP **G**eneral **P**urpose **P**rocessor

GUI **G**raphical **U**ser **I**nterface

HDL **H**ardware **D**escription **L**anguage

HLS **H**igh **L**evel **S**ynthesis

HVL **H**ardware **V**erification **L**anguage

HDVL **H**ardware **D**escription and **V**erification **L**anguage

HTML **H**yper**T**ext **M**arkup **L**anguage

IBM **I**nternational **B**usiness **M**achines

IDE **I**ntegrated **D**evelopment **E**nvironment

IEEE **I**nstitute of **E**lectrical and **E**lectronic **E**ngineers

ISA **I**nstruction **S**et **A**rchitecture

I/O **I**nput/**O**utput

IP **I**ntellectual **P**roperty

I2C **I**nter-**I**ntegrated **C**ircuit

IRQ **I**nterrupt **R**e**Q**uest

ITEA2 **I**nformation **T**echnology for **E**uropean **A**dvancement 2

JTAG **J**oint **T**est **A**ction **G**roup

LAB **L**ogic **A**rray **B**lock

LC **L**ogic **C**ell

LE **L**ogic **E**lements

LRM **L**anguage **R**eference **M**anual

LUT **L**ook**U**p **T**able

MIPS **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages

MPI **M**essage **P**assing **I**nterface

OEM **O**riginal **E**quipment **M**anufacturer

OS **O**perating **S**ystem

PCIe **P**eripheral **C**omponent **I**nterconnect **e**xpress

PLB **P**rocessor **L**ocal **B**us

PLI **P**rogramming **L**anguage **I**nterface

POSIX **P**ortable **O**perating **S**ystem **I**nterface

QEMU **Q**uick **EMU**lator

RAM **R**andom **A**ccess **M**emory

ROM **R**ead-**O**nly **M**emory

RMS **R**oot **M**ean **S**quared

RC **R**elease **C**andidate

RISC **R**educed **I**nstruction **S**et **C**omputer

RPU **R**eal-time **P**roccessing **U**nit

RTL **R**egister-**T**ransfer **L**evel

RTC **R**eal-**T**ime **C**lock

RTOS **R**eal-**T**ime **O**perating **S**ystem

SDK **S**oftware **D**evelopment **K**it

SPI **S**erial **P**eripheral **I**nterface

SRAM **S**tatic **R**andom-**A**ccess **M**emory

SoC **S**ystem-**o**n-**C**hip

THD **T**otal **H**armonic **D**istortion

TCP/IP **T**ransmission **C**ontrol **P**rotocol/**I**nternet **P**rotocol

TF **T**ask/**F**unction

UART **U**niversal **A**synchronous **R**eceiver **T**ransmitter

USART **U**niversal **S**ynchronous **A**synchronous **R**eceiver **T**ransmitter

UML **U**nified **M**odeling **L**anguage

USB **U**niversal **S**erial **B**us

UUT **U**nit **U**nder **T**est

VPI **V**erilog **P**rocedural **I**nterface

XML e**X**tensible **M**arkup **L**anguague

Chapter 1

Introduction

This chapter, being the first of the dissertation, introduces the context of the developed work along with defining its motivations and main objectives. The structure of this document is also presented at the final section of this chapter.

1.1 Contextualization

Most everyday and worldwide used devices are controlled by an embedded software, ranging from a simple washing machine to an automobile or a train. With the rapid growth and intrusion of technology in human life, there is also a growing need for better embedded system solutions. The development of those softwares is particularly more demanding than other areas of software engineering due to the functional and non-functional requirements in terms of quality, reliability and performance.

Embedded systems are computational systems designed for a specific purpose, and are usually a smaller part of a larger system, performing lower-level tasks and acting as bridges to the physical world. Embedded systems are everywhere, and can be thought as every digital system that is not a desktop computational system.

The development process for embedded systems is usually supported by **C**ommercial **O**ff-**T**he-**S**helf (COTS) embedded development boards, which are usually shipped with a toolchain that comprises tools like compilers, linkers and debuggers. The software that will run on the target platform is usually developed and debugged in the host platforms, and only later deployed.

The development process when it comes to hard real-time embedded systems is even more demanding, since they demand deterministic behavior. Hard real-time embedded systems are systems with strict timing constraints that must be met. Some systems, such as the control process for a nuclear power plant, have such real-time requirements that when not met may fail utterly and completely, causing catastrophic damage.

Therefore, it is clear that embedded software development for real-time systems requires different design flows and a broader set of skills from the developer. Modern techniques to enhance real-time requirements satisfaction include computational offloading of software tasks to dedicated hardware co-processors. With the fast evolution of the **F**ield **P**rogrammable **G**ate **A**rray (FPGA) technologies, the use of hardware-software co-design flows for the development of real-time embedded systems is an increasingly used solution. When using these design flows, the system can be sped up due to the inherent parallel nature of hardware present in custom hardware co-processors.

However the development of such systems can be very time consuming due to the complexity and multi-technology integration, their implementation involves the development of software modules, hardware modules and interfaces to connect them. Software application acceleration is often an optimization compromise in a cost-benefit relationship during the migration of software processes to hardware dedicated co-processors.

In order to characterize the whole system, an integrated co-simulation environment where all the metrics are contemplated would be ideal. Such an environment would not only be useful in order to measure these metrics, but also in the system modeling process. As an example, a power electronics simulation tool such as PSIM® may be used in order to simulate the behavior of the systems' hardware. Figure 1.1 presents an overview of a co-simulation environment.

Hardware-software co-design is usually a long and iterative process, and the development time can grow exponentially if the different domains of the system are modeled and simulated separately, and only then implemented and tested. An isolated validation for the different application domains does not guarantee integral system functionality, while with an integrated co-simulation environment, system problems can be detected earlier, before moving to the physical implementation phase.

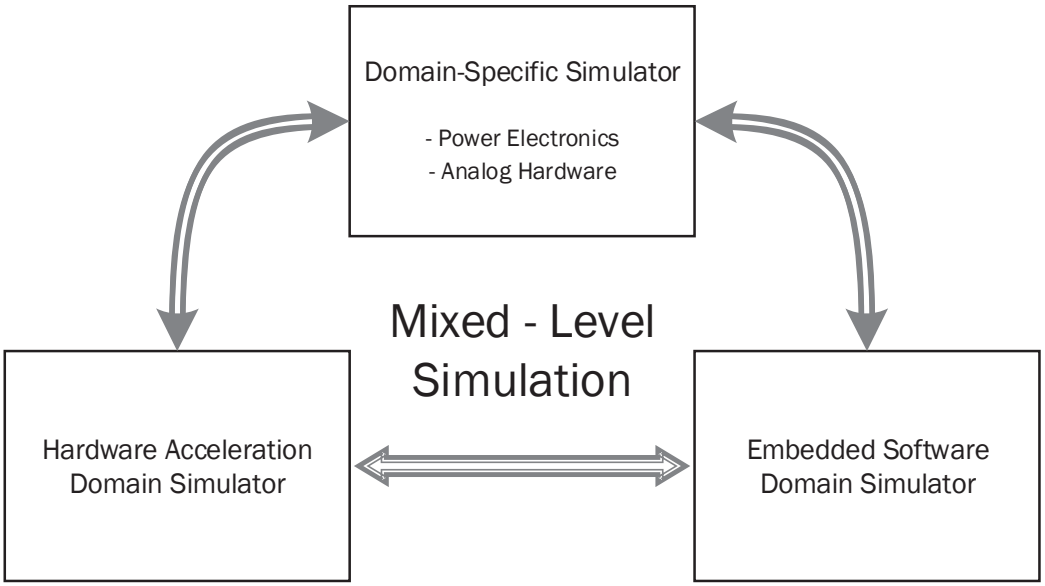


Figure 1.1: Co-Simulation Environment overview

By using a design flow based on co-simulation, the development process becomes more efficient/reliable since the system can be modeled first, profiled and analyzed later according to certain metrics, and only then is the target platform selected, thus reducing the number of iterations needed in order to achieve the final system design.

1.2 Motivation and Objectives

Since power electronics is a area of great interest to the author of this dissertation, and those systems usually require fast-responding, deterministic controllers with real-time constraints, it is a good area to apply the above-mentioned co-design flow for hardware-software systems. Developing such a system, integrating multiple domains like power electronics, hardware acceleration and an embedded platform usually translates into a long iterative process when using conventional design flows. Also, when simulating such a system, local caches must be used to stimulate and validate each application domain, implying lengthy development cycles in most cases.

These demands can be met by using an environment that allows simulating domain crossing interactions. Currently, there are no tools available on the market to

perform this, i.e., supporting a design flow based on co-simulation. Hence, one of the main objective of this project is to develop a simulation environment that allows for cross-domain validation.

Co-simulation has being tackled at the **E**mbedded **S**ystems **R**esearch **G**roup (ESRG) of *Universidade do Minho* through previous dissertations. A co-simulation framework for hardware accelerated embedded systems has been implemented, as it could greatly improve future project development processes. This dissertation follows that vision, being framed as the continuation of the work developed by Naia (2015) towards creating such a framework.

1.3 Dissertation Structure

This document presents the development of libraries and simulation tools extensions that allow creating such an integrated co-simulation environment for real-time hardware accelerated applications. Its contents are divided in six chapters, with the current chapter being the first, and the rest being briefly presented next.

The second chapter presents some knowledge background on the project: topics on embedded systems and real-time systems, along with operating systems are present. Hardware acceleration and the hardware-software co-design design flow are also discussed.

After that, in the third chapter, the most common tools and methodologies used in hardware accelerated application development for embedded systems are presented, along with the most commonly followed co-simulation models. The QEMU basics are also presented, along with a brief mention to the simulation extensions developed in the past that will support the development of this dissertation. Next, the Verilog and SystemVerilog simulation interfaces are approached, with a small mention to the Verilogs **P**rogramming **L**anguage **I**nterface (PLI) and a more extensive analysis to the SystemVerilogs DPI. The PSIM® basics are also presented along with its DLL blocks interface. Finally, the **F**unctional **M**ock-up **I**nterface (FMI) co-simulation extension standards are mentioned, bringing up their relevance in the context of this dissertation. The chapter ends with a comparison between the previous work, developed by Naia (2015), and the current work, with the main changes being addressed.

In the fourth chapter, the developed work, having in view the creation of a co-

simulation environment, is approached by presenting the developed extensions and libraries. The content is presented in a functional manner, providing insights on how development was made, as well as on how to make use of these mechanisms. The mechanisms used in the implemented work are presented and examples are given whenever possible, to help the reader understanding how to use the provided interfaces developer perspective.

The fifth chapter presents an application scenario, providing a practical example on how the co-simulation environment features can be used and also proving its' functionality. The chosen case of study targeted power electronics domain, namely a shunt active power filter, with simulated power hardware components along with the controller for the system.

The sixth and last chapter of this document discusses the obtained results and conclusions concerning the developed work. Some interesting topics regarding future work are also mentioned, clarifying what can be further done in terms of functionalities and further improve it to create a full hardware accelerated embedded system validation framework, one that minimizes the design flow iterations.

Chapter 2

Basic Knowledge Background

This chapter presents some basic background on the technologies and methodologies used in this dissertation. Given that this work is mainly focused on co-simulating Embedded Systems, the first topics are about them, with greater emphasis on Real Time Embedded Systems, that sometimes require a supporting **O**perating **S**ystems (OSs) due to their complexity. Given that, the main Linux features are presented, as it was the chosen OSs to back up this dissertation. Hardware-software co-design will be addressed, and since it is this work's main subject and involves programmable hardware acceleration, FPGAs technologies will then be discussed, along with its' advantages.

2.1 Embedded Systems

Nowadays, embedded systems are all around us, and they shape the way we live, even though we don't realize it. From consumer electronics (toys, cameras, microwaves) to industrial automation (robotics), automotive components (antilock brakes), military defense systems, energy generation, medical equipment (cardiac monitors) and networking components (routers, hubs, gateways), embedded systems create both huge value and unprecedented risks.

In 2008, there was an average of 30 embedded microprocessors per person, in developed countries. Embedded microprocessors account for more than 98% of all produced microprocessors, rather than personal computer purposes. As more devices become automated and consumers acquire more such devices, the volume

of embedded systems is increasing at 10 to 20 percent per year depending on the domain (Ebert and Jones, 2009).

These numbers may very well have even increased with the smartphone and tablet business boom of recent years, and subsequent rise of popularity of the ARM architecture, resulting in ever greater demand in the embedded market (Naia, 2015).

2.1.1 Definition

An embedded system is an applied computer system that distinguishes from other types of computer systems such as personal computers or supercomputers because they are specifically designed to perform certain tasks. They are often subsystems of bigger systems, and the user doesn't even notice their presence. However, the definition of "embedded system" is fluid and difficult to pin down, as it constantly evolves with advances in technology and dramatic decreases in the cost of implementing various hardware and software components (Noergaard, 2013). Still, some common characteristics can be identified:

- Presence of a processing unit, like a microprocessor or microcontroller, **S**ystem-**o**n-**C**hip (SoC), etc.;
- Usually designed to only contain the strictly necessary hardware to perform the task or tasks assigned to them;
- Might have restrictions as regards to power, being powered by batteries or renewable energy sources, like solar or eolic (Silva, 2011).

Current embedded systems can be split into three groups as low-end, middle-end and high-end embedded systems, regarding their complexity. Their processors can either be 4/8-bit simple processors or multi-core 32/64 bit processing units. Their softwares can go from very basic bare-metal applications to complex OSs running applications in parallel.

Some examples of embedded systems are:

- Low-end:

  Soldering iron, electric heaters, digital watches, vending machines, washing machines, guitar tuners.

- Middle-end:

  Washing or cooking systems, routers and hubs, printers, scanners and faxes, digital cameras, TVs.

- High-end:

  Biomedical systems, cars and trains, airplanes and space shuttles, smart-TVs, tablets and smartphones.

2.1.2 Hardware

Processors are the main functional units of an embedded system, as these latter must contain at least one master processor acting as the central controlling device. Still, this does not mean that this processing unit must me a programmable processor. There are several types of processors, and unlike desktop computers, a processor with lots of instructions and capabilities does not mean that it is best suited for an embedded system. Figure 2.1 presents a performance/power efficiency vs. flexibility overview between the different kinds of processors.

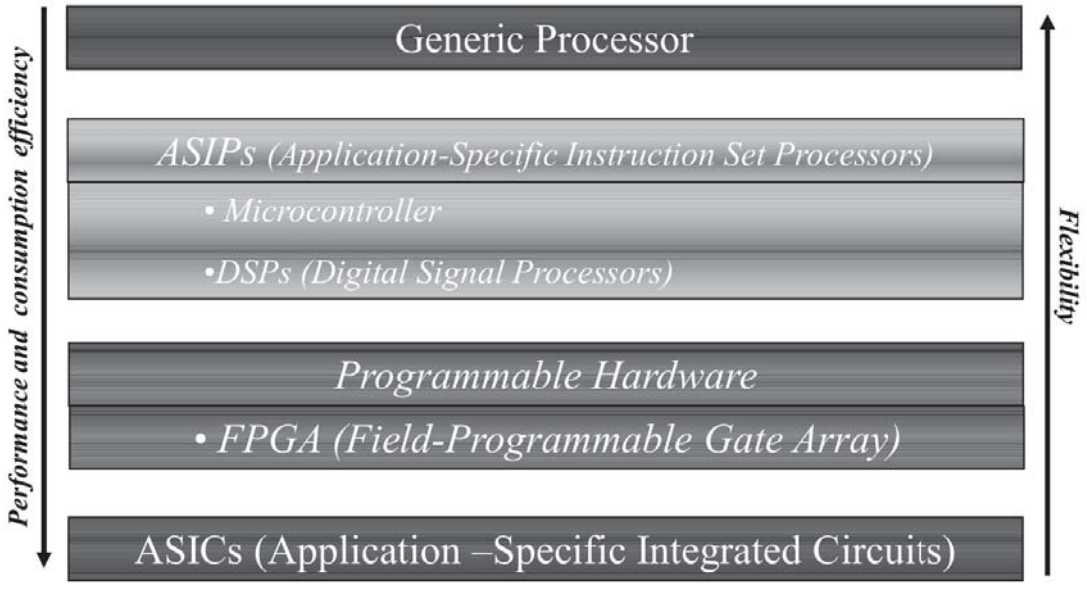


Figure 2.1: Processor types (Naia, 2015)

Starting from the top of figure 2.1, we have **G**eneral **P**urpose **P**rocessors (GPPs), also know as microprocessors, the most flexible of all processors. The **C**entral **P**roccessing **U**nits (CPUs) used in our personal computers belong to this family of processing units, but even though they have high performances, mainly due to the use of parallelism. They are the least efficient as far as energy consumption/performance ratio is concerned. Nevertheless, the flexibility they bring to personal computers is useful for that use, but most of the times useless for embedded systems. Given that, along with their unpredictable execution times due to resource sharing and dynamic decisions, makes these processors usually not suited for embedded systems applications.

On the opposite side, we have **A**pplication-**S**pecific **I**ntegrated **C**ircuits (ASICs), and they somehow are the opposite of GPPs, because they are the least flexible of all processing units, being built to perform a specific task. These chips are only used when maximum efficiency and performance is necessary and a large number of sales are assured, because they have very high time-to-market and development costs, sometimes reaching six digit values.

Between these two processor "families", we have FPGAs, which are a low-cost solution to the above-mentioned ASICs, although they need more area and consume more power than their ASIC counterparts. The bigger advantage of these processors is that they are reprogrammable, being more flexible than ASICs. Studies from 2008 state that "designs implemented on FPGAs need on average 40 times as much area, draw 12 times as much dynamic power, and run at one third the speed of corresponding ASIC implementations", but "most recent FPGAs provide significantly reduced power, increased speed, lower materials cost, minimal implementation real-estate, and increased possibilities for re-configuration 'on-the-fly'" (Hauck and DeHon, 2008). They are frequently used as a low-cost alternative to prototype and debug ASICs designs during the development process, as the manufacturing costs of creating a new ASIC mask design are what make the technology expensive.

Lastly, **A**pplication-**S**pecific **I**nstruction Set **P**rocessor (ASIP) are defined as "software programmable processing elements tailored for particular applications" (Sato et al., 1991), which means they are somewhat flexible processors, but not as generic as microprocessors, being a good compromise between single-purpose processors and microprocessors. These processors are particularly optimized for a particular class of applications, namely graphics, video, networking and signal processing

(Gries and Keutzer, 2005), and often have hardwired components additional to the actual processing unit, such as memory blocks or other peripherals. For these reasons, they are often the ideal choice for embedded systems, and comprise the most frequent choice in embedded system design. Some examples of ASIPs are microcontrollers, **D**igital **S**ignal **P**rocessors (DSPs) and SoC, which will be now discussed.

Microcontrollers are chips that integrate microprocessors, usually 8 to 16 bits or 32 bits, and memory blocks in the same package. They are generally aimed towards low-end control-oriented embedded systems with common characteristics, being packed with little memory and sometimes even additional peripherals, to provide a complete single chip control solution. They are usually "reactive", which means their behavior is driven by events. These chips usually have low power consumptions.

DSPs are a particular set of ASIPs which are optimized for data stream oriented applications, with high data throughput, often including dedicated instructions and peripherals that are suited for digital signal processing, such as floating-point units and specialized memory.

An **SoCs** integrates all components of a computer system in a single chip, and is best suited for middle-end/high-end embedded purposes, being one of the most popular choices in embedded system design nowadays. SoCs may be comprised of one or more microprocessors, usually 32 or 64 bits, DSPs, microcontrollers, memory blocks (**R**ead-**O**nly **M**emory (ROM), **R**andom **A**ccess **M**emory (RAM), **E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory (EEPROM) and Flash), peripherals (like timers and **R**eal-**T**ime **C**locks (RTCs)) and external interfaces, such as **U**niversal **S**erial **B**us (USB), **S**erial **P**eripheral **I**nterface (SPI), **I**nter-**I**ntegrated **C**ircuit (I2C), **U**niversal **S**ynchronous **A**synchronous **R**eceiver **T**ransmitter (USART) or Ethernet, also found in some microcontrollers. Usually, the various hardware blocks present in SoCs are connected via proprietary or industrial standard buses such as **A**dvanced **R**ISC **M**achines (ARMs)® **A**dvanced-**M**icrocontroller **B**us **A**rchitecture (AMBA)® or Altera® Avalon®, and **D**irect **M**emory **A**ccess (DMA) controllers route data directly between external interfaces and memories, which increases data throughput in SoCs, as mentioned above.

In figure 2.2 we have a real example of an SoC chip (Intel® Atom™ x5/x7) used in mobile applications (smartphones and tablets), where a lot of the above-mentioned technologies are present, amongst some others.

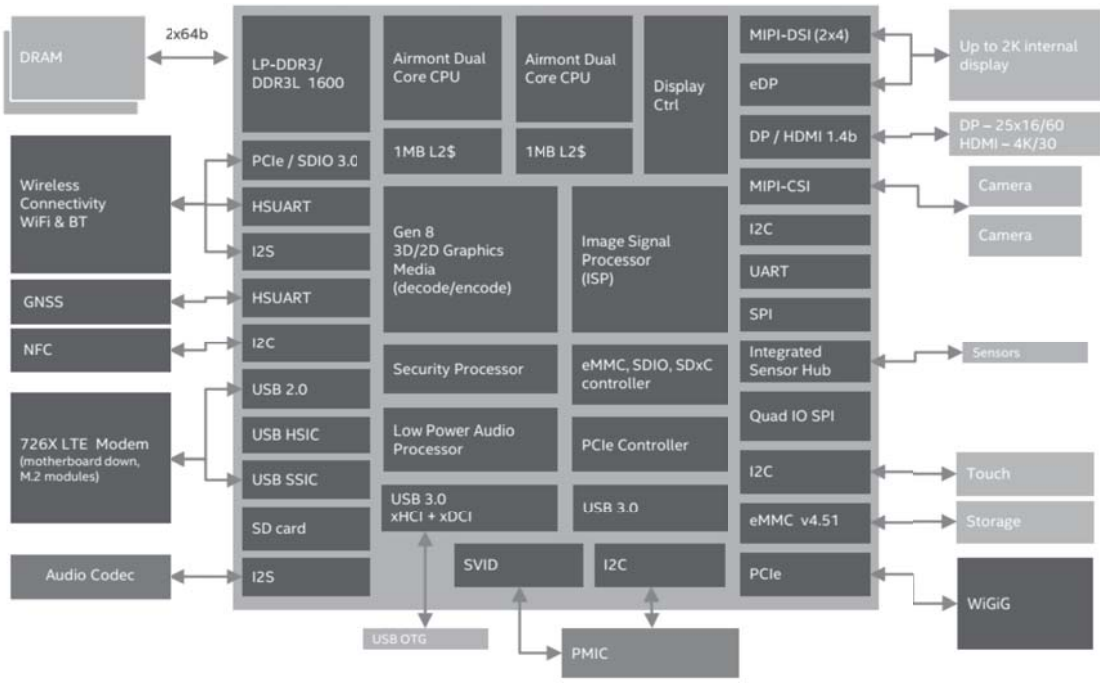


Figure 2.2: Intel® Atom™ x5 and x7 Processor Platform Block Diagram (Tu, 2015)

2.1.3 Development

Several setups can be used when it comes to embedded systems development, but only three will be briefly discussed: linked setup, removable storage setup, and standalone setup.

- Linked Setup

  This setup presupposes that the target and the host are permanently linked together using a physical link, usually serial or Ethernet. Given this, no physical hardware storage devices are needed, and all data transfers are made via the physical connection. The host contains the cross-platform development environment, and the target must contain an appropriate bootloader, functional kernel and a minimal root filesystem. Figure 2.3 illustrates such setup.

  This is the most common setup, and the physical link can also be used for debugging purposes. However, the most common approach is to have a different link for debugging (Yaghmour et al., 2008).

- Removable Storage Setup

  In this setup, there are no direct physical links between the host and the

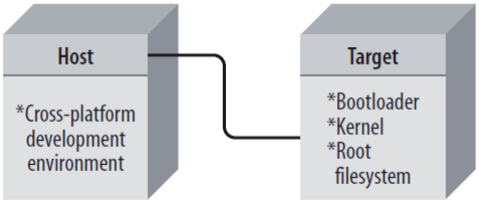


Figure 2.3: Host/Target linked setup (Yaghmour et al., 2008)

target. A storage device is used instead, which is written by the host and then used to boot the target device. As seen in the previous setup, the host contains the cross-platform development environment, but this time the target only contains a minimal bootloader, as the storage device contains the rest of the components. This setups are alike the one in 2.4.

Using a setup like this, the target may not contain any persistent storage at all, and a flash storage device is used in this case. This setup is mostly used in the early stages of embedded system development (Yaghmour et al., 2008).

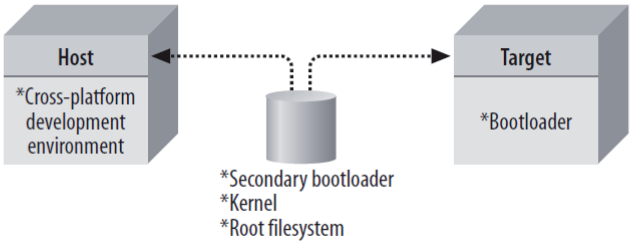


Figure 2.4: Host/Target removable storage setup(Yaghmour et al., 2008)

- Standalone Setup

  In this case, the target includes all the required software to boot, operate and develop additional software. This setup is similar to an workstation, except for the hardware, and does not require any cross-platform development environment. Figure 2.5 represents this type of setups.

  These setups are most popular between developers building high-end PC-based embedded-systems, since they can use standard off-the shelf Linux distros on the system. When the development is done, they then trim down the distribution, customizing it for their specific needs (Yaghmour et al., 2008).

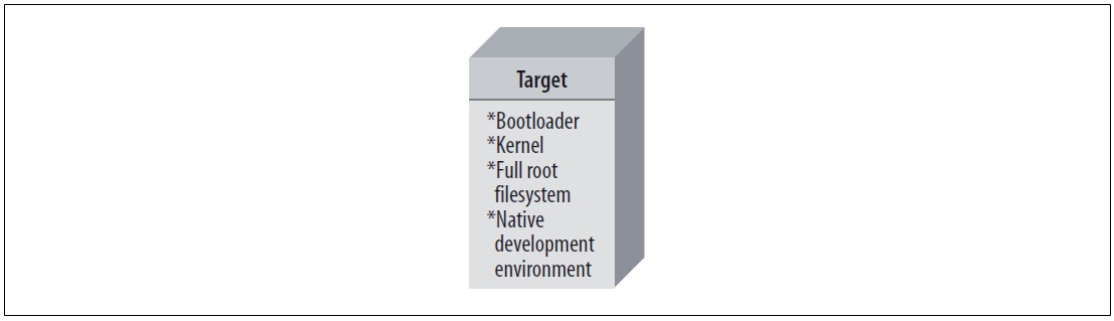


Figure 2.5: Host/Target standalone setup(Yaghmour et al., 2008)

2.1.4 Startup - Bootloaders

In the previous section, the main hardware components of a complex embedded system were discussed, along with their development setups, but a description of an embedded system wouldn't be complete without discussing its' startup. A bootloader is the first software to run upon startup, being highly dependent on the target's hardware, performing low-level hardware initialization and then jumping to the kernel's startup code. In more complex systems, the bootloader is also responsible for loading the operating system (Yaghmour et al., 2008).

In the latter case, the steps executed by the bootloader are the following:

- Initialize the critical hardware components, essential to the systems' startup, such as RAM controllers, **I**nput/**O**utput (I/O) controllers and graphic controllers;
- Initialize the systems' memory, in the proper format to transfer the control to the OS;
- Allocate resources such as memory and interrupts, associated to hardware peripherals;
- Initialize hardware and software controllers that allow loading the image of the OS to the systems' memory;
- Loading the OSs' image to memory, decompress it and transfer the processors' control to the newly created OS (Silva, 2011).

There is a big variety of embedded boards available for Linux in the market, along with as many bootloaders. Also, there are many possible boot configurations for a single board! Due to this, choosing the bootloader is up to the embedded developer,

taking into the account the architecture of the target system under development. Some architectures have well-known, established bootloaders providing support for a range of hardware, while others have few or no standard bootloaders and mainly use those provided by the hardware manufacturer (with highly varying quality).

Most typical desktop or server Linux systems use a bootloader such as LILO or GRUB, whose job it is to program various core system components and to provide various information for use by the OS. Because many embedded systems don't come with prewritten firmware, the implementations available must determine precisely what hardware is installed within the system and supply this to the Linux kernel. An embedded project's first task is likely to be porting U-Boot, or a similar bootloader, to the board (Yaghmour et al., 2008).

There are some open source bootloaders that can be used on embedded applications:

- **U-Boot**, the Universal Bootloader by Denx is undoubtedly the most used bootloader in embedded systems, being the most flexible and active in the open source comunity. U-Boot supports development platforms based on the following architectures: ARM, **A**dvanced **V**irtual **R**ISC 32 (AVR32), Blackfin, MicroBlaze, NIOS, PowerPC, MIPS, X86 and m68K, amongst others (DENX Software Engineering, 2016).

- **Barebox** derives from U-Boot, but is not as flexible and supported as the Universal Bootloader. It runs on a variety of architectures including x86, ARM, **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages (MIPS), PowerPC and others (Barebox.org, 2016).

- **RedBoot** is an bootloader based on **e**mbedded **C**onfigurable **o**perating **S**ystem (eCos), written by Cygnus Solutions, and then bought by Red Hat, Inc. eCos is very popular in very small embedded systems, that won't run Linux, but RedBoot has been extended to boot other OSs, including Linux (Red Hat, Inc., 2016).

Bootloaders are compiled in cross-compiling environments, and loaded into the non-volatile memory of the system usually using hardware interfaces such as **J**oint **T**est **A**ction **G**roup (JTAG) or **B**ackground **D**ebug **M**ode (BDM). The memory region of the bootloader usually has mechanisms to prevent data corruption, that could prevent the system from booting in the future.

2.1.5 Real-Time Embedded Systems

"The basic idea behind real-time embedded systems is that we expect them to respond to its environment in time. Real-time does not means real fast, it simply means fast enough in the context in which the system is operating. The essence of real-time computing is not only that the computer responds to its environment fast enough, but that it responds **reliably fast enough**" (Abbott, 2003).

"Real-time systems span a wide range of domains including industrial plants control, automotive, flight control systems, monitoring systems, multimedia systems, virtual reality, interactive games, consumer electronics, industrial automation, robotics, space missions and telecommunications. In these systems, a late action might cause a wrong behavior (e.g., system instability) that could even lead to a critical system failure. Hence, the main difference between a real-time task and a non-real-time task is that a real-time task must complete within a given deadline, which is the maximum time allowed for a computational process to finish its execution" (Lee et al., 2007).

Real-time embedded systems can be classified into two categories, regarding the consequences caused by a missed deadline, as soft or hard real-time systems.

- **Hard real-time** embedded systems are those where missing a deadline may have catastrophic consequences on the controlled system. Examples of hard real-time systems are nuclear reactor control systems or anti-lock brakes on a vehicle. In both these cases, if the system does not meet its required deadlines, total system failure occurs with very harmful consequences.

- An embedded system is said to be **soft** if missing a deadline causes a performance degradation but does not jeopardize the correct system behavior. Typical soft real-time systems include user command interpretation, keyboard input, message visualization, system status representation and graphical activities. (Lee et al., 2007)

"The most important property of a real-time system is not performance, or high speed, but its' predictability. In a predictable system, we should be able to determine in advance whether all the computational activities can be completed within their timing constraints. The deterministic behavior of a system typically depends on several factors ranging from the hardware architecture to the operating system up to the programming language used to write the application. Architectural

features that have major influence on task execution include interrupts, direct memory access (DMA), pipelining, cache, and prefetching mechanisms. Although such features improve the average performance of the processor, they introduce a non-deterministic behavior in process execution, prolonging the worst-case response times" (Lee et al., 2007).

2.2 Operating Systems

Operating systems are "the chief pieces of software" (McHoes and Flynn, 2013), who manage machine resources, controlling who can use the system, and how. They are basically programs who perform the interface between applications and the systems hardware (Stallings, 2014). As embedded technology grew to meet ever growing system demands, the adoption of OS for system development became very common, being an integral part for several embedded systems.

2.2.1 Concurrency & Scheduling

Concurrency means that several tasks must be carried out simultaneously, but single-core microprocessors are capable of executing only one instruction at a time, and if an external event triggers a processing task, the processor would be unavailable to attend that request.

Even though, there are concurrent programming techniques to overcome this limitation, where the illusion of concurrent execution is achieved by interleaving the execution steps of each task via time-sharing slices, where only one task runs at a time, and if it does not complete during its time slice, it is paused. Then, another task begins or resumes, and then later the original task is resumed. In this way, multiple task are part-way through execution at a single instant, but only one is being executed at each instant. Figure 2.6 presents an example diagram of two tasks being executed concurrently.

As above-mentioned, the two tasks are executed by the processor once at a time, with CPU execution jumping from one task to the other. Task switching can be implemented using timer interrupts, with scheduling code being ran in the interrupt to decide which task code should the interrupt return to. If the task switches are frequent enough, it may seem as the system is running the two tasks in par-

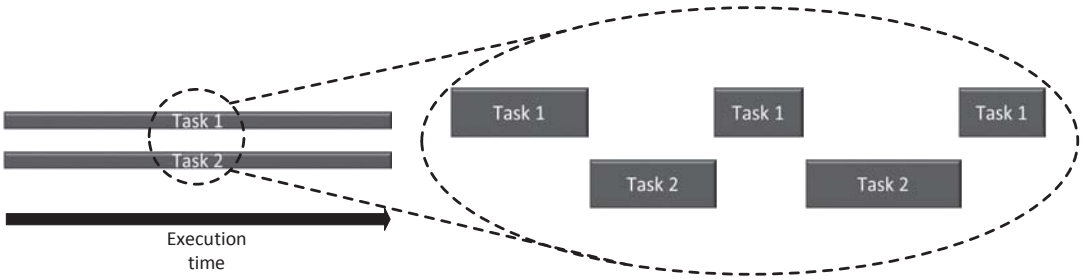


Figure 2.6: Virtual parallelization of two tasks (Naia, 2015)

allel. This is the basis of the multi-tasking paradigm used in most computational systems nowadays, and it is one of the most important aspects of an operating system (Naia, 2015).

At its absolute most basic form, an OS can be a software program that allows for task code implementation, running a scheduler that decides which task should be in execution at a given instant. There are many scheduling algorithms and techniques, but nowadays, the most well known OS, such as Windows, Mac-OS and Linux use preemptive schedulers.

2.2.2 Kernel & User Spaces

An seen in the last section, OSs can be defined as software programs that allow for concurrent task execution. However, most OSs are much more complex than that, taking care of software and hardware resource management and providing a set of services that are common to most applications. In modern OS, the kernel is the core component of the OS and can be defined as the resource manager, being responsible for task scheduling and hardware access management. Most modern large OSs implement at least two execution spaces: kernel space and user space. The OS is often called the system kernel, or simply the kernel, to emphasize its isolation from the user and applications (Stallings, 2014). Figure 2.7 provides a general description of the classic UNIX architecture.

Kernel space runs code that accesses hardware like memory and I/O, providing an interface through system calls to upper layers. User space runs OS services that do not require direct hardware access, like network services and user interface services such as **G**raphical **U**ser **I**nterfaces (GUIs) and command line shells. User

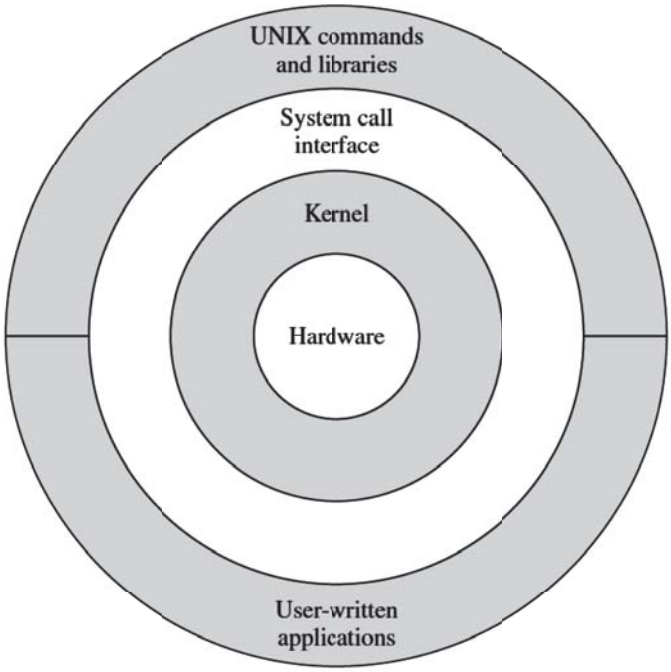


Figure 2.7: UNIX OS architecture (Stallings, 2014)

applications are also ran in user space.

There are two main type of kernels: microkernels and monolithic kernels. Microkernels contain the minimum amount of services running in kernel space, with most services being ran in user space, while monolithic kernels run most of their services in kernel space. Regardless of being implemented in user space or kernel space, most OSs provide a set of basic services, using a set of concepts that make possible the modern computing systems that we see today (Naia, 2015).

A closer look at an example of a UNIX-based monolithic kernel is provided in Figure 2.8:

- User programs can invoke OS services either directly or through library programs.
- The system call interface is the boundary with the user and allows higher-level software to gain access to specific kernel functions (Stallings, 2014).

At the other end, the OS contains primitive routines that interact directly with the hardware. Between these two interfaces, the system is divided into two main parts, one concerned with process control and the other concerned with file management and I/O:

- The process control subsystem is responsible for memory management, the scheduling and dispatching of processes and the synchronization and inter-process communication of processes. The scheduling is perhaps the most important feature of an OS as was previously covered, because it allows for multi-tasking, which is essential to the system responsiveness and concurrent application execution.

- Virtual memory is a technique that greatly improves system performance, usually tightly coupled with the concept of processes, although it is also used in single address space OS. With this technique a process assumes that the whole system's memory is free, liberating processes from the responsibility of managing memory addresses. Processes' memory addresses are interpreted by the OS as virtual addresses and then translated to actual physical addresses.

- The file subsystem is an abstraction that is provided to manipulate data in storage devices. It provides a way of organizing data, usually in a hierarchy of directories or folders arranged in a directory tree. Each group of data is called a file, and follows a given structure and logic rules, so data can be recognized and organized.

- This system exchanges data between memory and external devices either as a stream of characters or in blocks. To achieve this, a variety of device drivers are used. Device drivers are software entities that contain implementation of software that is related to the specificness of the underlying hardware. Monolithic kernels usually implement device drivers in kernel space to prevent user applications from crashing the whole system with ease. For block-oriented transfers, a disk cache approach is used: a system buffer in main memory is interposed between the user address space and the external device (Stallings, 2014).

2.2.3 Linux

Linux started out as a UNIX variant for the **I**nternational **B**usiness **M**achines (IBM) PC architecture "Intel i386". Linus Torvalds, a Finnish student of computer science wrote the initial version of its monolithic kernel as a hobby, and posted it online in October 1991. Since then, a number of people, collaborating over the Internet, have contributed to the development of Linux, all under the control of

Torvalds. It has evolved rapidly, being widely adapted to most modern systems and becoming one of the most successful OSs, on par with some famous commercial others such as Windows and MacOS. One of the keys to the success of Linux has been the availability of free software packages under the auspices of the **F**ree **S**oftware **F**oundation (FSF). Its goal is to achieve stable, platform-independent software that is free, high quality, and embraced by the user community (Stallings, 2014).

The Linux kernel is licensed under FSF's **G**eneral **P**ublic **L**icense (GPL) which

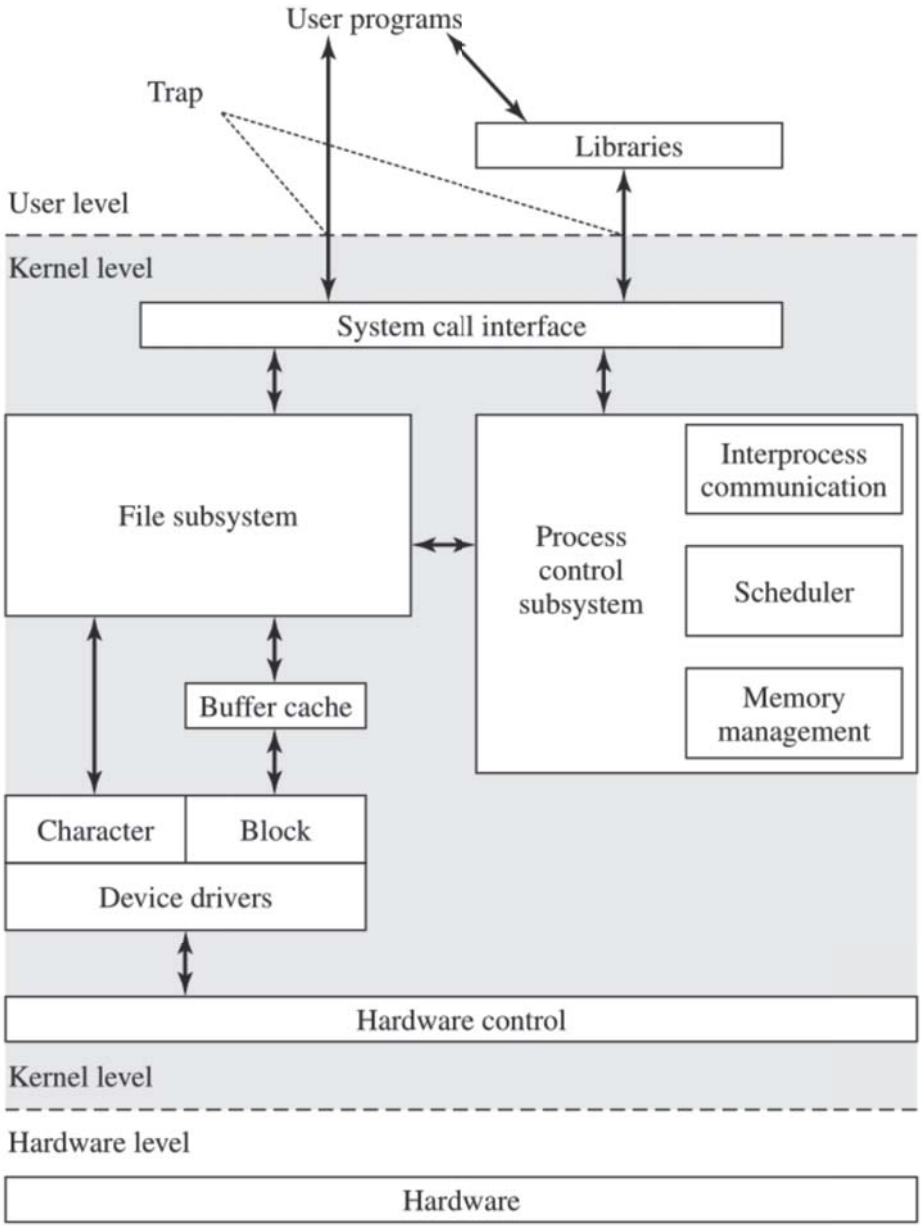


Figure 2.8: Traditional UNIX Kernel (Stallings, 2014)

is a license that grants free access to the original software code, as well as the possibility to change it and redistribute at will. In addition to its use by many individual programmers, Linux has now made significant penetration into the corporate world. This is not only because of the free software but also because of the quality of the Linux kernel. Many talented programmers have contributed to the current version, resulting in a technically impressive product. Moreover, Linux is highly modular and easily configured. This makes it easy to squeeze optimal performance from a variety of hardware platforms (Stallings, 2014).

Several OS make use of the Linux kernel, being referred to as Linux distributions. Although there are not that many desktop Linux users compared to other popular OSs, it is the preferred OS of choice in supercomputers, servers and most embedded devices, such as MP3 players, DVD players and HD TVs.

There are a wide range of motivations for choosing Linux over a traditional embedded OS:

- **Quality and reliability of the code** that comprises the kernel and the applications that are provided by distributions.
- **Availability of code**: Linux's source code and all build tools are available without any access restrictions.
- **Broad hardware support** means that Linux supports different types of hardware platforms and devices. Also, it is expected that the software writen on one Linux architecture can be easily ported to another architecture Linux runs on.
- **The variety of tools** existing for Linux make it very versatile. If one thinks of an application he needs, chances are others already felt the need for it. It is also likely that someone took the time to write the tool and make it available on the Internet.
- **Community support** is perhaps one of the biggest strengths of Linux. This is where the spirit of the free software and open source community can be felt most.

Many of these motivations are shared by those in the desktop, server and enterprise spaces, while others are more unique to the use of Linux in embedded devices (Yaghmour et al., 2008).

Linux was conceived and built as a general-purpose multiuser operating system in the model of UNIX. The goals of a multiuser system are generally in conflict with the goals of real-time operation. General-purpose operating systems are tuned to maximize average throughput even at the expense of latency, while real-time operating systems attempt to minimize, and place an upper bound on latency, sometimes at the expense of average throughput (Stallings, 2014).

There are several reasons why standard Linux is not suitable for real-time use (Abbott, 2003):

- **"Coarse-grained Synchronization"** – Kernel system calls are not preemptible: once a process enters the kernel, it can't be preempted until it's ready to exit the kernel. If an event occurs while the kernel is executing, the process waiting for that event can't be scheduled until the currently executing process exits the kernel.

- **Paging** – The process of swapping pages in and out of virtual memory is, for all practical purposes, unbounded. There is no way to know how long it will take to get a page off a disk drive, making a process to be delayed due to a page fault.

- **"Fairness" in Scheduling** – Conventional Linux scheduler does its best to be fair to all processes: the scheduler may give the processor to a low-priority process that has been waiting a long time even though a higher-priority process is ready to run.

- **Request Reordering** – Linux reorders I/O requests from multiple processes to make more efficient use of hardware. For example, hard disk block reads from a lower priority process may be given precedence over read requests from a higher priority process in order to minimize disk head movement or improve chances of error recovery.

- **"Batching"** – Linux will batch operations to make more efficient use of resources. For example, instead of freeing one page at a time when memory gets tight, Linux will run through the list of pages, clearing out as many as possible, delaying the execution of all processes.

The consequences of these issues in using Linux or even Windows a PC are known to us: if we try moving the mouse while executing a compute-intensive task, it occasionally stops and jumps because the computer or I/O-bound process has the

CPU locked up. In a real-time environment this behavior is unacceptable and may even be catastrophic: by definition this is not real-time (Abbott, 2003).

To help solve some of these limitations, there is a Linux kernel patch for hard real-time systems, formally known as RT Preempt, that changes several kernel source files to implement real-time policies on the kernel. However, this patch is insufficient for some scenarios, and other techniques should be used to meet hard to meet deadlines, like user space memory mapping to inhibit boundary crossing memory copying latencies, using DMA peripherals or accelerate applications through programmable hardware (Naia, 2015).

Appendix A.1.2 presents the steps that should be taken in order to apply the RT Preempt patch and compile the kernel.

2.2.4 Real-time Operating Systems

As seen in the last section related to Linux OS, most general purpose scheduling algorithms are not greatly suited for real-time scenarios as task's deadlines are not taken into account and there is uncertainty on when a context switch is going to occur.

For all of the above reasons, real-time embedded systems often deploy **R**eal-**T**ime **O**perating **S**ystem (RTOS). These OSs are operating systems that are specifically designed for real-time application purposes, and its scheduler is designed to provide a predictable (normally described as deterministic) execution pattern usually under a system of priorities, with high priority tasks guaranteed to run under a certain amount of time.

The most common designs for RTOSs are using preemptive schedulers, where tasks only switch when an event of higher priority needs to be served, also called event-driven; or using round robin schedulers, where tasks switch on a regular clocked interrupt, also called time-sharing. Time-sharing designs switch tasks more often than strictly needed, but give smoother multitasking, giving the illusion that a process or user has sole use of a machine.

The following RTOSs are among the top 10 operating systems used in the embedded systems market: Wind River **VxWorks**, Real-Time Executive for Multiprocessor Systems **(RTEMS)**, **Windows** Embedded **CE**, **MontaVista** Embedded **Linux** and **FreeRTOS**.

2.2.5 Buildroot

"Buildroot is a simple, efficient and easy-to-use tool to generate embedded Linux systems through cross-compilation" (Free Electrons, 2016). It is an open-source project, currently developed by Free Electrons, consisting of several makefiles and patches to allow building for multiple target platforms on a single Linux-based development system.

This tool automatically downloads, extracts and builds packages, beeing able to build the required cross-compilation toolchain, create a root file system, compile a Linux kernel image and generate a boot loader for the targeted embedded system, or it can perform any independent combination of these steps. For example, an already installed cross-compilation toolchain can be used independently, while Buildroot only creates the root file system.

It is easily configurable through a set of graphical interfaces being triggered as makefile rules, which allow target package selection, as well kernel fine-tuning, overall enabling an easy configuration of the provided services.

Buildroot is primarily intended to be used with small or embedded systems based on various computer architectures and **I**nstruction **S**et **A**rchitectures (ISAs) including x86, ARM, MIPS and PowerPC. Buildroot also comes with default configurations for several off-the-shelf available embedded boards, such as Cubieboard and the well-known Raspberry Pi.

Figure 2.9 presents menuconfig, one of the possible graphical interfaces for configuration.

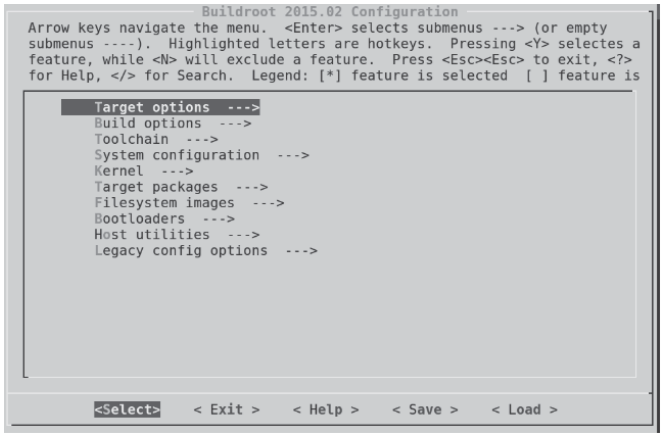


Figure 2.9: Buildroot make menuconfig prompt

Being actively maintained and frequently updated as well as thoroughly documented, it is a very popular tool with some notorious adopters such as Google, Atmel and Analog Devices. Recent hobbyist trends in development boards such as Raspberry Pi or BeagleBone Black have prompted Buildroot popularity also, due to the offered simplicity in getting Linux to run on these boards. Appendix A.1 contains some information on how to use Buildroot, namely downloading and installing it along with compiling a Linux system with real-time kernel patching (Naia, 2015).

2.3 Hardware Acceleration

Some real-time controllers require tight deadlines, proving to be very difficult scenarios where the software resources are not enough to meet the overall system constraints.

As mentioned in subsection 2.2.4, the use of an OS introduces unpredictability and overhead to the computing systems. Sometimes these systems have hard real-time constraints to a point where even enforcing real-time scheduling policies and zero-copy boundary crossing techniques isn't acceptable. In these scenarios, it is a modern practice to incorporate hardware acceleration, making use of the hardware true parallel nature and offload critical application kernels to the custom hardware co-processors, thus freeing up CPU resources (Naia, 2015).

Most of these hardware accelerators are not stand-alone platforms, but are co-processors to a CPU. In other words, a CPU is needed for initial processing, before the compute intensive task is off-loaded to the hardware accelerators (Khatri and Gulati, 2010).

2.3.1 Hardware Description Languages

Hardware **D**escription **L**anguages (HDLs) are programming languages used to describe the structure and behavior of electronic circuits, most commonly digital logic circuits. HDLs can be either vendor designed languages or general languages that are independent of the vendor.

A hardware description language enables a precise and formal description of an electronic circuit that allows for an automated analysis and simulation of such

circuit. It also allows the synthesis for physical deployment.

There are several abstraction levels for these designs, from high-level architectural models to low-level switch models. These levels, from least amount of detail to most amount of detail, are as follows (Maxfield, 2009):

- Behavioral models
  - Algorithmic
  - Architectural
- Structural models
  - RTL
  - Gate level
  - Switch level

Behavioral models consist of code that represents the behavior of the hardware while abstracting its actual register-level implementation. Behavioral models don't include timing numbers, buses don't need to be broken down into their individual signals.

Structural models consist of code that represents specific pieces of hardware. RTL specifies the logic on a register level. In other words, the simplest RTL code specifies register logic. Actual gates are avoided, although RTL code may use Boolean functions that can be implemented in gates. This level is the level at which most digital design is done.

The advantage of HDLs is that they enable all of these different levels of modeling within the same language. Simulating the design at a behavioral level is easy, and then various behavioral code modules can be substituted with structural code modules. For system simulation, this allows analyzing the entire project using the same set of tools. First, algorithms can be tested and optimized, and then the behavioral models can be used to partition the hardware into boards, ASIC, and FPGAs. RTL code may then be developed and substitute in for the behavioral blocks, one at a time, to easily test the functionality of each block. From that, the design can be synthesized , creating gate and switch level blocks that can be again simulated with timing numbers to get actual performance metrics. Finally, this low-level code can be used to generate netlists for layout and implementation

(Maxfield, 2009).

Due to the circuit's concurrent nature, HDL programming paradigms are radically different from control-flow programming paradigms used on languages such as C/C++ (Naia, 2015).

HDL designs are systems that continuously act on a set of inputs and outputs being composed by a set of modules, each with its own set of inputs and outputs, that can be thought of as blackbox subsystems interconnected to form the overall device. Figure 2.10 presents a diagram of an HDL design example, to illustrate the mentioned hierarchical organization.

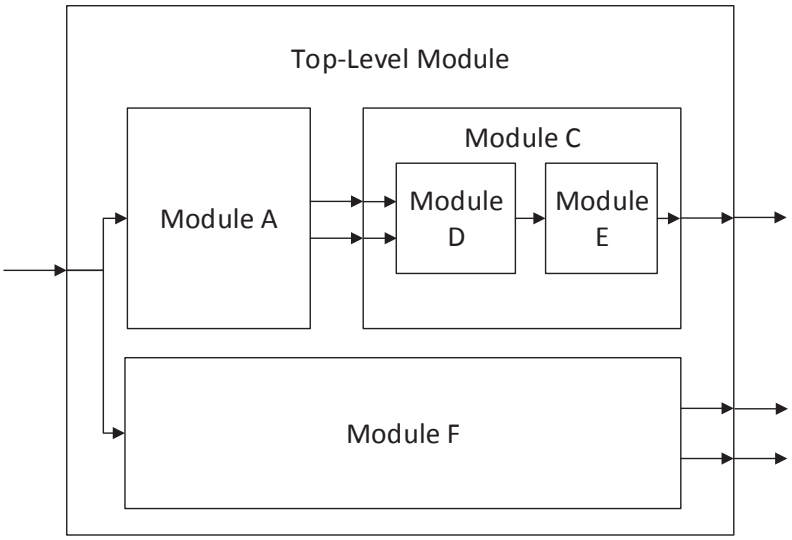


Figure 2.10: HDL design diagram (Naia, 2015)

Synthesis tools compile the HDL code, and generate the physical implementation of the design based on parsed code. However, before the design is properly deployed, it should be validated through the use of HDL simulation tools. The designed devices requires input stimuli and produce outputs accordingly, so in order to simulate the design, testbenches are generally used. A testbench is HDL code that receives a documented and repeatable set of stimuli that is portable across different simulators. A testbench can be as simple as a file with clock and input data or a more complicated file that includes error checking, file input and output and conditional testing. The devices to be tested are usually called **D**evice **U**nder **T**ests (DUTs) or **U**nit **U**nder **T**ests (UUTs), and must also be included in the testbench, which is usually the top-level module.

Under this concept, non-synthesizable constructs are separated from synthesizable construct, mimicking a traditional electronic testbench with signal generation to provide stimuli for the devices under test, subsequently monitor their behavior. Figure 2.11 presents a possible block diagram of an HDL testbench.

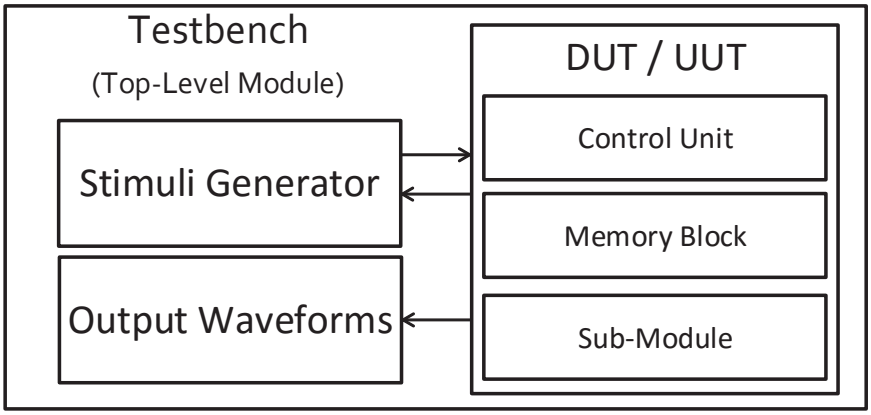


Figure 2.11: HDL testbench diagram

The main HDLs in existence today are Verilog and VHDL. Both are open standards, maintained by standards groups of the **I**nstitute of **E**lectrical and **E**lectronic **E**ngineers (IEEE). Although some engineers prefer one language over the other, the differences are minor. As these standard languages progress with new versions, the differences become even fewer. These system level design languages are still evolving.

VHDL's syntax is rich and strongly typed, deterministic and more verbose than Verilog. As a result, designs written in VHDL are generally extensive, and thus considered self-documenting. VHDL emphasizes unambiguous semantics making it well suited for implementing designs under system level abstractions (Naia, 2015).

Verilog's syntax is very C-like, which is a common language used by engineers in the field, thus providing a good entry level learn-curve. It is weakly typed and doesn't emphasize semantics as rigidly as VHDL, which may cause design problems that are only later found when being used by inexperienced developers. On the upside, it is best suited for lower level abstractions and structural implementations for the same reasons (Naia, 2015).

VHDL is more commonly used in academic contexts and the scientific community, while corporations usually employ Verilog. There are also regional preferences, with Europe showing a predominant use of VHDL and Verilog being more adopted

in America. These languages are not mutually exclusive, with most modern synthesis and simulation tools allowing for designs that mix VHDL and Verilog.

Hardware Verification Languages

For many years, the behavioral coding features of languages such as Verilog, plus a few extras such as display statements and file I/O, gave Verilog-based hardware design engineers all they needed to both model hardware and to define a test bench to verify the model (Sutherland, 2003).

As design sizes increase, the amount of verification required escalates dramatically. Writing test benches and verification routines using pure HDLs is still possible, but the amount of coding far exceeds what can be accomplished in a reasonable amount of time. Due to this, proprietary **H**ardware **V**erification **L**anguages (HVLs) such as OpenVera were developed, specialized in giving verification engineers powerful constructs to describe stimulus and verify functionality in a much more concise manner.

These HVLs typically include features of high-level programming languages such as C++ or Java, along with easy bit-level manipulation similar to those found in HDLs. Many of them may also provide random stimulus generation and other tools to aid complex hardware verification.

Hardware Verification and Design Languages

The proprietary HVLs above-mentioned solve a need, but at the costs of requiring engineers to learn and work with multiple languages, and often at the expense of simulation performance. Having different languages for the hardware modeling and the hardware verification has also become a barrier between those engineers doing the design work and those doing the verification, as they "don't speak the same language" (Sutherland, 2003).

When combining an HDL with an HVL, attempting to merge both design and validation constructs into a single standard, the result is what's called an **H**ardware **D**escription and **V**erification **L**anguage (HDVL).

The first HDVL was SystemVerilog, which was initially only an extensive set of enhancements to the Verilog-2001 standard. In fact, since 2009, when IEEE

merged the Verilog-2005 standard with the SystemVerilog extensions, the Verilog name is gone, and was substituted with SystemVerilog. The new IEEE standard, SystemVerilog-2009, is a complete hardware design and verification language (Sutherland and Mills, 2013), and there is no longer a current Verilog standard.

These enhancements provide powerful new capabilities for modeling hardware at the RTL and system level, along with a rich set of new features for verifying model functionality (Sutherland, 2003). Besides this, SystemVerilog allows convenient interface to foreign languages (currently only C/C++), via SystemVerilog DPI.

Many companies are including support for HVLs and/or HDVLs in their simulation and design tools, such as SystemVerilog or SystemC. SystemVerilog is widely used in the chip-design industry, as the three largest **E**lectronic **D**esign **A**utomation (EDA) vendors (Cadence, Mentor Graphics and Synopsys) have incorporated it into their mixed-language HDL Simulators.

2.3.2 FPGA

FPGAs, briefly mentioned in subsection 2.1.2, are integrated circuits containing reprogrammable hardware. The current FPGA market leaders and long-time industry rivals are Xilinx and Altera, controlling over 80 percent of the market. There are a few other manufacturers, such as Lattice Semiconductor and Microsemi. The FPGA technology's uniqueness and bigger advantage derives from the fact that unlike other processor technologies, the hardware fabric may be reprogrammed, conceding them a huge flexibility. This allows the implementation of devices that are usually implemented in chips using ASIC technology, like microprocessors or other chips hardwired in silicon.

FPGAs contain an array of programmable logic blocks, organized in an array with reconfigurable interconnections amongst them that may be activated or not, implementing simple logic circuits, such as AND and XOR gates, or more complex circuits. These logic blocks are basically composed of two elements: flip-flops and **L**ook**U**p **T**ables (LUTs).

Figure 2.12 presents a diagram the internal architecture of an FPGA.

LUTs implement the behavior of digital gates, by containing a table of gate outputs that are to be used according to gate inputs. By combining digital gates implemented in LUT and flip-flops, any kind of combinatorial or sequential circuit

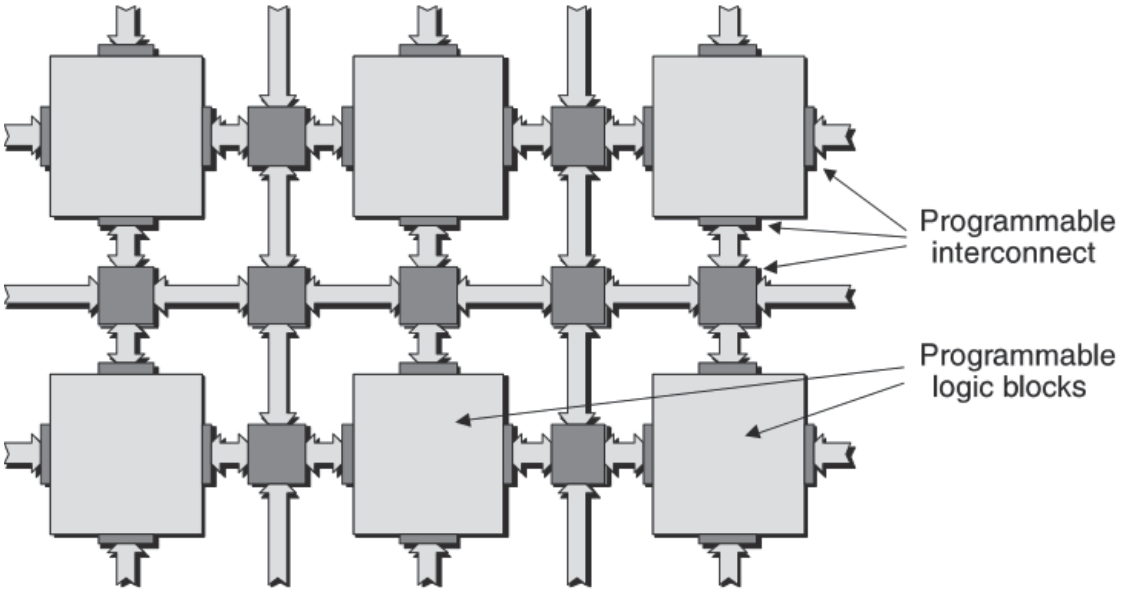


Figure 2.12: FPGA internal architecture (Maxfield, 2009)

may be implemented (Naia, 2015). Figure 2.13 presents a diagram a simplified view of a 2-input LUT implementing an AND gate.

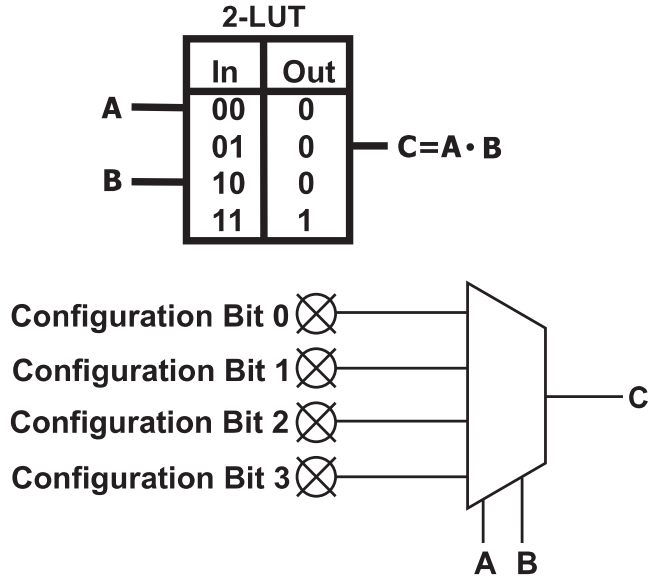


Figure 2.13: FPGA LookUp Table (Huffmire et al., 2010)

LUTs can be very efficient in terms of resource utilization and input-to-output delays, in a design where there is a large number of logic gates between the inputs

and the outputs. However, there are downsides to this approach: in case the design to implement is only composed of a 2-input AND gate, for example, an entire LUT will be used for it. In addition to being wasteful in terms of resources, the resulting delays are high for such a simple function.

In addition to its primary role as a LUT, some vendors allow the cells forming the LUT to be used as a small block of RAM (the 16 cells forming a 4-input LUT, for example, could be cast in the role of a 16 x 1 RAM). Some vendors even allow the **S**tatic **R**andom-**A**ccess **M**emory (SRAM) cells forming a LUT to be used in the form of a shift register (Maxfield, 2009). Thus, each LUT may be considered to be multifaceted, as can be seen in 2.14.

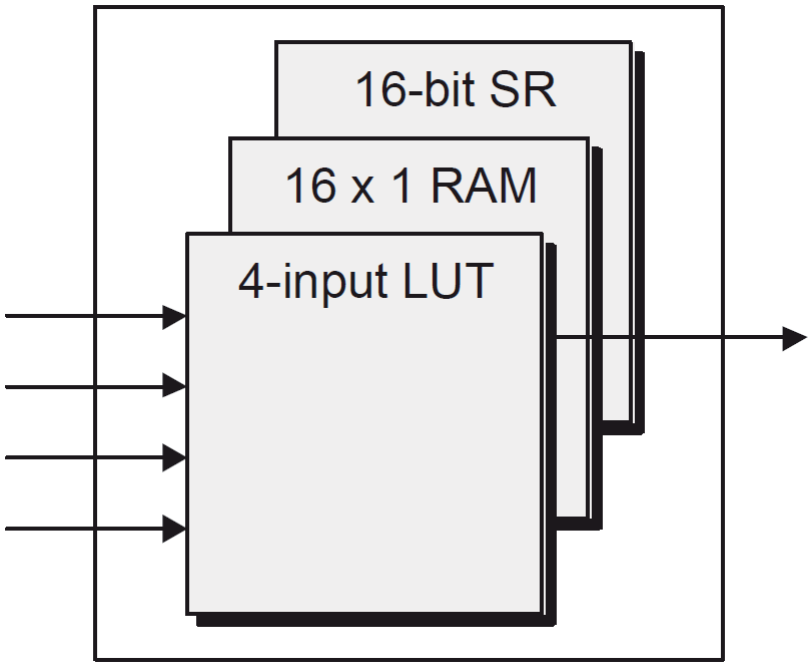


Figure 2.14: View of a multifaceted LookUp Table (Maxfield, 2009)

Logic Blocks - CLBs, LABs & Slices

Most modern FPGA introduce larger elements in the interconnected array, with vendors often adopting their own nomenclature for these elements.

The main blocks in Xilinx's FPGAs are called **L**ogic **C**ells (LCs). Along with other elements, like special fast carry logic for arithmetic operations, an LC contains a 4-input LUT, a multiplexer and a register. This can be seen in Figure 2.15, which is a simplification, but suitable for the purpose.

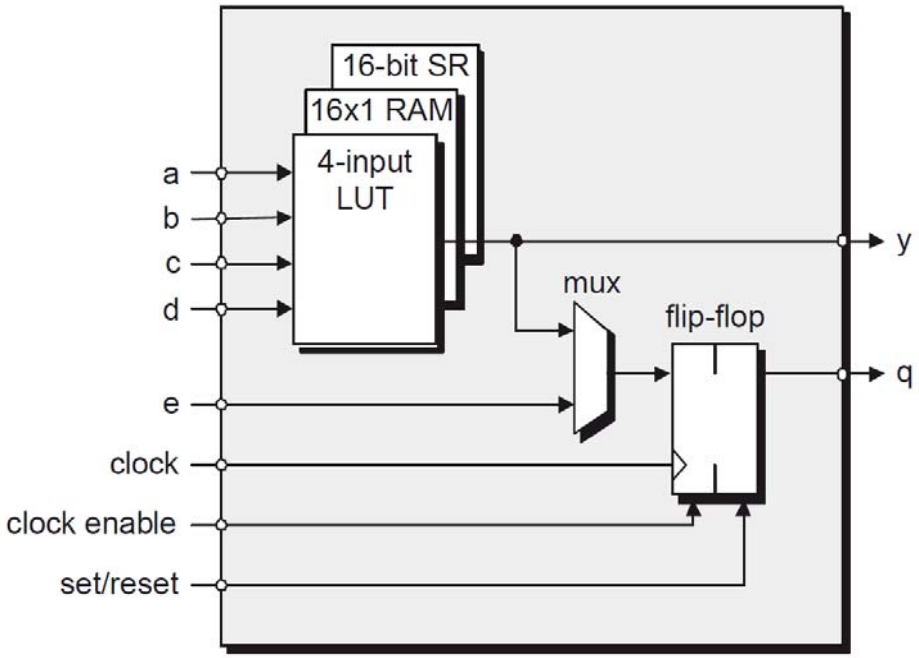


Figure 2.15: Simplified view of a Xilinx LC (Maxfield, 2009)

The equivalent building blocks in Altera's FPGAs are called **L**ogic **E**lementss (LEs). The overall concepts are very similar for the long-time industry rivals' core building blocks, but still there are differences.

Now focusing on Xilinx's FPGA architecture exclusively, the next step up the hierarchy is called a **slice**. Depending on the FPGA family being considered, a slice can contain different numbers of Logic Cells, but usually, slices are composed of two logic cells, as can be seen in Figure 2.16.

Moving one more level up the hierarchy, we come to what Xilinx calls **C**onfigurable **L**ogic **B**locks (CLBs) and Altera refers to as **L**ogic **A**rray **B**locks (LABs). These larger blocks contain LUTs, flip-flops and multiplexers in a single block, allowing for faster designs with better resource utilization.

Using Xilinx's CLBs as an example, 7 series Xilinx FPGAs slices contains four LUTs and eight flip-flops, only some slices can use their LUTs as distributed RAMs or Shift Registers, and two slices form a CLB (Xilinx, 2015). In other FPGA families, each CLB may have four slices, as shown in Figure 2.17: each CLB contains four slices, and each slice contains two LCs.

As mentioned above, implementing all circuits using only LUT-based logic blocks may be rather inefficient and costly when compared to their silicon hardwired

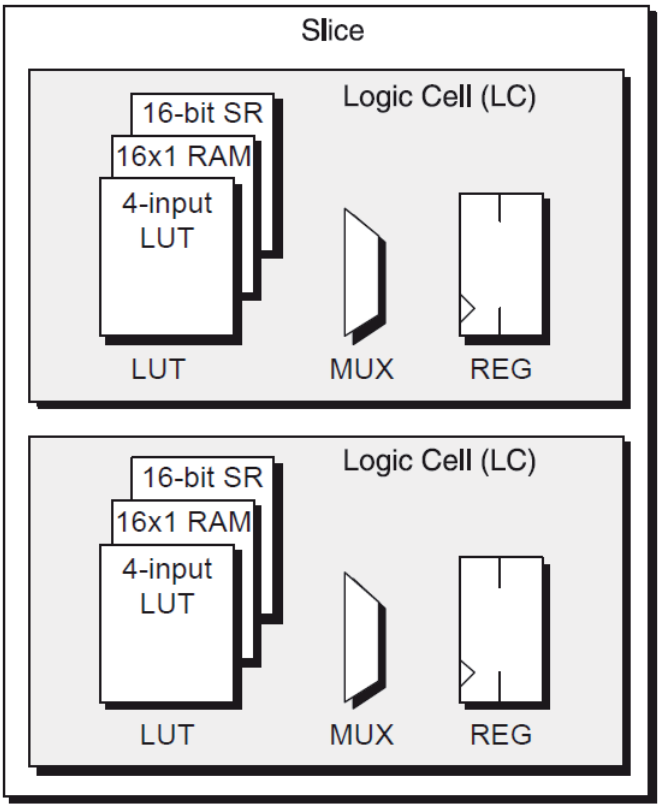


Figure 2.16: A Slice with two Logic Cells (Maxfield, 2009)

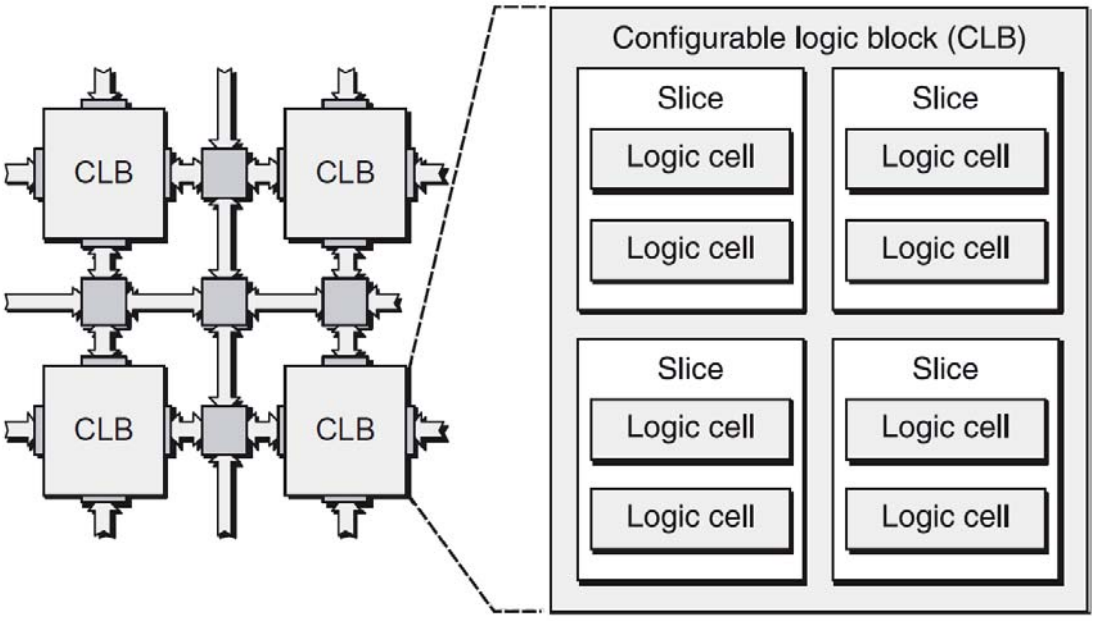


Figure 2.17: A Configurable Logic Block containing four slices (Maxfield, 2009)

counterparts, being slower, occupying large silicon areas and consequently dissipating more power. For this reason modern FPGA chips include hardwired elements that implement commonly used functions such as multipliers, memories and generic DSP blocks. FPGA commonly store their configuration in volatile memories, needing to be programmed when turned on. The configuration file of an FPGA chip is called the bitstream. Most modern FPGA development boards include non-volatile memory chips to store bitstreams, along with circuitry that may program the FPGA chip upon power-on (Naia, 2015).

FPGA SoC

Traditionally, FPGAs were used to design and prototype chips that would be later implemented on ASIC fabric. As mentioned in subsection 2.1.2, ASICs have extremely high development and manufacturing costs, which makes any mistake very costly. Still, implementing product designs on FPGA chips is not a viable option, due to significantly higher power dissipation and latency when compared to hardwired silicon chip designs. Therefore, the chosen flow was to use FPGA as prototyping platforms, and then implement the designs in ASIC chips.

However, in cases where only a small amount of ASIC-based chips is to be built, the chip's mask costs may be unbearable, and with recent advances in FPGA technology, they are now an available option for final product implementations, as is already happening in some modern HD TV models. Using FPGA chips for final products has some other advantages, like allowing hardware architecture updates and reconfigurations, adding further flexibility to designs and eventual hardware bug fixes. Typically, when FPGAs were employed this way, softcore processors were used. Softcore processors are general purpose microprocessors implemented in FPGA fabric, often being implemented along with other peripherals constituting SoC architectures (Naia, 2015).

When the FPGA fabric is being used for hardware acceleration purposes, offloading critical algorithms to dedicated hardware co-processors executing in parallel with the CPU. In these kind of designs, the CPU has better performance if it is hardwired in silicon. Due to this, modern FPGA devices integrate both CPU and FPGA fabrics into a single device, along with other commonly used SoC peripherals, providing higher integration, lower power, smaller board size, and higher bandwidth communication between the CPU and the FPGA. They also include a rich set of peripherals, on-chip memory and high speed transceivers (Altera, 2014).

This kind of systems are refered to as Hybrid Systems.

For designs that already use an FPGA and a separate CPU, an FPGA SoC should definitely be considered. It is likely to provide comparable, even superior functionality and performance, but at a lower board space, power, and system costs. This kind of architecture is demonstrated in Figure 2.18.

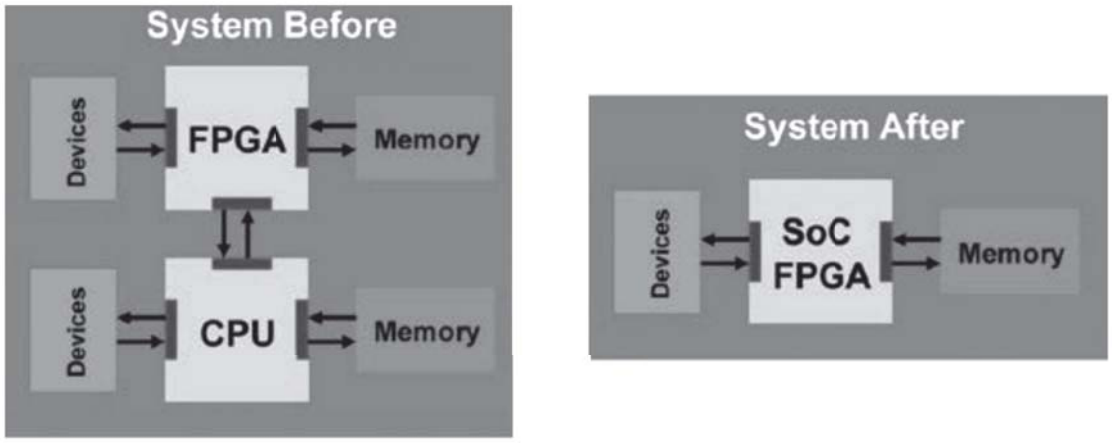


Figure 2.18: Standalone CPU and FPGA integrated into single FPGA SoC (Altera, 2014)

In case both CPU and FPGA use separate external memories, it may also be possible to consolidate both into one memory device for further savings, as shown in the above figure. As the signals between the CPU and the FPGA now reside on the same silicon area, communication between the two consumes substantially less power compared to using separate chips. The integration of thousands of internal connections between the CPU and the FPGA leads to substantially higher bandwidth and lower latency compared to a two-chip solution (Altera, 2014).

Figure 2.19 presents the architecture of the latest Xilinx SoC architecture, the Zynq® UltraScale+™ MPSoC. This heterogeneous processing platform contains a scalable 32 or 64-bit, dual or quad-core ARM® Cortex™-A53 CPU, a dual-core ARM®® Cortex™-R5 **R**eal-time **P**roccessing **U**nit (RPU), an ARM **G**raphics **P**roccessing **U**nit (GPU) core and other elements such as **D**ouble **D**ata **R**ate (DDR) controllers, DMAs and many other peripherals hardwired in silicon, as well as a programmable logic silicon area, all integrated into a single chip (Hansen, 2016).

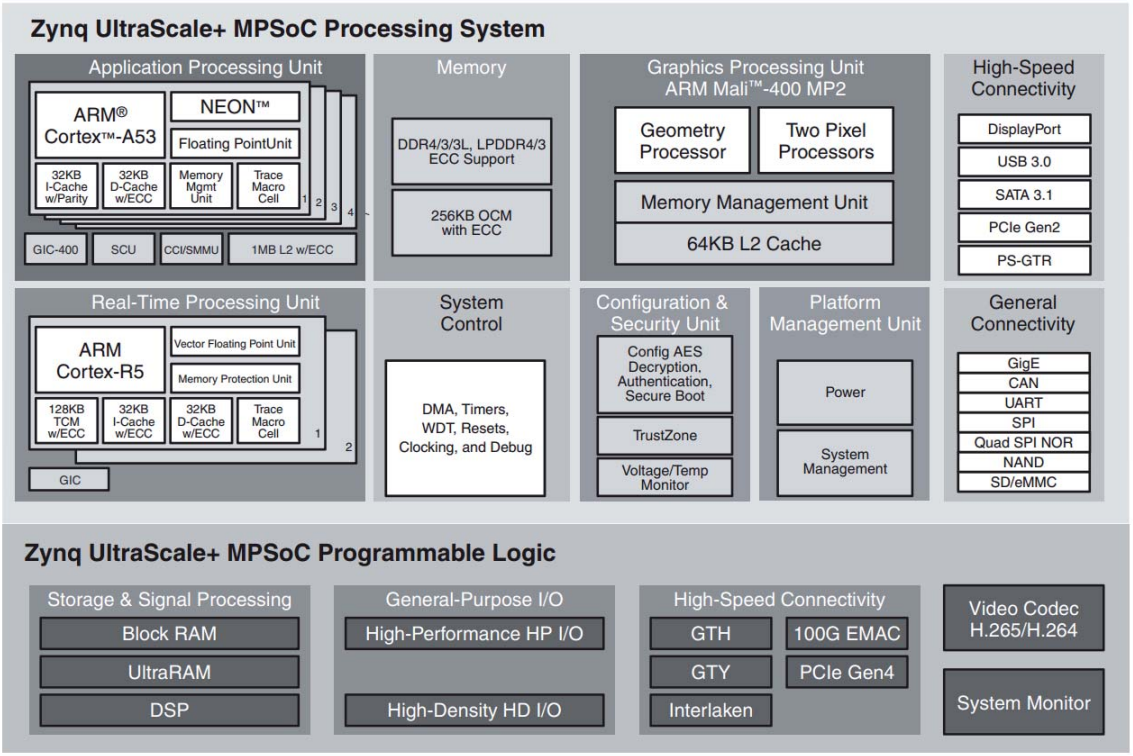


Figure 2.19: Zynq® UltraScale+™ MPSoC Block Diagram (Hansen, 2016)

2.4 Hardware-Software Co-Design

Hardware-Software Co-Design tries to exploit the synergy of hardware and software with the goal of optimizing and/or satisfying design constraints such as cost, performance and power of the final product, along with reducing the time-to-market frame considerably. As seen in the previous sections, embedded systems may depended on co-design, making use of software for flexibility and specialized hardware for performance increase. When referring to real-time embedded systems, using Hybrid Systems brings even more advantages, because the real-time requirements of the system may be more easily achieved.

2.4.1 Hardware-Software Co-Design Flow

When designing hardware accelerated embedded software systems, some patterns should be followed. The first step should be to develop a software application representing the behavior of the system being designed, disregarding any metrics, because in this phase only the algorithms debugging is crucial. The developed application may be deployed in the host platform initially, only later being cross-

compiled for the target platform. After this phase, the applications' core processing tasks are to be identified and later parallelized, using multi-threaded programming models.

Next, profiling tools should be used to characterize the application, so that the most "processor intensive" tasks within the application are identified. This way, some of them can be selected as candidates to be offloaded into hardware FPGA fabric. After identifying the tasks to be converted into HW **I**ntellectual **P**ropertys (IPs), HDLs should be used to develop RTL models that describe the behavior of those selected tasks.

At this point, the system should be cross-compiled to the target architecture, in order to validate the design. At this validation phase, the system should be implemented, in order to ensure that the system metrics are met. This validation is an iterative process, so if the desired metrics are not met, the development may return to the hardware IP development phase, or even the profiling phase. Each one of these steps will be addressed with more detail next.

System Modeling

As above-mentioned, the first step is to model the behavior of the system using a more flexible software programming language, like C or C++, the most used for embedded software. By doing this, the code is more generic and portable, and also allows for debugging on the host machine and cross-compiling for the target architecture. In this phase, the whole system may be tested on the host platform, because only the system's behavior is under test, to validate the algorithms. This allows abstracting the application away from the target platform implementation details, in the early phases of the design.

Software Parallelization

Initially, the system tasks must be identified, dividing the system into smaller parts. Then, software parallelization will be used, by means of multi-thread programming models such as the **P**ortable **O**perating **S**ystem **I**nterface (POSIX) **A**pplication **P**rogram **I**nterface (API), widely used in Unix-based systems. In this step, the multiple task algorithms are assigned to different threads. Since threads usually share data, synchronization mechanisms are mandatory to avoid

race conditions, and are usually provided by the multi-threading API. This technique is not a true nature parallelization, as only one thread is executing at one CPU core in a certain time, hence being a virtual parallelization.

Profiling

Next in the design flow, the application must be analyzed, so that the different tasks are characterized. This step is called the profiling, and is performed using different software profiling tools. Since the differences between the host and target platform are not relevant enough to justify profiling in the target platform, profiling the host application is acceptable. However, if the system has real-time constraints, it is highly recommended that the OS in the host platform runs a real-time scheduler, so that the application being profiled can enforce a deterministic behavior. The profiling tools should be used several times, in order to gather more reliable data from the application. These data allow the developer to identify the critical tasks of the system, along with its bottlenecks.

Hardware Design

The critical tasks identified in the last step are candidates to be offloaded into HW IPs. Hardware IPs are developed with the aid of HDL languages, like the ones described in subsection 2.3.1, namely Verilog or VHDL. The development is done on a Register Transfer Level (RTL), and is then validated through debug on RTL simulation tools, like Vivado Simulator (Xsim) from Xilinx, Modelsim from Mentor Graphics or even Quartus II Simulator (Qsim) from Altera.

After using these simulators to validate the design of the IP, synthesis tools will be used to synthesize it. Routing tools must also be used in order to map the floorplan, so the design can be implemented in an ASIC or FPGA. These tools are usually part of a toolchain, available in **I**ntegrated **D**evelopment **E**nvironments (IDEs), such as Xilinxs' Vivado Design Suite or Alteras' Quartus Prime Design Software.

The development process of this kind of accelerators may be long and complex, and very time consuming even with the current available tools, specially in this approach, as validation must be performed at the target platform. Besides, FPGA coding paradigms are inherently different from software programming paradigms (Naia, 2015).

Using **H**igh **L**evel **S**ynthesis (HLS) tools, the hardware and software domains are brought together. High-level synthesis is a design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements such behavior. Using HLS tools, the algorithms may be developed and verified in software languages like C/C++/SystemC. The code is then analyzed, architecturally constrained and scheduled to create a RTL design, which may then be synthesized. This kind of translation may be less efficient than developing the HW IPs in HDL, but presents undoubted advantages, being more and more used in the industry.

There are some HLS tools on the market, like Vivado High-Level Synthesis (Vivado HLS) from Xilinx, which allows C/C++/SystemC as inputs, and as outputs VHDL/Verilog/SystemC (Xilinx, 2016a). Altera recently announced their new Spectra-Q^{TM} engine, which supports their new A++ compiler for high-level synthesis, allowing the use of C/C++ algorithms to create VHDL/Verilog IP cores, significantly boosting productivity through faster simulation and IP generation (Altera, 2015).

Validation

In this phase the developed system must be validated integrally. Using an integrated co-simulation environment, the development process can be accelerated in a substantial way, anticipating design decisions before committing to the hardware platform and allowing for full-system testing and debugging along with its development. This way, the target platform deployment is done only when full-system integration and validation is concluded.

Implementation / Metrics Verification

In this last phase, the system must be implemented in the target platform, so its functionality can be tested, and the design metrics can be verified. Tools like Xilinx's ChipScope Pro Analyzer, that create a silicon wrapper in the developed system, it is possible to analyze the temporal signals of the physical system, in order to confirm that the temporal requirements are met.

The concept of an integrated co-simulation environment may be expanded, allowing a multitude of metrics to be validated after qualitative validation. Performing

a quantitative validation after the qualitative validation greatly expands the tool's utility (Naia, 2015).

Chapter 3

Co-Simulation Models, Mechanisms and Tools Overview

This chapter presents an overview on hybrid embedded systems simulation, along with some commonly used models and mechanisms to perform mixed-level simulation on such systems. Some simulation tools along with their interfaces that allow for co-simulation are also described.

The Verilog and SystemVerilog simulation interfaces along with PSIM® and its' DLL blocks interface are deeply analyzed since they target the most important domains in the developed work. Finally, the FMI is an important standard for co-simulation and model interchange widely adopted by the industry, and as an important reference for this project's system design, it is also covered.

3.1 Hybrid Embedded Systems Simulation

As mentioned in the last section, simulating an embedded system allows engineers to test designs and simulate the systems, being almost indispensable in the development process. When talking about Hybrid Embedded Systems, comprising hardware acceleration, development is usually done on multiple application domains, with system complexity often standing in the way of accurate simulations.

These domains usually include specialized hardware, the embedded system software used to control hardware, process and retransmit data, and software running on that embedded computing system. Testing all parts of the system separately

is usually difficult, or even impossible, because these systems are usually complex, making the development process a very expensive and time consuming task.

In classical development cycle we have to design the hardware prototype, basing on some assumptions regarding possible software solution. The work on software part may be started when the hardware specifications are ready, but thorough testing of the interactions between the hardware and software parts must be delayed, until the hardware prototype is ready. If the results of tests show the need for significant changes in hardware design, it is necessary to prepare next prototype and subject it to tests. Sometimes the above step must be repeated a few times, depending on the design complexity and the skills of the development team (Zabołotny, 2012).

As seen, both software and hardware simulators are required in such an co-simulation environment:

- Software simulators like QEMU can emulate the target platform and provide instruction-accurate simulation of the software running on it. The machine's hardware is emulated functionally, and the instruction set for the target machine is emulated, allowing for full software stack simulation for the target platform. With this kind of simulation, user application software may be validated, as well as OS components themselves (Naia, 2015).
- Hardware RTL simulators, as already mentioned in section 2.4.1, namely Vivado Simulator (Xsim) from Xilinx and Modelsim from Mentor Graphics, simulate the behavior of hardware specified by HDL sources.

Approaches based on co-simulation of software and hardware components of the designed system are needed to solve the presented problems in hardware-software co-design flow, and some solutions will now be presented.

3.1.1 Full System RTL Simulation

The simples approach would be to use an hardware simulator and perform RTL simulation for the whole system. Figure 3.1 presents a diagram of full-system RTL simulation on a development host.

Although it is possible, the performance of such simulation is usually very poor, except when the system is centered around systems containing FPGA chips connected to simple microcontrollers. Since RTL simulation analyzes the state of all

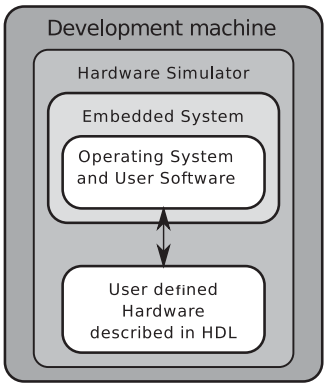


Figure 3.1: Full system RTL simulation diagram (Zabołotny, 2012)

logic gates and registers in every integration step, it is highly ineffective when simulating systems containing bigger embedded computers, running under control of Operating Systems.

An example of the inadequacy of this simulation for software is that a simulation of an OpenRisc CPU running a simple C program takes 40 seconds of simulation time for a simple welcoming message display Balducci (2009). The solution to this problem is to avoid software simulation in RTL simulators altogether, running software parts externally of RTL simulation (Zabołotny, 2012).

3.1.2 RTL Simulation with Host Software

In this approach, hardware designs that need validation are simulated in RTL, while software is ran directly on the host development machine. Figure 3.2 presents a diagram of an RTL simulation being provided with stimuli from software being executed on the host development machine.

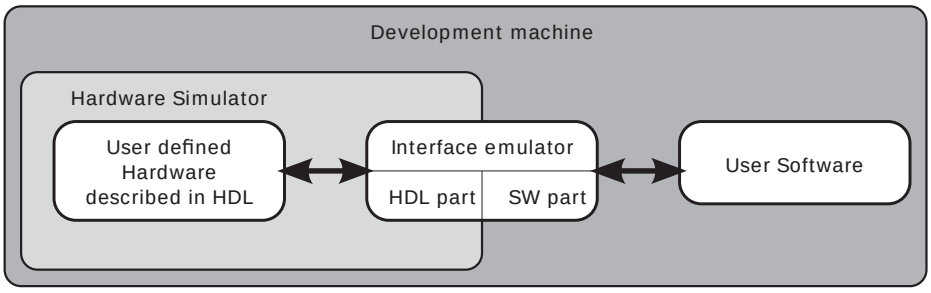


Figure 3.2: RTL simulation with host software diagram(Zabołotny, 2012)

The software application must be compiled and ran on the host machine, and device driver system calls must be replaced with API calls to the RTL simulation

tool, emulating system bus transactions. In the HDL design, an interface that emulates software accesses to the design must also be implemented. This approach presentes advantages when compared to full system RTL simulation, as the hardware simulator only simulates what's relevant. Also, the embedded application behavior may be debugged and validated, being an effective validation method for simple software.

However, more complex software often needs further validation, as OS components are not validated, like device drivers, performance of data transfers, and so forth. To emulate these elements of an embedded system, software simulation must be combined with RTL simulation (Naia, 2015).

3.1.3 RTL-Software Co-Simulation

Using this approach, an software simulator such as QEMU is used together with an hardware simulator. Figure 3.3 presents a diagram of such co-simulation environment.

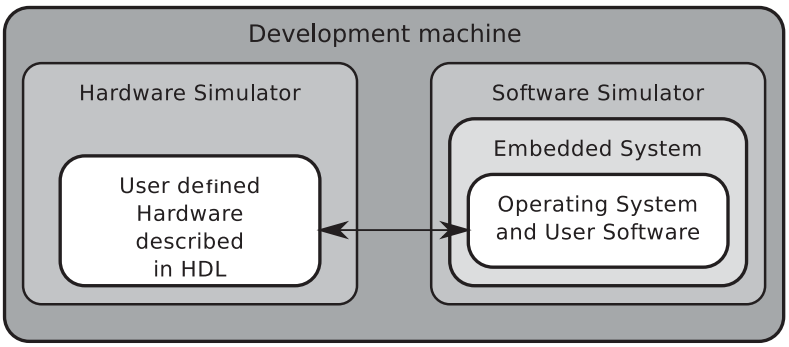


Figure 3.3: RTL-Software co-simulation diagram(Zabołotny, 2012)

This approach is very useful, as the entire software stack may be properly simulated for the target platform and the HDL models being developed can be properly simulated using RTL tools. Interfaces may be implemented to allow data transfer between the simulation tools, such as hardware access information to emulate system bus transactions or simulation time to synchronize simulations. This simulation approach allows the validation of OS elements, such as the device drivers for custom hardware designed.

The downside to this approach is that due to different simulation granularity in both simulations, synchronization may be difficult. If approaches are used which

involve synchronization with other simulation tools simulating other domains, such as physical systems or analog electronics, synchronization may be even impossible, as the RTL simulator will be always delayed in relation to the software simulator (Naia, 2015).

3.1.4 Full System Software Simulation

In this approach, the full system is emulated in the software simulator, and hardware is only emulated functionally, as represented in Figure 3.4.

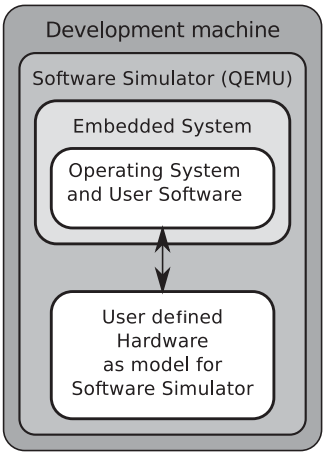


Figure 3.4: Full system software simulation diagram(Zabołotny, 2012)

Here, hardware models are implemented functionally and integrated into the software simulation. This can only be done when using open-source simulators, using the simulator's internal framework and adding them to the list of supported models, or for simulators that allow loading external models as plugins. When using this approach, device driver development and validation may start earlier, before any commitment to an HDL implementation. This allows for a bigger flexibility in the design and for concurrent development, given that software design teams may start device driver development concurrently with hardware design teams.

3.2 QEMU

QEMU, which stands for **Q**uick **EMU**lator, is an open source emulator and virtualizer who uses dynamic translation to achieve good emulation speeds. When used as an emulator, QEMU is capable of full-system emulation, emulating functionally

system components like interrupt controllers and memories, so QEMU will be used as an emulator during this work, with its virtualization capabilities being ignored.

It is a very useful tool in the context of embedded system development, enabling development and debug of a target platform system without a physical target machine. The entire software stack may be developed entirely in QEMU, and then deployed to the target machine once validated and debugged (Naia, 2015).

When QEMU was first developed by Fabrice Bellard, it was a major breakthrough due to its dynamic binary translation algorithms. It was widely adopted as an emulator by companies for development purposes, although each company maintained their internal private versions of QEMU, implementing modifications as suited and implementing support for their machines and platforms. Nowadays, QEMU continues to be very popular, as it is part of the Android **S**oftware **D**evelopment **K**it (SDK) and Xilinx's PetaLinux solution as an emulator. Popular virtualization products such as VirtualBox or Xen-HVM also draw heavy inspiration on QEMU (Naia, 2015).

Since QEMU is open source, as already mentioned, many extensions may be developed to increase its' features. These extensions are crucial in order to use QEMU in a co-simulation context, enabling hardware devices to be modeled externally in other simulation tools. The need for these extensions led to the development of two extensions in the previous work developed by Naia (2015): firstly, the QEMU Plugin Extension, that allows QEMU to load behavioral hardware models as dynamic libraries, which is useful for device driver development during a design space exploration phase, earlier in the project. An External Model Extension for QEMU was also developed, opening up the possibility to use connect QEMU with RTL simulators and/or domain specific simulators, such as PSIM for the power electronics domain. This extension, along with the fact that QEMU emulates the machine's hardware functionally, makes possible the creation of an integrated co-simulation environment.

3.2.1 QEMU Plugin Extension

This extension, developed by Naia (2015), intends to allow developers to speed-up device driver development, by abstracting some of the intricacies of QEMU's internal API, thus making it easier to extend QEMU with a hardware device *ex nihilo* via a simplified API, and secondly by reducing compilation effort by allowing

the use of plugins containing device models.

This is specially useful when the project is in an early stage and hardware IPs are being designed, enabling the designer to try out behavioral hardware models and their respective device drivers (Naia, 2015).

Figure 3.5 presents an overview of a hardware accelerated embedded Linux case of study running on a QEMU emulation that makes use of the extended plugin capabilities.

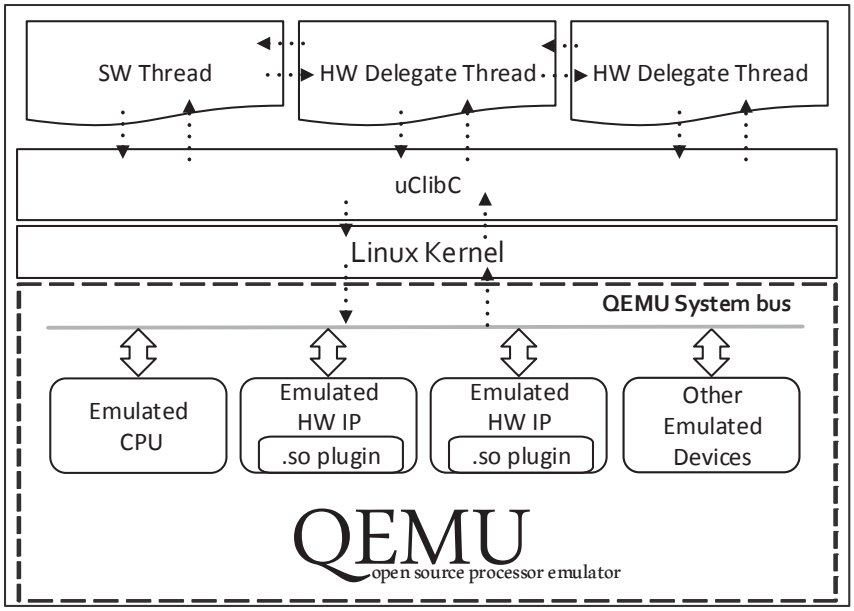


Figure 3.5: Plugin extension overview (Naia, 2015)

Hardware accelerators are accessed whenever device drivers perform hardware transactions, and are modeled in the emulated machine through device models contained in .so files. For each shared object that contains a plugin device model, there is a corresponding plugin interface device object that is instantiated in QEMU (Naia, 2015)

Plugin interface devices are instantiated per existing plugin device. They are instantiated according to information loaded from the plugin device, such as its' base address and hardware transaction behavior. The plugin interface device is mapped in the emulated machine address space upon instantiation using memory region information that is loaded from the corresponding plugin device.

Hardware transaction functions are also registered upon plugin interface device

instantiation, and are executed whenever a plugin interface hardware transaction function gets called (Naia, 2015).

3.2.2 QEMU External Model Extension

This extension, developed by Naia (2015), intends to allow QEMU to be used in a co-simulation context, enabling hardware devices to be modeled externally in other simulation tools. This is useful in several application domains, namely in hardware acceleration.

Figure 3.6 presents an overview of a hardware accelerated embedded Linux system running on QEMU, along with its extended co-simulation capabilities. QEMU translates and runs the whole software stack, including the Linux kernel, the C-library and the embedded application that's running on top of the operating system.

A hardware accelerated embedded application is usually composed by a collection of threads, including hardware delegate threads, who delegate processing algorithms to hardware accelerators by interacting with them via device driver system calls. This extension enables hardware accelerators to be modeled externally, by incorporating models contained in other simulation tools into the QEMU emulation (Naia, 2015).

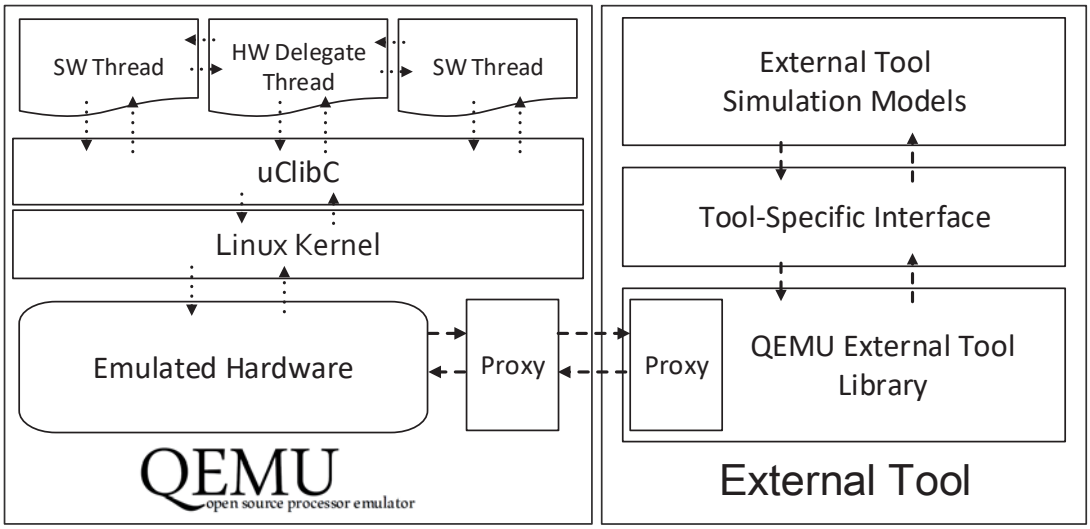


Figure 3.6: QEMU External Model Extension overview

To make QEMU use these features, an '-ext-tools' argument must be provided on the QEMU launch command. The following is a template of a QEMU launch command with external tools co-simulation active (Naia, 2015):

```
$ qemu-system-[arch] [flags] -ext-tools [number of tools
   ],[server port(optional)]
```

At start-up, a server is started to synchronize with external tools. This server is temporary and blocks QEMU until the requested number of external tools establish the connection. Each tool must provide information, such as its name, domain or number of models. Information on every model must also be provided, like its name, base address, block size or **I**nterrupt **R**e**Q**uests (IRQs) (Naia, 2015).

When hardware transactions occur, their information will be issued to the respective external tool and then, usually through tool-specific interface frameworks or libraries, to the respective external model.

3.3 (System)Verilog Simulation Interfaces

Most FPGA design flows are heavily conditioned by the vendors' IDEs and imposed workflows. Using use open-source tools is only possible to a certain extent, so it is relevant to adopt standard HDL interfacing mechanisms when implementing HDL simulation interfaces, as not only to be independent of simulation tools, but also to be able to use commercial simulation tools that are supported by the the main vendors HDLs (Naia, 2015).

In the next subsections, Verilog simulation interfaces will be presented, as it is the HDL of choice in this dissertation, along with SystemVerilogs' simulation interfaces with foreign languages, which can be used to perform co-simulation involving hardware accelerators.

3.3.1 Verilog Programming Language Interface

Due to Verilog being a weakly typed language and not having many constructs that support validation, system verification is not so easy to do, unlike for instance VHDL. One of the major strengths of Verilog HDL is its' PLI, which allows

users and Verilog application developers to infinitely extend the capabilities of the Verilog language and simulations (Naia, 2015).

Verilogs' PLI has had several upgrades: the "old" standard, known as PLI 1.0, was released in 1990 and uses the **T**ask/**F**unction (TF) and **ACC**ess (ACC) routines; the new standard, known as PLI 2.0 , was released in 1993 and makes use of the **V**erilog **P**rocedural **I**nterface (VPI) routines. It is important to note that the VPI functions provide a nearly 100% duplication of the functionality of the TF and ACC functions. This redundancy is necessary in order to provide full PLI capability in the new PLI standard, and yet remain fully backward compatible with the old standard (Sutherland, 1998).

PLI Overview

A PLI application is a user-defined C language application which can be executed by a Verilog simulator (Sutherland, 2002). The PLI application can interact with the simulation in numerous ways, either by reading or modifying simulation logic values or performing certain actions in a specific simulation moment (Naia, 2015).

The interaction with the simulation is performed via user-defined system tasks/-functions, who are, in the Verilog language, commands executed by Verilog simulators. Their names must always start with a dollar sign ($) (Sutherland, 2002).

User-defined system tasks/functions are associated with a PLI application. When a Verilog simulator encounters a system task/function name, it will execute the PLI application code associated with the name (Sutherland, 2002).

3.3.2 SystemVerilog Direct Programming Interface

As seen in the above subsection, Verilog does not have many constructs for validation, and that problem can be solved using the VPI routines. But as seen in section 2.3.1, those limitations were most recently suppressed with the coming of SystemVerilog HDVL.

The PLIs greatest strength is also its greatest weakness, as the learning curve related to the Verilog PLI is somewhat steep, as it includes understanding the PLI API and workflow, and knowing how to compile and link a PLI application to a specific simulator (Spear and Tumbush, 2012).

As also seen in section 2.3.1, SystemVerilogs' DPI allows for simple C/C++ code invocation without complex system task definitions and other PLI associated implementations (Naia, 2015).

Still, the DPI has some weaknesses when compared to the PLI, as it doesn't provide a way to interact with simulation data structures or any other complex interactions like synchronization with time or value changes.

Even so, SystemVerilog DPI was adopted in this dissertation in order to develop simulation extensions that support co-simulation approaches with QEMU, as it is a faster alternative to the PLI when it comes to validating code integration.

DPI Overview

The Direct Programming Interface (DPI) is an easier way brought by SystemVerilog to communicate with C, C++, or any other foreign language. Currently SystemVerilog only supports an interface to the C language, with C++ code having to be wrapped to "look like C". With a little work other languages may also be used (Spear and Tumbush, 2012).

Once a C routine has been declared or "imported" with the import statement, it can be called as if it were a standard SystemVerilog routine (task or function), and C code can also call SystemVerilog routines (Spear and Tumbush, 2012). With the DPI, the SystemVerilog code is unaware that it is calling C code, and the C function is unaware that it is being called from SystemVerilog. Also, values can be directly passed to the C functions and received directly back from them (Sutherland, 2004).

DPI Import Declaration

The DPI import declaration defines the prototype of the C function name, arguments and function return type. A C function can be imported as either a SystemVerilog task, or as a SystemVerilog function. A task in SystemVerilog can input and output arguments, but does not return a value, while functions can (Sutherland, 2004).

The next example shows the import declaration to allow SystemVerilog code to call the "sin" funcion in the C math library (Sutherland, 2004). SystemVerilog code

can call Unix functions directly by importing them, with no need for a wrapper.

```
import "DPI" function real sin(real in);
```

The *import* statement declares that the SystemVerilog routine *sin* is implemented in a foreign language such as C. The modifier *DPI* specifies that this is a Direct Programing Interface routine, and the rest of the statement describes the routine arguments (Spear and Tumbush, 2012).

The next example shows the import declaration to allow SystemVerilog code to call the user-defined "file_write" function (Sutherland, 2004).

```
import "DPI" task file_write(string data);
```

If the name of the imported C function conflicts with a SystemVerilog name, the function can be imported using a new name. In the following example, the C function expect is mapped to the SystemVerilog name *fexpect*, since the name *expect* is a reserved keyword in SystemVerilog. The name *expect* becomes a global symbol, used to link with the C code, whereas *fexpect* is a local SystemVerilog symbol. SystemVerilog does not allow routines overloading (Spear and Tumbush, 2012).

```
import "DPI-C" task expect = function int fexpect();
```

The DPI import declaration can be placed anywhere a native SystemVerilog function can be defined, like modules, interfaces, program blocks or clocking blocks, and the imported routine will be local to the declaration space in which it is declared.

DPI Argument Passing and Return Values

Imported C functions can have any number of formal arguments, including zero. By default, each formal argument is assumed to be an *input* into the C function. The DPI import declaration can override this default by declaring each argument as an *input*, *output* or bidirectional *inout* argument. In the following example, a square root function is defined as having two arguments: a double precision *input* and an output flag representing an error (Sutherland, 2004).

```
import "DPI-C" function real sqrt(input real base,
   output bit error);
```

In order to prevent any bugs in the code, any input arguments should be declared as *const* in the C code, as the C function should not modify its copy of the argument value. Declaring the input arguments as *const* allows the C compiler to warn the user about any writes to that variable. The example below shows an example of C code representing the above-mentioned (Spear and Tumbush, 2012).

```
int factorial(const int i)
{
        if(i <= 1) return 1;
        else return i*factorial(i - 1);
}
```

The C function that is imported as a function into SystemVerilog can have any return value type that is legal in the C language, such as *char*, *int*, *short*, *float*, *double*, *void* or even a pointer. SystemVerilog extends Verilog by adding a *void* data type and a special *chandle* data type for importing C functions that return a pointer data type. The C pointer can be saved in a *chandle* variable, and passed back to other imported C functions as a function argument (Spear and Tumbush, 2012).

DPI Argument Data Types Restrictions

Each variable that is passed through the DPI has two matching definitions: one for the SystemVerilog side, and one for the C side. Table 3.1 presents the data type mapping between SystemVerilog and C (Spear and Tumbush, 2012).

As seen in the table, SystemVerilog vectors of *reg*, *logic* and *bit* data types can be passed into and out of imported C functions. How these vectors are represented in C can be complex, and will not be further addressed, as it is explained in the SystemVerilog standard (IEEE, 2013).

Also, the SystemVerilog Standard (IEEE, 2013) limits imported function return values to "small values", which include *void*, *byte*, *shortint*, *int*, longint, *real*, *shortreal*, *chandle* , and *string* , plus single bit values of type *bit* and *logic* . A function cannot return a vector such as *bit [6:0]* (Spear and Tumbush, 2012).

It is the users' responsibility to use compatible types for both languages declarations, as the SystemVerilog simulator cannot compare the types at the *import*

Table 3.1: Data types mapping between SystemVerilog and C

SystemVerilog Data Type	**C Data Type**
byte	char
shortint	short int
int	int (32-bit)
longint	long long
real	double
shortreal	float
chandle	void *
string	const char *
bit	unsigned char
logic/reg	unsigned char
bit[N:0]	svBitVecVal
reg[N:0], logic[N:0]	svLogicVecVal
unsized array[]	svOpenArrayHandle

statement and the DPI does not provide a mechanism for the C functions to test what type of value is on the SystemVerilog side. In the above-mentioned Verilog PLI, there are mechanisms for the C function to test the data types of system task/function arguments. But when using DPI, the C function simply reads or writes to its arguments, unaware that it was actually called from the SystemVerilog language. If the SystemVerilog prototype does not match the actual C function, it might read or write erroneous values (Sutherland, 2004).

In order to aid engineers using SystemVerilog DPI meeting these requirements, the largest EDA vendors are incorporating some features in their design tools and/or simulators. These tools usually analyze the SystemVerilog code and create a file with the C headers for any routine that may have imported. This file may be included in the C code, along with the *svdpi.h* file (which contains the definitions for SystemVerilog DPI structures and methods), so the C compiler can warn the user in case he is not complying with the headers provided by the SystemVerilog compiler.

As an example, Xilinxs' Vivado Design Suite ships with an *elaborator* named *xelab*. This HDL "compiler" includes some DPI-related switches that help binding the C code to SystemVerilog, and amongst them is *-dpiheader*. When used, this switch generates a DPI C header file containing C declaration of imported and exported functions (Xilinx, 2016b).

DPI Pure, Context and Generic Imported Methods

The DPI allows classifying C imported functions in order to prevent improper declarations that can lead to unpredictable simulation behavior or software crashes, as it does not check for proper declarations.

A call to a C function that was incorrectly declared as pure may return incorrect or inconsistent results, and can cause unpredictable run-time errors, even crashing the simulation. Similarly, if a C function accesses the Verilog PLI libraries and is not declared as a context function, unpredictable simulation results can occur, or the simulation may crash (Sutherland, 2004).

- **Pure functions**: the results of the function must depend solely on values that are passed into the function through formal arguments. The SystemVerilog compiler may optimize calls to a pure function if the result is not needed, or replace the call with the results from a previous call with the same arguments, improving simulation performance (Spear and Tumbush, 2012).

  A pure function cannot use global or static variables, cannot perform any file I/O operations, cannot access operating system environment variables and cannot call functions from the Verilog PLI libraries. Only non-void functions with no *output* or *inout* arguments can be specified as *pure*. Pure functions cannot be imported as a Verilog task (Sutherland, 2004).

  Next is an example of an imported C function declared as pure (Sutherland, 2004).

  ```
  import "DPI-C" pure function real sin(real in);
  ```

- **Context functions**: context C functions are aware of the SystemVerilog hierarchy scope in which the function is declared, which is needed when they must access information relative to that scope. This allows imported C methods to call functions from the Verilog PLI libraries, allowing DPI functions to take advantage of PLI features (Sutherland, 2004). On the other side, overhead is added to the simulation when invoking a context imported routine as the simulator needs to record the calling context (Spear and Tumbush, 2012).

  Next is an example of a context-dependent imported task (Sutherland, 2004).

  ```
  import "DPI-C" context task print(input int file_id,
  ```

```
    input bit [127:0] data);
```

- **Generic functions**: There is no reference in the SystemVerilog Standard (IEEE, 2013) to this kind of methods, but Sutherland (2004) refers to them as *generic*.

  In case an imported method needs to acess global storage, it cannot be declared as *pure*, as seen above. But if it also does not need to access the PLI libraries, there is also no need to declare it as *context*, as it will add overhead. By default, an imported routine is generic, and can be imported as either a SystemVerilog function or a SystemVerilog task. The task or function can have *input*, *output* and *inout* arguments. Functions can have a return value or be declared as void (Sutherland, 2004).

DPI Export Declaration

In addition to importing functions from C, the DPI allows SystemVerilog tasks and functions to be exported and subsequently called from C code (or potentially other foreign languages) (Sutherland, 2004).

Export declarations are similar to DPI import declarations, except that only the name of the SystemVerilog routine is specified, with the type and arguments of the routine not being listed as part of the DPI export declaration (Spear and Tumbush, 2012). Next is an example of one of these declarations.

```
export "DPI-C" myfunc;
```

Optionally, a different name can be given to the task or function within the C language, as seen in the example below (Sutherland, 2004).

```
export "DPI-C" sv_func = myfunc; // myfunc is now called
    "sv_func" within C
```

Only one DPI export declaration for a task or function is allowed, and the formal arguments of an exported task or function must adhere to the same data type rules as with DPI import declarations (Sutherland, 2004).

In SystemVerilog, a task can call other functions or tasks, but a function can only call other functions. This restriction is also true for exported tasks or functions. An exported SystemVerilog function can only be called from a C function that has

been imported as a context function or context task. An exported SystemVerilog task can only be called from a C function that is imported as a context task (Sutherland, 2004).

SystemVerilog tasks can consume simulation time through the use of nonblocking assignments, event controls, delays, and wait statements. When a C function calls an exported SystemVerilog task that consumes time, execution of the C function will halt until the SystemVerilog task completes execution and returns back to the calling C function.

The ability for C functions to call SystemVerilog tasks and functions is a powerful capability that is unique to the DPI. There is no equivalent to exporting tasks and functions in the Verilog PLI standard (Sutherland, 2004).

Connecting DPI to Other Languages

The latest SystemVerilog standard published by the IEEE (2013) only defines the DPI interface for the C language, although it was designed to be an extensible interface to support other languages. That standard defines two layers for the DPI: the SystemVerilog layer and the foreign language layer.

The SystemVerilog layer contains the DPI import and export declarations along with the rules for calling imported foreign functions from SystemVerilog code. This layer will look the same regardless of what foreign language is being called.

The foreign language layer is only defined for the C language in the latest DPI standard for SystemVerilog(IEEE, 2013). Definitions for additional foreign language layers could be added as part of future versions of the SystemVerilog standard. Proprietary foreign language layers may also be defined for other languages. Since the SystemVerilog layer is independent of the foreign language layer, it is transparent to SystemVerilog as to what foreign language an imported function is defined in (Sutherland, 2004).

3.4 PSIM®

In the last two sections QEMU and the SystemVerilog DPI were described, as they are tools that allow the development of an integrated co-simulation environment,

respectively by emulating the target machine's hardware and allowing mixed-level simulation with HDL simulators. In a co-simulation environment, other tools may be used, namely some domain specific simulators. In the case of the power electronics domain, PSIM® is a good alternative, as it is a simulation software specifically designed for power electronics systems.

3.4.1 PSIM® Overview

The PSIM® simulation environment includes PSIM® Schematic and PSIM® Simulator along with the waveform processing tool SIMVIEW, that can act as the co-simulation environment waveform demonstrator (Powersim, 2016).

A circuit in PSIM® is represented in four blocks: power circuit, control circuit, sensors and switch controllers (Powersim, 2016). These blocks, along with their relationships, can be seen in 3.7.

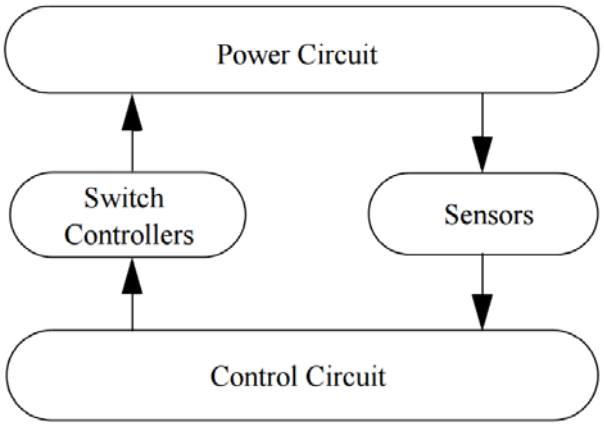


Figure 3.7: PSIM circuit structure (Powersim, 2016)

- The **power circuit** may be composed of switching devices, RLC branches, transformers and coupled inductors.
- The **control circuit** is represented via block diagrams, such as s-domain and z-domain components, logic components (logic gates and flip flops) and non-linear components (such as multipliers and dividers).
- **Sensors** are used to measure power circuit quantities (like current, voltage, active power, reactive power, etc) and pass them to the control circuit.
- Gating signals are then generated by the control circuit and sent to the **switch controllers** in order to control the switching devices present in the

the power circuit (Powersim, 2016).

The feature that makes PSIM® the power systems domain simulator of choice in this dissertation is its' Function Blocks, namely its' external DLL block component, which allows PSIM® to interact with other simulators or tools.

These external DLL blocks allow users to write code in C/C++, compile it into a DLL and then link it with PSIM®. A DLL block receives values from PSIM® as its' inputs, performs the needed calculation and sends the results back to PSIM® (Powersim, 2016).

3.4.2 PSIM® DLL Blocks

Two types of DLL blocks are provided with PSIM®:

- **Simple DLL Block**: fixed number of inputs and outputs, and the DLL file name is the only parameter that needs to be defined.
- **General DLL Block**: allows the user to define arbitrary number of inputs/outputs and additional parameters.

General DLL Blocks Interface

The general DLL block provides more flexibility and capability in interfacing PSIM® with custom DLL files. These libraries export four functions to PSIM®, being that the simulation engine uses three of them, and the other one is used by the user interface (Powersim, 2004).

- **Simulation Functions**
  - **RUNSIMUSER**: This function is the only one in the DLL routine that is mandatory, and is called by PSIM® at each **time step**. The prototype of this function, along with every variable meaning, is presented below (Powersim, 2004).

```
void RUNSIMUSER(
     double t, // Time in seconds.
     double delt, // Time step in seconds.
     double *in, // Array of input values.
     double *out, // Array of output values.
```

```
    void ** ptrUserData, // Pointer of the user
       -defined data.
    int *pnError, // On successful return, set
       to 0. On error, set it to 1.
    char * szErrorMsg // Error message string.
       )
```

– **OPENSIMUSER**: This function is optional, and is called only once at the **beginning** of the simulation. It receives information from the DLL routine, and allows memory allocation for its own use. The prototype of this function, along with every variable meaning, is presented below (Powersim, 2004).

```
void OPENSIMUSER(
    const char *szId, // String ID of the DLL
       block.
    const char * szNetlist, // Netlist string
       of the DLL block.
    void ** ptrUserData, // Pointer to the user
       -defined data.
    int *pnError, // On successful return, set
       pnError to 0. On error, set it to 1.
    LPSTR szErrorMsg, // Error message string.
    void * pPsimParams // Pointer to a
       EXT_FUNC_PSIM_INFO structure.)
```

– **CLOSESIMUSER**: This function is optional, and is called only once at the **end** of the simulation. Its main purpose is to allow the DLL to free any memory or resources that it has allocated. The prototype of this function, along with every variable meaning, is presented below (Powersim, 2004).

```
void CLOSESIMUSER(
    const char *szId, // String ID of the DLL
       block.
    void ** ptrUserData // Pointer to the user-
       defined data.)
```

- **User Interface Function**

  - REQUESTUSERDATA: This function is optional, and handles the user interface with PSIM®. It is called by PSIM® when the general DLL block element is created or its properties are modified in the property box. The prototype of this function, along with every variable meaning, is presented below (Powersim, 2004).

```
void REQUESTUSERDATA(
     int nRequestReason, // Describes the user
        action when the function is called.
     int nRequestCode, // Describes the
        information being requested.
     int nRequestParam,
     void ** ptrUserData, // Pointer to the user
        -defined data. Allows the user to manage
        memory (allocate and free).
     int * pnParam1,
     int * pnParam2,
     char * szParam1,
     char * szParam2)
```

    nRequestParam depends on *nRequestCode's* value. Also, *pnParams* and *szParams* are dependent of *nRequestReason*, *nRequestCode* and *nRequestParam* values.

3.5 Functional Mock-up Interface

FMI is a tool independent standard to support both model exchange and co-simulation of dynamic models using a combination of .xml files and C-code (either compiled in DLL/shared objects or in source code) (MODELISAR and Modelica Association, 2014).

The development of FMI was initiated and organized by Daimler AG within the **I**nformation **T**echnology for **E**uropean **A**dvancement 2 (ITEA2) project MODELISAR. Its' first version, FMI 1.0, was published in 2010. The primary goal is to support the exchange of simulation models between suppliers and **O**riginal

Equipment **M**anufacturers (OEMs), even if a large variety of different tools are used (Blochwitz et al., 2011).

As of today, the FMI specification is now managed and developed as a Modelica Association Project, according to the Modelica Association Bylaws, and profits with the membership of companies such as BOSCH or Daimler, along with some research institutes. FMI 1.0 is supported by over 80 tools, with FMI 2.0 already being supported by over 40 tools. A table with all the tools that currently support the FMI standard can be found online, at the FMI standard site, on `https://www.fmi-standard.org/tools`. This standard is used by automotive and non-automotive organizations throughout Europe, Asia and North America (MODELISAR and Modelica Association, 2014).

As the MODELISAR project ended in December 2011, the new FMI version, 2.0, began to be developed. This new version, published in 2014, combines the formerly separated interfaces for Model Exchange (MODELISAR, 2010a) and Co-Simulation (MODELISAR, 2010b) in one standard. The new specification document was clarified, which increases the compatibility of implementations. New features ease the use and increase the performance especially for larger models (Blochwitz et al., 2012). FMI 2.0 is not backwards compatible to FMI 1.0 (MODELISAR and Modelica Association, 2014).

The FMI specifications and source code that accompanies the specification documents are provided under the **B**erkeley **S**oftware **D**istribution (BSD) license. Modifications must be also provided under the BSD license (MODELISAR and Modelica Association, 2014).

3.5.1 FMI Overview

The FMI defines an interface to be implemented by an executable called **F**unctional **M**ock-up **U**nit (FMU). The FMI functions are used by a simulation environment in order to create one or more instances of the FMU and to simulate them, typically together with other models. A FMU may either have its own solvers, as in FMI for Co-Simulation, or require the simulation environment to perform numerical integration, as in FMI for Model Exchange (MODELISAR and Modelica Association, 2014).

An FMU is distributed in one zip file, containing:

- The FMI Description File, in e**X**tensible **M**arkup **L**anguague (XML) format.
- The C sources of the FMU, including the needed run-time libraries used in the model, and/or binaries for one or several target machines, such as Windows dynamic link libraries (.dll) or Linux shared object libraries (.so).
- Additional FMU data like tables or maps, in specific file formats (MODELISAR and Modelica Association, 2014).

A schematic view of an FMU can be seen in Figure 3.8.

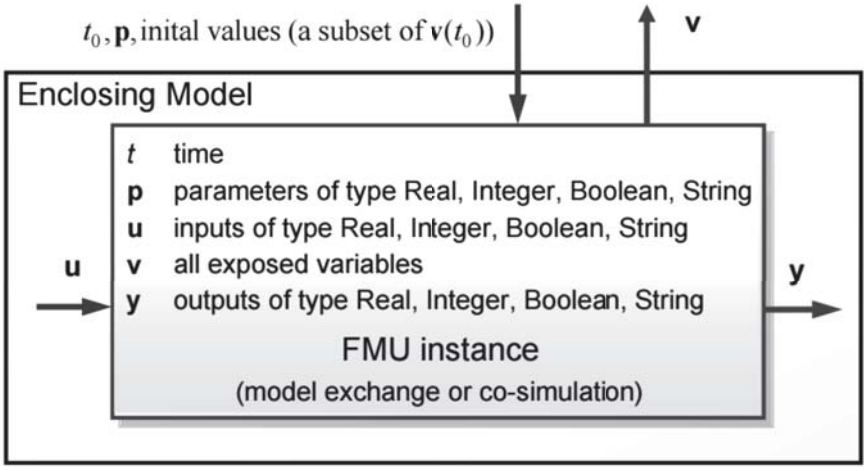


Figure 3.8: Data flow between the environment and an FMU (MODELISAR and Modelica Association, 2014).

The FMI 2.0 standard consists of two main parts:

- The **FMI for Model Exchange** interface defines an interface to the model of a dynamic system described by equations (differential, algebraic and discrete). The interface is designed to allow the description of large models. Figure 3.9a represents this scenario (MODELISAR and Modelica Association, 2014).
- The **FMI for Co-Simulation** interface is designed both for the coupling of simulation tools (simulator coupling, tool coupling) and coupling with subsystem models (MODELISAR and Modelica Association, 2014). The intention is to couple two or more models with solvers in a co-simulation environment. The data exchange between subsystems is restricted to discrete communication points. In the time between two communication points, the subsystems are solved independently from each other by their individual solver. Master algorithms control the data exchange between subsystems and

the synchronization of all slave simulation solvers (Blochwitz et al., 2012). Figure 3.9b represent this scenario.

(a) FMI model exchange diagram (MODELISAR and Modelica Association, 2014)

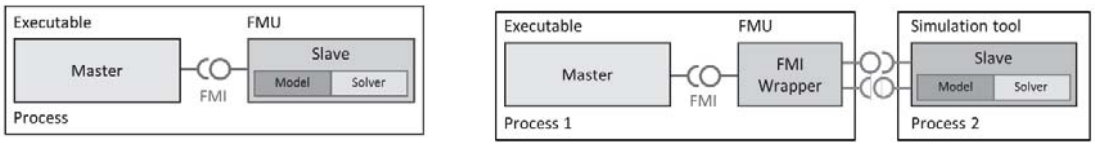
(b) FMI co-simulation diagram (MODELISAR and Modelica Association, 2014)

Figure 3.9: FMI simulation standards

From these two, the most important, regarding this work, is obviously the **FMI for Co-Simulation** interface, as it is a industry standard for Co-Simulation environments.

3.5.2 FMI for Co-Simulation

Co-Simulation is a rather general approach to the simulation of coupled technical systems and coupled physical phenomena in engineering with focus on in-stationary (time-dependent) problems (MODELISAR and Modelica Association, 2014). The FMI for Co-Simulation is designed both for coupling with subsystem models which have been exported by their simulators together with its solvers as runnable code (Figure 3.10a) and for coupling of simulation tools (Figures 3.10b) (MODELISAR and Modelica Association, 2014).



(a) Stand alone co-simulation (MODELISAR and Modelica Association, 2014)

(b) Tool coupling co-simulation (MODELISAR and Modelica Association, 2014)

Figure 3.10: FMI for Co-Simulation schemas.

Usually, a tool coupling based co-simulation is implemented on distributed hardware with subsystems being handled by different computers with maybe different OSs. The data exchange and communication between the subsystems is typically done via network, using **M**essage **P**assing **I**nterface (MPI) or **T**ransmission

Control **P**rotocol/**I**nternet **P**rotocol (TCP/IP). The definition of this communication layer is not part of the FMI standard, but co-simulation scenarios of this kind can stil be implemented using FMI, with the master having to implement the communication layer, as can be seen in Figure 3.11 (MODELISAR and Modelica Association, 2014).

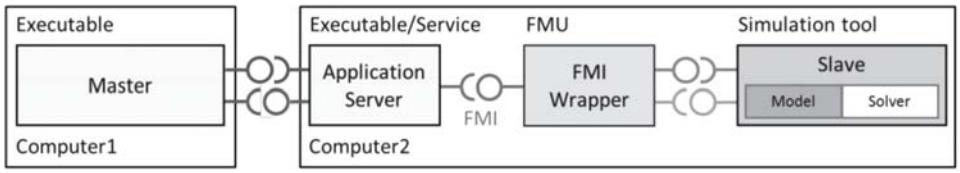


Figure 3.11: Distributed tool coupling co-simulation infrastructure (MODELISAR and Modelica Association, 2014)

In co-simulation stand alone, an FMU contains not only a model, but also solver code exported by another simulation tool to solve the model during simulation. Figure 3.12 represents a co-simulation slave FMU, which contains both model and solver.

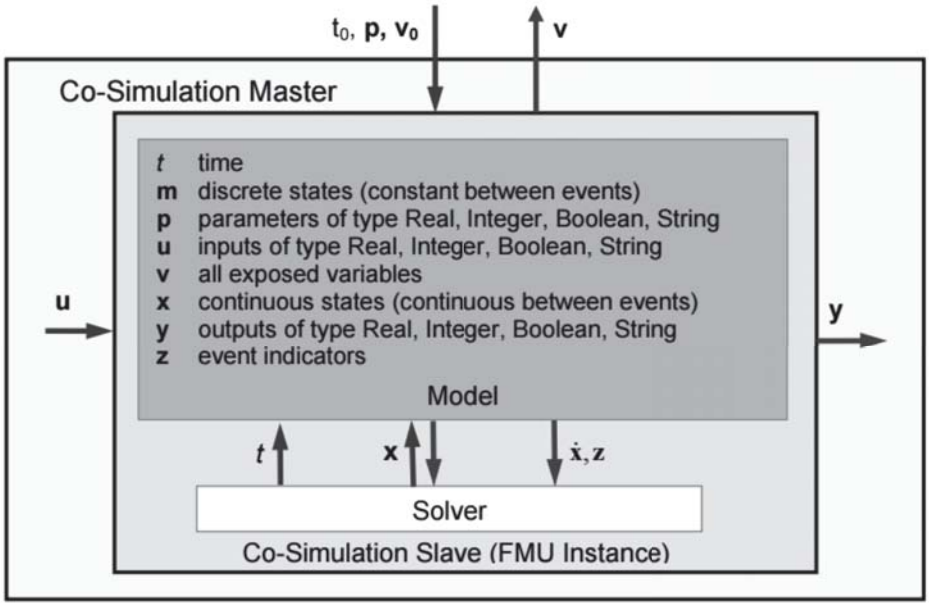


Figure 3.12: Data flow at communication points for Co-Simulation Master FMU (MODELISAR and Modelica Association, 2014)

3.5.3 FMI Library

The **F**unctional **M**ock-up **I**nterface **L**ibrary (FMIL) is a software package written in C that enables integration of FMUs import in applications. FMIL is an

independent open-source implementation of the FMI standard, presented above. The library provides a C API for interacting with all parts of FMUs, and that includes unzipping, loading of shared object files contained in FMUs as well as parsing of XML model metadata files. The user is thereby relieved from managing the details of FMU interaction, which significantly reduce the time required to implement FMU import capabilities (Modelon, 2016).

This library is suitable in contexts where FMUs need to be integrated in existing applications or in custom software projects. Its' key features include:

- Full support for FMIs Model Exchange and Co-simulation 2.0
- Full support for FMIs Model Exchange and Co-simulation 1.0
- A unified C API encapsulating all parts of the FMU interaction
- Build system based on CMake, enabling generation of native build scripts (Microsoft Visual C++ on Windows, **G**NU **C**ompiler **C**ollection (GCC) on Linux and Mac)
- Extensive API documentation in **H**yper**T**ext **M**arkup **L**anguage (HTML) format (Modelon, 2016)

The FMIL is also the basis for the official FMU Compliance Checker, which is a free software provided by the Modelica Association, implemented by Modelon AB to check a given FMU compliance with the FMI standard. The FMU Compliance Checker relies on the FMI Library for loading and interacting with FMUs (Modelon, 2016).

The FMIL is licensed under the BSD license, and as such the source code is available for download from the JModelica.org Subversion server, at `https://svn.jmodelica.org/FMILibrary/trunk` (Modelon, 2016).

3.6 Previous Work vs. Developed Work

As already mentioned, this project follows Naia (2015) work, and some minor changes were made to the design flow adopted there. The tools and interfaces used to co-simulate hybrid embedded systems in that previous work were also modified, with new domain specific tools being added. In this section, the main changes in what comes to these topics are presented.

3.6.1 Design Flow Changes

As for the first stage of validation, which occurs during system modeling and consists of validating the software application to the desired system, only the inputs and outputs were changed. During this phase, the application is running directly on the host OS, and will now be connected to PSIM, the power electronics simulation tool. This opens up the possibility to replace the previously used input/output from text files with data from the simulation. After the system has been modeled, the software application should undergo parallelization, turning it into a multi-threaded software application.

When the developed application behaves as desired, the validation advances to the next stage, with the software application being profiled, in order to find the threads that should be migrated to hardware. These threads will be replaced with delegate threads, and C/C++ behavioral models will be used.

After the hardware behavior is validated, the target machine is emulated using QEMU and the hardware IP's are simulated using an RTL simulator, e.g. Vivado Simulator, that will communicate with QEMU. The hardware-accelerated software application now runs on the QEMU emulated Linux-based target machine, with the Vivado Simulator simulating the RTL designs of migrated IP's, and PSIM performing the same task as in the first stage of validation. Both the Vivado Simulator and PSIM are coupled together with QEMU using a network interface developed by Naia (2015).

3.6.2 Verilog PLI vs. SystemVerilog DPI

The adoption of the SystemVerilog DPI as the (System)Verilog simulation interface over the Verilog PLI, which was the interface used by Naia (2015), opened new doors when it comes to integrating SystemVerilog code with C code. By using a simple DPI import declaration, a C function can be made to look as if it were a native SystemVerilog function: it can be called directly from any place a native SystemVerilog function can be called, SystemVerilog logic values can be passed directly to the C function as inputs, and C function returns or output arguments can be passed directly back to SystemVerilog. The DPI eliminates the Verilog PLI overhead of creating system task/function names and indirectly passing values in and out of C functions through complex PLI libraries (Sutherland, 2004).

The simple and direct nature of the SystemVerilog DPI makes it ideal for calling functions from standard C libraries, such as the C math library, or from user-defined libraries and is the ideal interface to use when the C function is working with data exchanged directly with SystemVerilog through function arguments and return values (Sutherland, 2004).

However, the DPI does not provide direct access to the internals of a simulation data structure, limiting its capabilites in comparison to the Verilog PLI. It is important to note, though, that many of these limitations of the SystemVerilog DPI can be overcome by using the DPI and the Verilog PLI together. A DPI based application, if imported as a content task or context function, can then call functions from the Verilog PLI libraries. In this way, the user can have the simplicity of the DPI import mechanism, and still have access to some aspects of the simulation data structure (Sutherland, 2004).

Still, we cannot consider the DPI as being a replacement for the Verilog PLI, because the PLI has full access to the internal simulation data structure, whereas a DPI based application, even when calling functions from the PLI libraries, does not have access to the full simulation data structure. Also, DPI-based applications cannot directly synchronize to simulation activity. Due to this fact, the use of the PLI is required for co-simulation environments that need to synchronize event scheduling with the Verilog simulator (Sutherland, 2004).

Although, when the time synchronization is not imperative, and there is no need to access the internal simulation data structures, the DPI is a very good alternative to the PLI. The DPI is much easier to use, and unlike the PLI isn't difficult to learn. Also, when the DPI statements are done correctly, using the *pure* and *context* keywords, the performance of the simulation can be greatly optimized.

As already mentioned, the SystemVerilog DPI is the simulation interface used throughout in this dissertation, as it is a faster alternative to the PLI when developing the simulation extensions needed to support the co-simulation approaches with QEMU.

3.6.3 Modelsim vs Vivado Design Suite

In his work, Naia (2015) used Modelsim by Mentor Graphics, but as already mentioned in this document, the chosen RTL simulator was the Vivado Simulator,

which is part of the Vivado Design Suite. This software suite developed by Xilinx enables developers to use a single IDE to synthesize their designs, perform timing analysis, examine RTL diagrams and simulate design's reaction to different stimuli.

This choice was not only based in the *extra* tools provided by Xilinx that allow for an easier compilation and integration of DPI applications, but also in the authors' previous experience with the Vivado Design Suite tools.

As mentioned above, in section 3.3.2, the Vivado Design Suite includes tools to generate header files based on the SystemVerilog design files, namely *xelab*, Xilinx's *elaborator*. The generated files can be included in the C Layer of the DPI application in order to aid the developer by giving C compilers the ability to warn the user in case he is not complying with the headers provided by the SystemVerilog compiler.

The Vivado IDE allows the developer to use a single environment to write, synthesize and simulate the IP's in order to validate their behavior, and only then use them as hardware accelerators together with the hardware-accelerated software application.

Looking forward into possible improvements to this work, the Vivado Design Suite also contains an high-level synthesis tool, Vivado HLS, that is shipped with a toolchain that converts C/C++/SystemC code into programmable logic. This feature can be used to generate the hardware accelerators using the previously validated C/C++ code from the developed software-only application, being a faster alternative to the development of the hardware accelerators "from scratch". Vivado HLS is widely reviewed to increase developer productivity, supporting C++ classes, templates, functions and operator overloading.

3.6.4 Power Electronics Domain Simulation

As above-stated, the design flow is now tightly coupled with PSIM, a power electronics circuit simulator. PSIMs' DLL Block Interface (described above in subsection 3.4.2) opens up the possibility to use data from a simulated circuit as the input for the software-only application, along with interfacing with a waveform viewer to show its' outputs.

Regarding the developed integrated co-simulation environment, it now includes an

embedded platform simulation (QEMU), an RTL simulator (Vivado Simulator) and a power electronics simulator (PSIM), which is also useful to verify the results of the scenario under test, due to its integrated waveform viewer (SimView).

Chapter 4

Co-Simulation Extensions Design

In this chapter, the design flow adopted in this project is presented, along with the developed extensions and libraries that allowed the creation of an integrated co-simulation environment. Full System Co-Simulation allows reducing the time spent on designing and debugging the system, along with the computation offloading and validation phases, achieving improvements in the design flow.

In order to validate and stimulate the development of such extensions, a multi-threaded hardware accelerated Linux-based programming model was co-designed, and an overview diagram is presented in Figure 4.1.

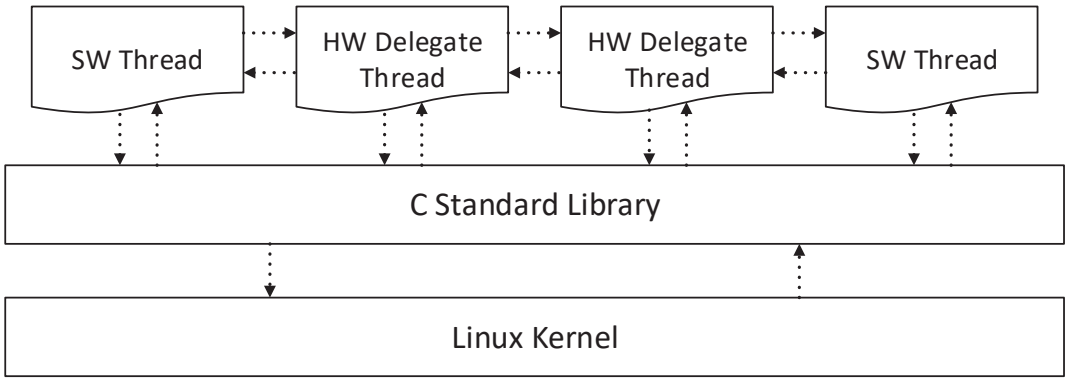


Figure 4.1: Co-designed Linux-based programming model overview

The presented co-designed Linux-based programming model is composed by a series of threads that can either be implemented as software threads or as hardware delegate threads. Software threads are the most "common" threads, and take care of their own data processing. Hardware delegate threads entrust their data pro-

cessing responsibilities to hardware accelerators (using device drivers for communication), and act as software representations for hardware accelerators, providing interfaces to the custom developed hardware accelerators .

4.1 System Co-Design Flow

The hardware software co-design scenarios presented in sections 3.1.3 and 3.1.4 were the foundations for the design flow adopted in this dissertation, with QEMU being used as the software simulator for the target platform, Vivado Simulator (XSim) as the HDL simulator and PSIM as the power electronics simulator.

The co-design flow proposed in this dissertation is directed to power electronics application scenarios, since they usually require fast-responding, deterministic controllers with real-time constraints, as already mentioned in the document. To achieve this, hardware acceleration may be incorporated, making use of the hardware true parallel nature and offloading critical application kernels to custom hardware co-processors, as mentioned in section 2.3.

Figure 4.2 presents an overview diagram of such design flow, that follows the one presented in section 2.4.1 and is discussed next in this section, with an overview diagram for each phase being presented.

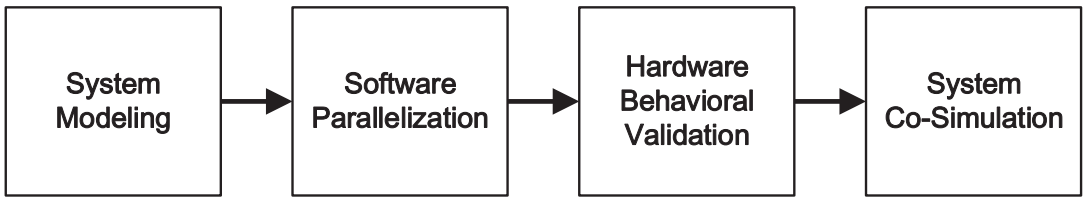


Figure 4.2: Design Flow Overview

4.1.1 System Modeling

The first stage of validation occurs during system modeling, and consists in validating the software application that corresponds to the desired system. During this phase, PSIM is used in order to model the power electronics hardware along with the desired controller, modeled in a DLL Block. The software application (.dll) runs directly on the host machine OS. An overview diagram for this phase of the design flow can be seen in Figure 4.3.

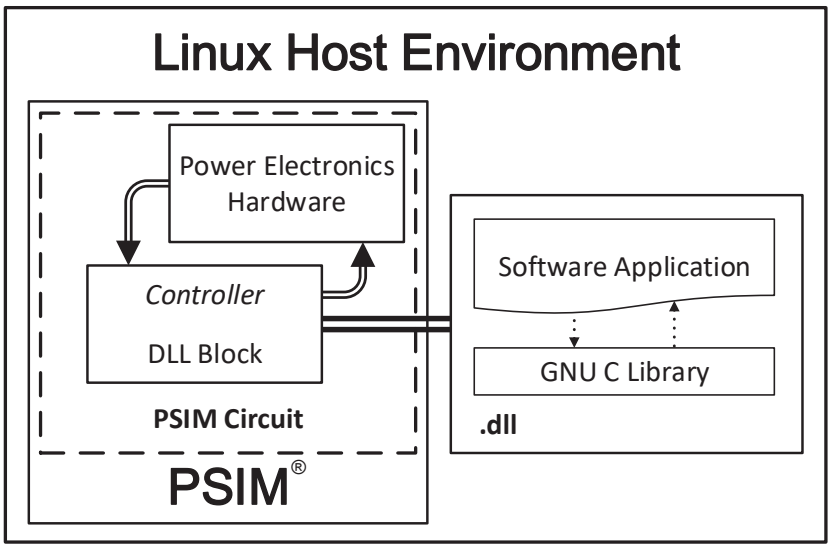


Figure 4.3: System Modeling Overview

4.1.2 Software Parallelization

As the software application becomes more and more complex, the system tasks must be identified in order to divide it into smaller parts. Then, the software will be parallelized by means of a multi-thread programming model, as mentioned in subsection 2.4.1.

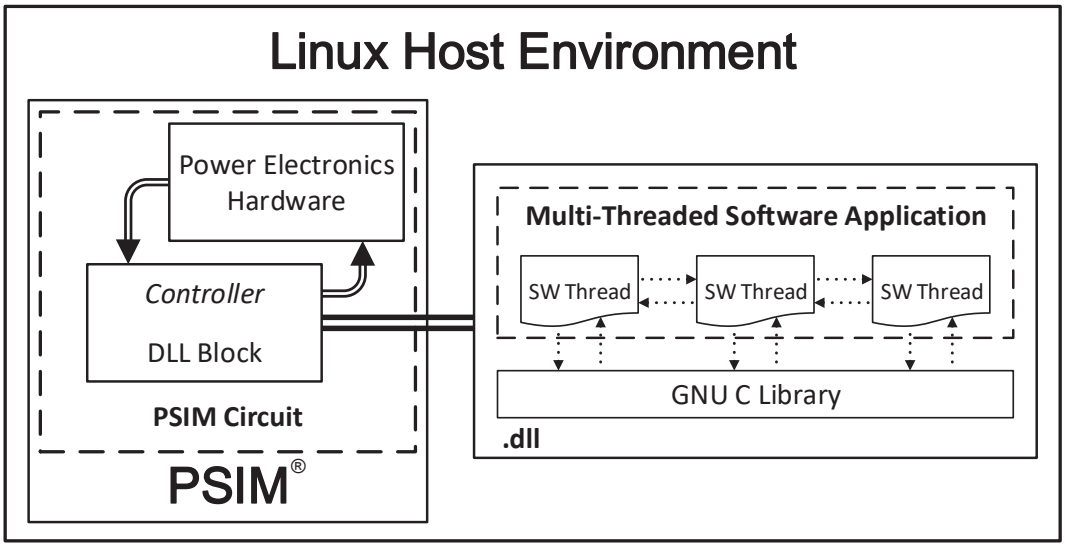


Figure 4.4: Software Parallelization Overview

In this step, the multiple task algorithms are assigned to different threads. Since

threads usually share data, synchronization mechanisms are mandatory to avoid race conditions, and are usually provided by the multi-threading API. The overview diagram for this phase of the design flow is presented in Figure 4.4.

4.1.3 Hardware Behavioral Validation

Later, when the software application fits the system's desired behavior, it must be profiled, in order to find out the most *CPU-hungry* threads, as they are the candidates for hardware acceleration. Before developing the accelerators, the hardware's behavior can be validated using the QEMU Plugin Extension 3.2.1, with behavioral C/C++ models for the migrated hardware being integrated into the emulated machine.

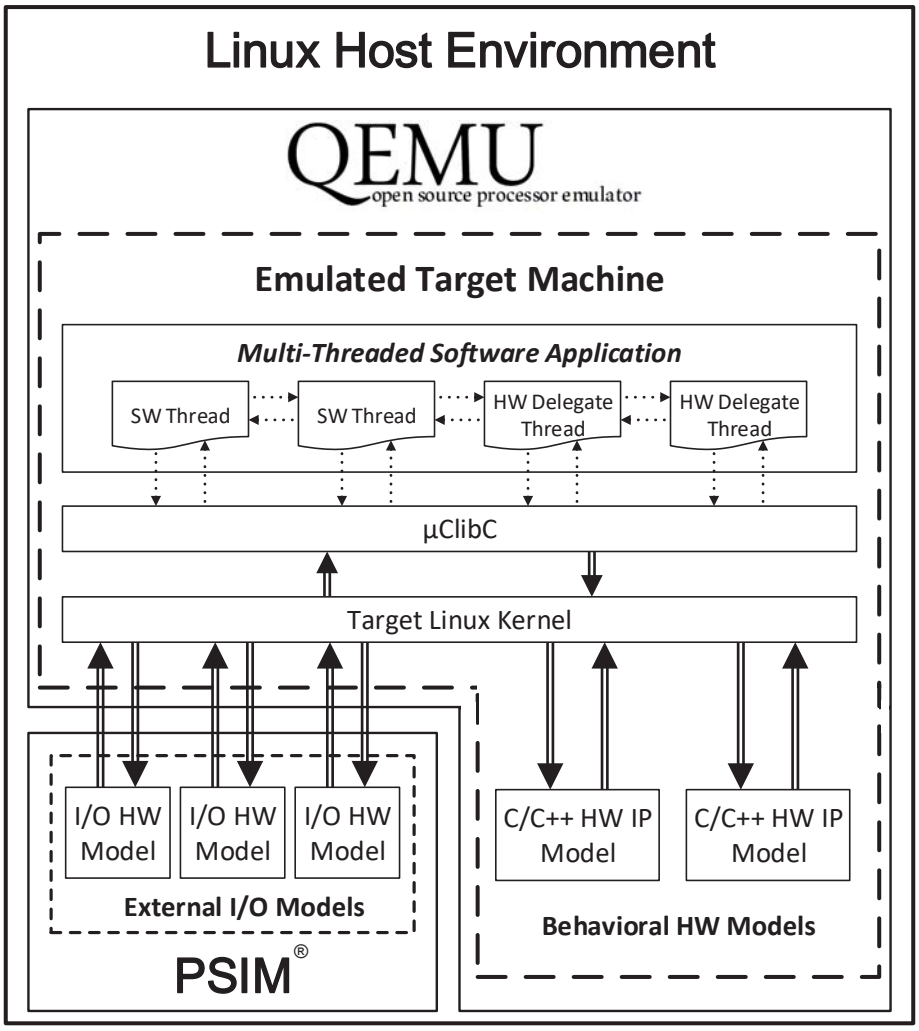


Figure 4.5: Hardware Behavioral Validation Overview

This allows the developer to try out behavioral hardware models before developing the HDL IP's, along with developing and validating their respective device drivers.

In this validation phase, the target machine is emulated using QEMU, and the hardware-software application will run over the Target Linux Kernel. The software application must be modified, with delegate threads replacing the software threads identified as candidates for hardware acceleration. The hardware delegate threads will entrust their processing to the C/C++ behavioral models. The power electronics circuit being simulated on PSIM remains, and its' interface with QEMU is now represented by the Input/Output hardware models, that act as the stimuli for the system. The overview diagram for the above-mentioned is presented above, in the Figure 4.5.

In order to establish the connection between QEMU and PSIM, a library based on the QEMU External Model Extension developed by Naia (2015) was developed.

4.1.4 System Co-Simulation

Since the hardware's behavior and the device drivers were already validated in the last phase, the hardware accelerators can now be developed and tested, with the RLT simulation being performed on Vivado Simulator and the simulated IP's replacing the previously used C/C++ behavioral models. The hardware delegate threads remain, and will now entrust their processing to the external HW models present in the external HDL simulator.

In order to do so, a VPI dynamic library for Verilog simulators that support this validation approach was developed by Naia (2015) in his work, but as mentioned in section 3.3 the SystemVerilog DPI was the chosen interface for this work. Given this, a DPI library for SystemVerilog simulators that makes use of the above-mentioned QEMU External Model Extension was developed.

After the hardware IP's have been developed, the system can be simulated in all it's domains: embedded software, hardware accelerators and electrical circuit. This is possible by using QEMU along with the simulation extensions developed during this work and the ones developed by Naia (2015). An overview diagram for the last step of the design flow is presented in Figure 4.6.

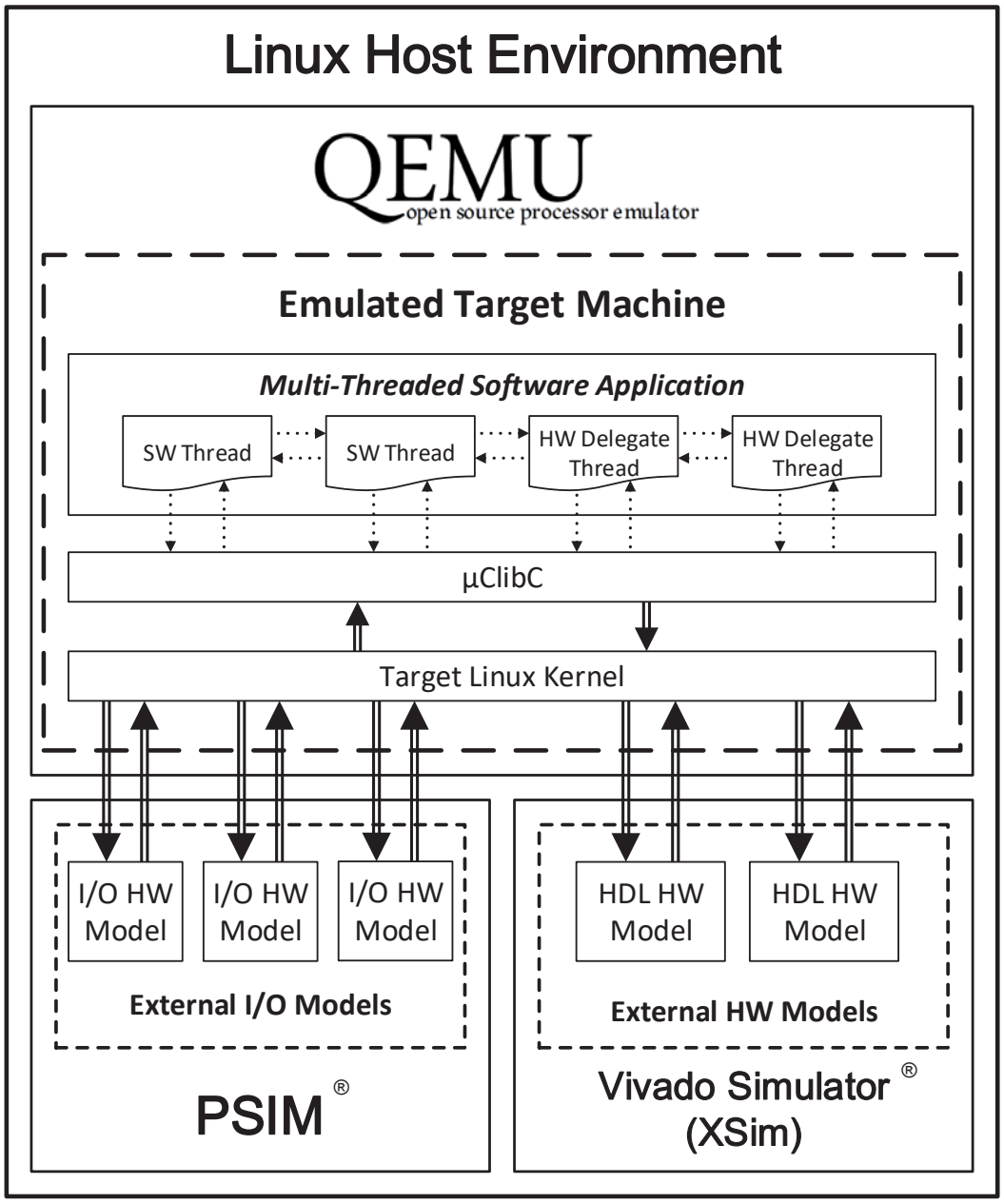


Figure 4.6: System Co-Simulation Overview

Co-Simulation Extensions

This chapter is then divided into three sections, with the next two sections describing the developed simulation extensions, with focus being placed on how both were implemented and how to use them.

Firstly, the already mentioned QEMU Co-simulation DPI library will be presented.

This dynamic library allows SystemVerilog simulators that support DPI to simulate hardware IPs integrated into QEMU's emulated machines, and is based on the QEMU External Model Extension (3.2.2).

Lastly, the QEMU Co-Simulation PSIM library is described. This library is a DLL that allows PSIM to interact with QEMU's emulated machines. As already mentioned, this library also uses the QEMU External Model Extension (3.2.2).

4.2 QEMU Co-Simulation DPI Library

As seen in section 2.3, when an embedded system demands strict time constraints, the software resources may be insufficient to meet them. By offloading critical tasks to hardware co-processors, the CPUs load will be "lighter", allowing the fulfillment of the system's timing constraints.

Typically, these co-processors and peripherals are developed using hardware description languages and validated by HDL simulators using testbenches that contain non-synthesizable constructs of the language. When using this kind of validation, the interactions between the software and the hardware co-processors are not taken into account.

In order to ensure the integration of both domains is on point, validating both device drivers and hardware acceleration domain components is needed. To do so, a DPI library was developed in the context of this dissertation, implementing an interface with QEMU based on the QEMU External Tool Library (3.2.2). This library makes possible the interaction between HDL simulators that support SystemVerilogs' DPI (presented in subsection 3.3.2) and QEMU, thus enabling the validation of the interactions between the software application and the developed hardware co-processors.

4.2.1 Library Overview

In Figure 4.7, a programming model corresponding to a hardware accelerated embedded Linux system emulated on QEMU with the hardware co-processors simulated on Xilinx's Vivado Simulator is presented. As seen in the figure, the hardware accelerated embedded application is composed by software threads and hardware

delegate threads. These hardware delegate threads delegate their processing tasks to hardware accelerators via device driver system calls.

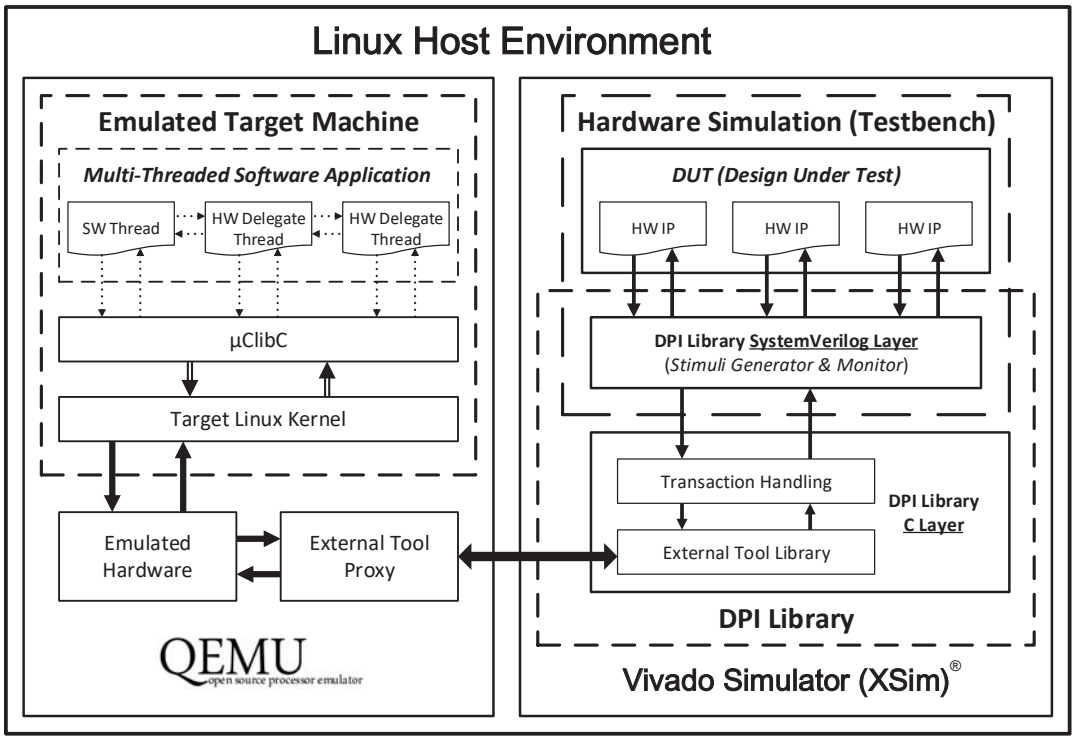


Figure 4.7: QEMU Co-Simulation DPI library overview

The DPI library allows a QEMU machine simulation to interact with the hardware accelerators being simulated in Vivado Simulator. This interaction is based on the QEMU External Tool Library (3.2.2), that will take care of the data exchange between the DPI Library and QEMU. The DPI library acts as the bridge between the hardware simulation and the External Tool Library.

Library Initialization Overview

Figure 4.8 presents a sequence diagram of a hardware accelerator from Vivado Simulator being registered as an external model in QEMU. As mentioned in section 3.2.2, QEMU is blocked at start-up, until all the simulation tools connect. In this case, let's assume Vivado Simulator is the only tool being used, so after the accelerator(s) are registered and the DPI saves that information, it will request the connection with QEMU, using the DPI library interface. Along with the model's information, tool information is also provided to the External Tool Library. Before

connecting with QEMU, the DPI library will register the transaction routines, and only then the connection is established, and the QEMU emulation may resume.

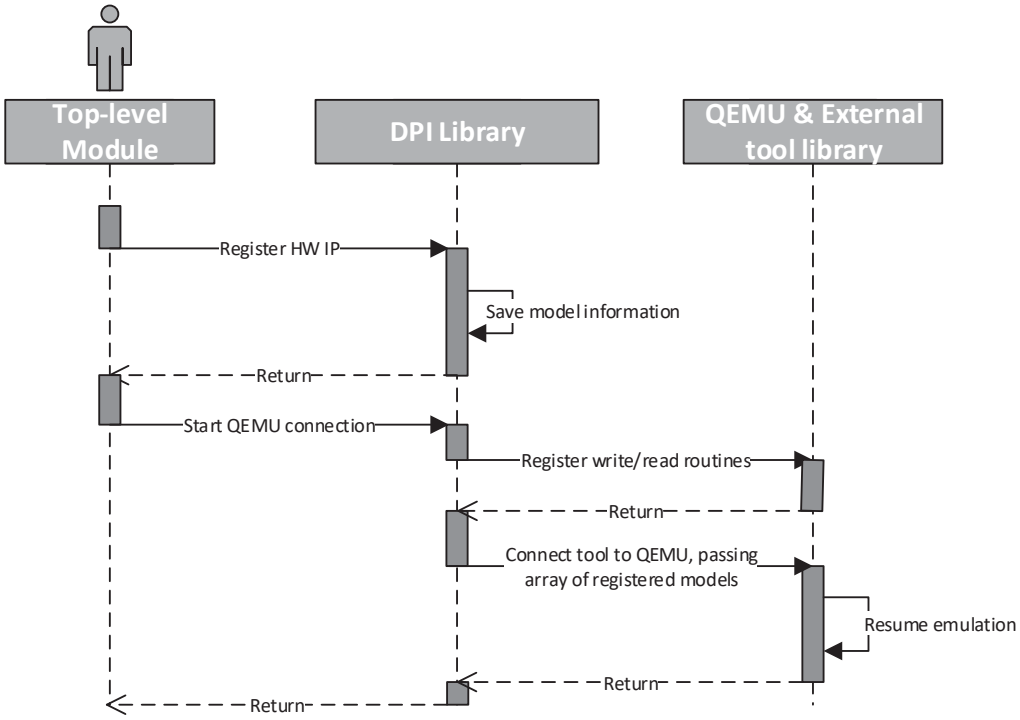


Figure 4.8: DPI library initialization sequence diagram

Library Transaction Handling Overview

Figure 4.9 presents a sequence diagram of a hardware transaction being issued by QEMU and resolved by Vivado Simulator. When a transaction happens in the QEMU emulation context, the External Tool Library will call the corresponding registered transaction read or write function on the DPI library, who will then pass the transaction information to the corresponding Vivado Simulator registered model.

After the write/read request has been completed by the Vivado Simulator, the DPI library will pass the data received back to QEMU, via the External Tool Library. QEMU will then resume its emulation, which is halted until the hardware transaction is completed.

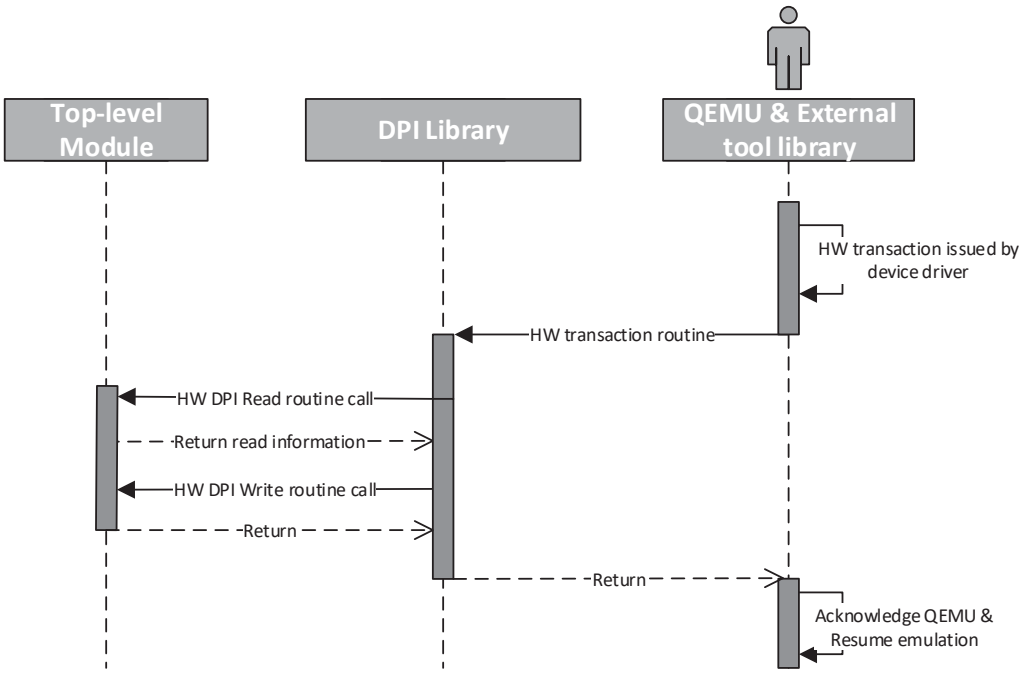


Figure 4.9: DPI library transaction sequence diagram

Library Interrupt Handling Overview

Figure 4.10 presents a sequence diagram of an interrupt being caught on Vivado Simulator, and passed into the QEMU emulation, so the respective device driver IRQ can be called. When an interrupt from an HW accelerator is triggered in Vivado Simulator, the DPI library raise/lower interrupt functions will be called. The interrupt information will then be passed to the QEMU emulation by the External Tool Library, so it can be handled by the emulated machine.

4.2.2 Library API

In order to allow the co-simulation between QEMU and SystemVerilog simulations, the DPI library provides a set of functions that can be used by any HDL simulator that supports the SystemVerilog DPI, such as the Vivado Simulator.

These functions must be *imported* using the *DPI import statement*, described in section 3.3.2, so the simulator can link them to the library. The import statements should be made in the top-level module of the simulation. Appendix B.4 contains

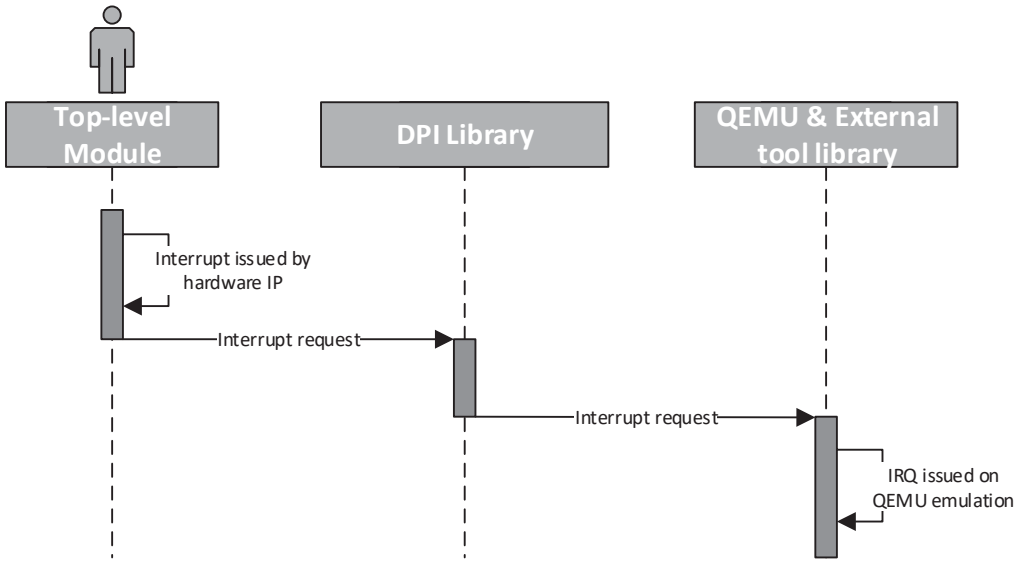


Figure 4.10: DPI library interrupt sequence diagram

a SystemVerilog top-level module with these statements.

Along with the *imported* functions, the HDL top-level module should also contain the *DPI export statements* (also presented in section 3.3.2), for the *dpi_write* and *dpi_read* functions, that are called every time a write or read transaction is received by the DPI Library, sent from the External Tool Library. Appendix B.4 also contains these statements.

Table 4.1 lists the above-mentioned functions, if they are *imported* or *exported* functions, and the module they should be used on.

In order to correctly use the DPI API, the top-level module must contain a set of standard signals, and every IP wrapper module must contain a set of parameters that will be used to register the model in QEMU.

Top-Level Module

As shown in Appendix B.4 , the top-level module must contain the following standard signals:

- clk

Table 4.1: QEMU Co-Simulation DPI library API

API	Description	DPI Statement Type	Module
qemu_register_ip	Register SysVerilog module containing external model	Import	Top-level
qemu_connect	Connect to QEMU using Ext. Tool Library and start co-simulation	Import	Top-level
qemu_raise_int	Raise an interrupt in QEMU	Import	IP Wrap
qemu_lower_int	Lower an interrupt in QEMU	Import	IP Wrap
dpi_write	Request write transaction on HDL simulator	Export	Top-level
dpi_read	Request read transaction into HDL simulator	Export	Top-level
dpi_print	Print information on HDL simulator console	Export	Top-level
dpi_stop_sim	Stop HDL simulation	Export	Top-level

This represents the simulation's clock and is common to all modules.

- rst

  This represents the simulation's reset signal and is also common to all modules

- addr

  This represents the address signal for the IPs, and has a maximum width of 64 bits. When transactions occur, the exported functions defined in the top-level module will write the value received from the DPI library into the *addr* signal.

- din

  This signal represents the input data, and is used only when write transactions occur. The exported *write* function defined in the top-level module will write the value received from the DPI library into the *din* signal. Its width is parametrized, in order to match the machine word bit length.

- write

  This signal is only used to indicate that the transaction in course is either a write or read transaction. *write* is 1 for a write transaction, and 0 for a read transaction.

- dout

  This signal is a tristate signal, in order to be shared between IPs, and is pulled down by default. It is only used when IPs need to output their values in read transactions.

- ready

  This is an acknowledgment tristate signal, also pulled down by default. It is used by IPs to acknowledge finished transactions.

As above-mentioned, these signals must be present in the top-level module, in order to achieve consistency when using this library. The IP wrapper modules will share these signals, as they are passed to them when being instantiated.

Below is shown a template of an IP wrapper module instantiation:

```
IPWrap #(.WORD(WORD), .MAPPED_REG_SZ(MAPPED_REG_SZ))
        IP_Mod  (.addr(addr),
                 .din(din),
                 .offset(offs),
                 .dout(dout),
                 .write(write),
                 .busy(busy),
                 .ready(ready),
                 .clk(clk),
                 .rst(rst)
                 );
```

IP Wrapper Modules

Regarding the IP wrapper modules, a set of parameters must be present in every one of them. These parameters are used when registering the external models on QEMU, as they are input parameters for the imported function *qemu_register_ip*.

Assuming an IP Wrapper module called *IP_Mod* was instantiated in the top-level module, an example of a call from Vivado Simulator to register that module is as follows:

```
qemu_register_ip
(
    IP_Mod.NAME,
    IP_Mod.MEMORY_MAPPED_ADDRESS,
    IP_Mod.MAPPED_AREA_SIZE,
    IP_Mod.INTERRUPTS_SIZE,
    IP_Mod.INTERRUPTS
);
```

As can be seen in Appendix B.5, every wrapper module must contain the following standard parameters, with examples being presented for each one:

- NAME

  SystemVerilog parameter of type *string*, can be consulted from QEMU Monitor as a device property.

  ```
  string NAME = "example_HW_IP";
  ```

- MEMORY_MAPPED_ADDRESS

  SystemVerilog parameter of type *longint*, corresponding to the base address of the IP, has a maximum width of 64 bits.

  ```
  longint MEMORY_MAPPED_ADDRESS = 64'h54013000;
  ```

- MAPPED_AREA_SIZE

  SystemVerilog parameter of type *int*, corresponding to the block size of the local memory in bytes, has a maximum width of 32 bits.

  ```
  int MAPPED_AREA_SIZE = 32'd28;
  ```

- INTERRUPTS_SIZE;

  SystemVerilog parameter of type *int*, reflecting the number of interrupts the model can use. QEMU uses this parameter in order to allocate IRQs in the machine emulation.

  ```
  int INTERRUPTS_SIZE = 32'd3;
  ```

- INTERRUPTS

  SystemVerilog parameter of type *string*, containing the interrupt numbers used by the requested IRQs registered in QEMU, ordered and separated by an underscore (_). It is mandatory to provide an interrupt number for each one of the requested IRQs. This parameter should have the following format:

  "INT1_INT2_INT3_INTN_"

  ```
  string INTERRUPTS = "8_21_16_";
  ```

As mentioned in the last subsection, after the QEMU emulation has been started, and is waiting for the simulation tools to connect, the top-level module must call the imported *qemu_register_ip* function for every wrapper module containing IPs that are going to be used as external models, as seen above. Only then the imported *qemu_connect* function may be called, in order to establish the connection between QEMU and Vivado Simulator, specifically its external models.

4.2.3 C Layer Transaction Handling

As above-mentioned, the QEMU Co-Simulation DPI library uses the External Tool Library (3.2.2) along with the exported functions (*dpi_read* and *dpi_write*) from the top-level module in order implement transactions.

The transaction routines registered on the External Tool Library by the DPI Library will receive the transaction requests coming from the device drivers being emulated on QEMU. In order to get the transaction information to the respective external model, DPI exported routines must be used.

DPI Implementation Issues

Given that the use of DPI exported tasks (tasks can consume simulation time, while functions can't, as mentioned in subsection 3.3.2) is not yet supported by the available HDL simulators, only DPI exported functions can be used.

This, combined with the fact that exported SystemVerilog routines can only be called from C functions that have been imported as context functions/tasks, as seen

in subsection 3.3.2, denies the direct call from the registered transaction routines from the DPI Library to the Vivado Simulator.

The above-mentioned is true because the transactions are sent into the DPI Library via callbacks that were previously registered, and their scope is not local to the Vivado Simulator DPI context.

Figure 4.11 represents a simple sequence diagram of the above-mentioned: when the functions calls are made in the same context, they are allowed, and when the call comes from an external scope, the DPI exported function calls will not work.

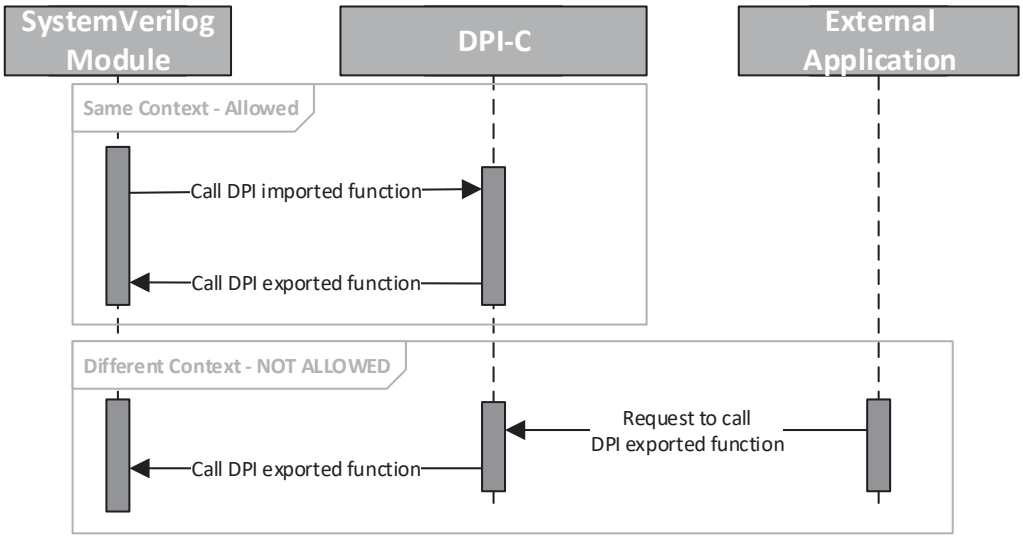


Figure 4.11: Sequence diagram showing different DPI contexts

Transactions Redirection Mechanisms

This makes it necessary to implement mechanisms in order to redirect the transactions from QEMU to the Vivado Simulator: a DPI imported function called *trans_handle* is called at each positive edge of the top-level signal *clk*, representing the clock of the system, checking for any pending write/read transactions on the DPI layer.

Calling the DPI C imported function *trans_handle* allows the DPI library to call SystemVerilog exported functions, such as *dpi_read* and *dpi_write*, that will manipulate the signals in the RTL simulation in order to effectively complete the

transactions. Since the calls to these functions will be made from inside the *trans_handle* function, which is a DPI imported function, they will consequently have the same context as the Vivado Simulator.

A more accurate sequence diagram for this particular case, the QEMU Co-Simulation DPI library, can be seen in Figure 4.12.

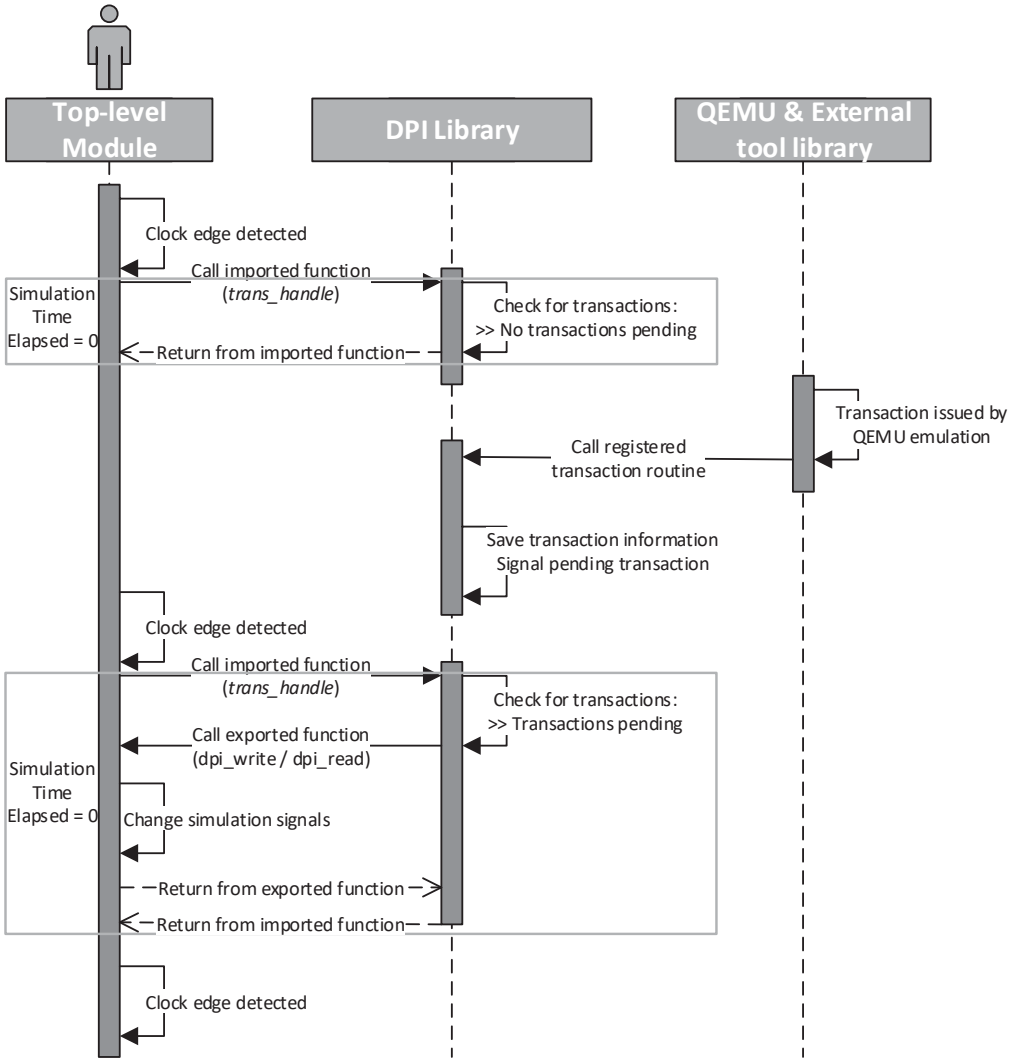


Figure 4.12: DPI Library transaction handling sequence diagram

An excerpt of code from the C Layer of the DPI Library can be found in Appendix B.7. Smaller excerpts will be presented as this mechanisms are described, in order to help the reader understand how they were implemented.

At each positive edge of the clock signal, the DPI Library will check for any pending

transactions. In case there aren't any transactions pending, it will simply return. Note that this process consumes no simulation time in Vivado Simulator, as seen in section 3.3.2: "all imported functions shall complete their execution instantly and consume zero simulation time".

In case a transaction is sent from the QEMU emulation, via the External Tool Library, into the DPI Library registered slave transaction routines, that same routine will save the transaction data (model address and offset, value sent and message size) and set flags to signal the reception of a transmission, as shown in the following code excerpt.

```
1  /////     HW ACCESS ROUTINES    /////
2  uint64_t hw_write(...)
3  {
4     pthread_mutex_lock(&trans_write.trans_mutex);
5
6     trans_write.address = memory_mapped_address;
7     trans_write.offset = offset;
8     trans_write.value = value;
9     trans_write.size = size;
10    trans_write.state = 1;
11
12    pthread_mutex_unlock(&trans_write.trans_mutex);
13 }
14
15 uint64_t hw_read(...)
16 {
17    pthread_mutex_lock(&trans_read.trans_mutex);
18
19    trans_read.address = memory_mapped_address;
20    trans_read.offset = offset;
21    trans_read.size = size;
22    trans_read.state = 1;
23
24    pthread_mutex_unlock(&trans_read.trans_mutex);
25
26    *value = trans_read.value;
27 }
```

At the next positive edge of the clock, the DPI library will notice there are pending transactions. Then, depending on the type of the request (write or read transaction), the corresponding exported function will be called, namely *dpi_read* or *dpi_write*, with the previously saved transaction data being passed as arguments. These arguments are needed in order to change the simulation signals on the Vivado Simulator, and in case it's a read action, return the requested data. The implementation of such mechanism can be seen in the next code excerpt.

```
1  /////     TRANSACTION HANDLER     /////
2
3  void trans_handle()
4  {
5     if(trans_write.state)
6     {
7        trans_write.state = 0;
8
9        pthread_mutex_lock(&trans_write.trans_mutex);
10
11       dpi_write(trans_write.address, trans_write.offset,
             trans_write.value, trans_write.size);
12
13       pthread_mutex_unlock(&trans_write.trans_mutex);
14    }
15
16    if(trans_read.state)
17    {
18       trans_read.state = 0;
19
20       pthread_mutex_lock(&trans_read.trans_mutex);
21
22       trans_read.value = dpi_read(trans_read.address,
            trans_read.offset, trans_read.size);
23
24       pthread_mutex_unlock(&trans_read.trans_mutex);
25    }
26 }
```

After these actions have finished, the exported function will return, and then

the imported function will return. From the moment the *trans_handle* imported function is called until the moment it returns, the simulation time elapsed is also zero, due to the above-mentioned reason.

In order to prevent race conditions and maintain data/transaction integrity, mutual exclusion mechanisms are used by the DPI library.

Interrupt Handling

As mentioned in the last subsection, triggering interrupts in QEMU's emulated machine from the RTL simulation is supported, using a mechanism similar to the transaction handling one. When it comes to interrupt handling, there aren't any context issues, as in the transaction requests, because the interrupts will always be called from the HDL simulator to the QEMU emulation, only requiring the use of imported functions.

The DPI library only purpose will be to "redirect" the interrupt request, with the calls to the imported functions *qemu_raise_int* and *qemu_lower_int*. Both these functions have as their only arguments the external model base address, along with the respective interrupt number. These arguments will be used to match the previously registered base address and interrupt numbers.

The above-mentioned DPI Library functions only task is to call the *qemu_set_interrupt* function, implemented by the External Tool Library, that has as arguments the base address of the model, the interrupt number and a flag to signal if it is a raise interrupt (1) or a lower interrupt (0) request.

A sequence diagram of an interrupt request being handled by the QEMU Co-Simulation DPI library can be seen in Figure 4.13.

The time elapsed on the RTL simulation when handling an interrupt request is zero, as when handling transactions, since only DPI imported functions are used, and they "complete their execution instantly and consume zero simulation time".

4.3 QEMU Co-Simulation PSIM® Library

As mentioned in section 3.4, when building an integrated co-simulation environment, simulation tools from domains other than hardware acceleration may be

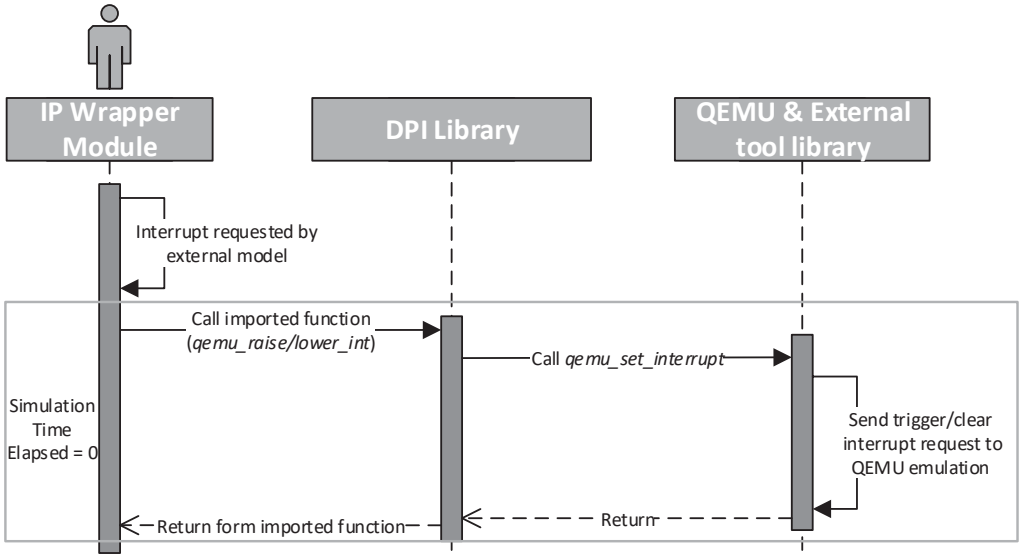


Figure 4.13: DPI Library interrupt handling sequence diagram

used, such as power electronics systems simulators.

A simplistic description for power electronics systems is that they are composed of the "power circuit" (semiconductors, passive elements, etc.), the controller, the sensors and actuators.

- The *power circuit* includes switching semiconductors, energy storing elements such as inductors and capacitors and other passive elements.
- The *controller* is almost always an embedded system software, and may belong to any of the embedded system types seen in section 2.1.2.
- The *sensors* and *actuators* are the systems' "interfaces" with the real world, therefore being essential.

Since most of the current power electronics systems use embedded systems software to perform their control algorithms, and many of them have real-time constraints, that takes us to this works' main topic: embedded system platforms with strict time constraints, leading to the offloading of critical tasks to hardware co-processors.

In order to simulate the interactions between the different components of power electronics system, the co-simulation environment must contain tools that allows

for precise simulation of the physical components of the system: the power circuit, the sensors and actuators.

As mentioned in subsection 3.4.2, PSIM® was the chosen tool in this work when it comes to power electronics systems simulation, mostly due to its external DLL block component that allows PSIM® to interact with other simulators or tools.

Given these facts, a library that allows connecting QEMU with PSIM® was developed in the context of this dissertation, again based on the QEMU External Tool Library (3.2.2). This library allows simulating the integration between the software application and the physical components of the system, namely the power system and the sensors/actuators.

When combined with the QEMU Co-Simulation DPI Library (4.2), this library allows the Co-Simulation Environment to simulate the integration between the software application, the hardware accelerators and the power electronics system.

4.3.1 Library Overview

Figure 4.14 presents an example of an embedded Linux system emulated on QEMU receiving stimulus from external models on PSIM®.

This figure is very similar to Figure 4.7, which is normal because both are based on the QEMU External Tool Library (3.2.2) and will communicate with QEMU using the same mechanisms.

The sensors and actuators of the system being simulated in PSIM® will act as inputs and outputs for the QEMU emulated machine software. In order to do this, hardware delegate threads will again be used, making use of device drivers system calls, in order to send/receive data to the PSIM® simulation.

Library Initialization Overview

Figure 4.15 presents a sequence diagram of a hardware model from PSIM® being registered as an external model in QEMU. As mentioned in section 3.2.2, QEMU is blocked at start-up, until all the simulation tools connect.

When the **Run Simulation** button is pressed on PSIM®, the *OPENSIMUSER* user-defined function described in subsection 3.4.2 will be called.

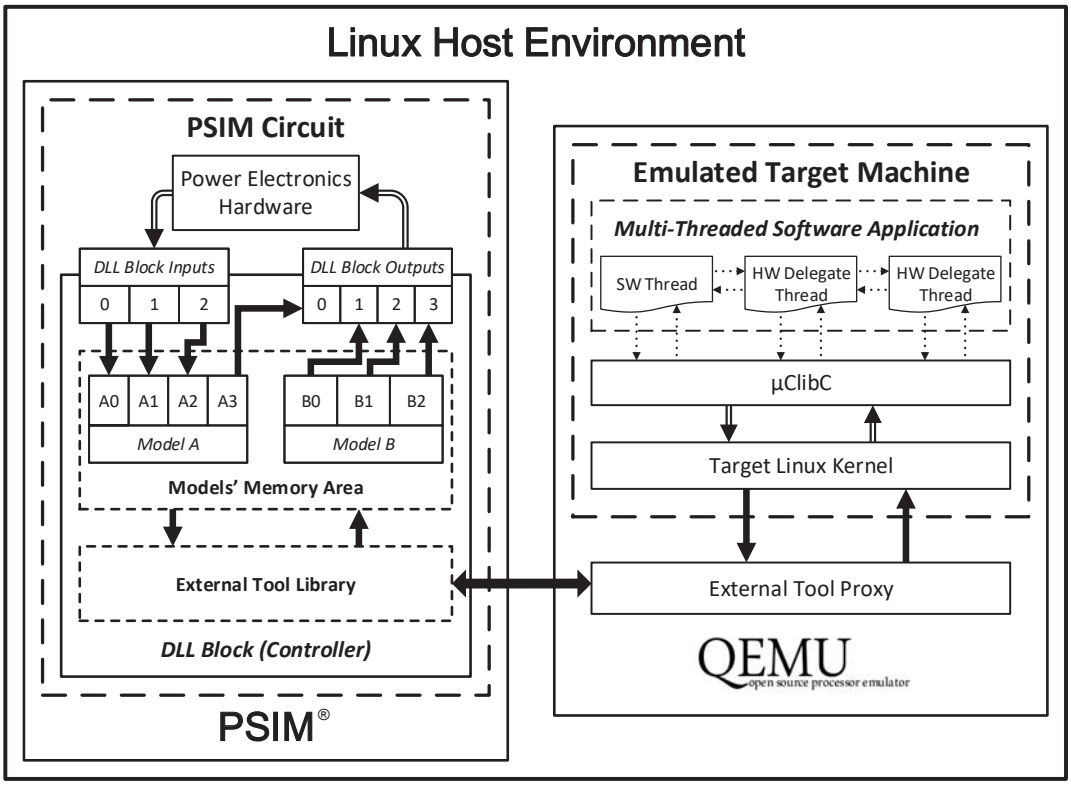


Figure 4.14: QEMU Co-Simulation PSIM® library overview

Assuming PSIM® is the only tool being used, this function will simply register the transaction routines and then request the connection with QEMU, passing the models' information along with the tools' information, which is also provided to the External Tool Library. After the connection has been successfully established, the QEMU emulation may resume.

Library Transaction Handling Overview

Figure 4.16 presents a sequence diagram of a transaction being issued by QEMU and resolved by the PSIM® DLL. When an hardware access happens in the QEMU emulation context on an device that is registered as a PSIM® external model, the External Tool Library will call the corresponding registered transaction function on the DLL. In case it is a read transaction, the transaction function will return the corresponding value. In case it is a write request, the transaction function will write the value to the models' memory mapped area.

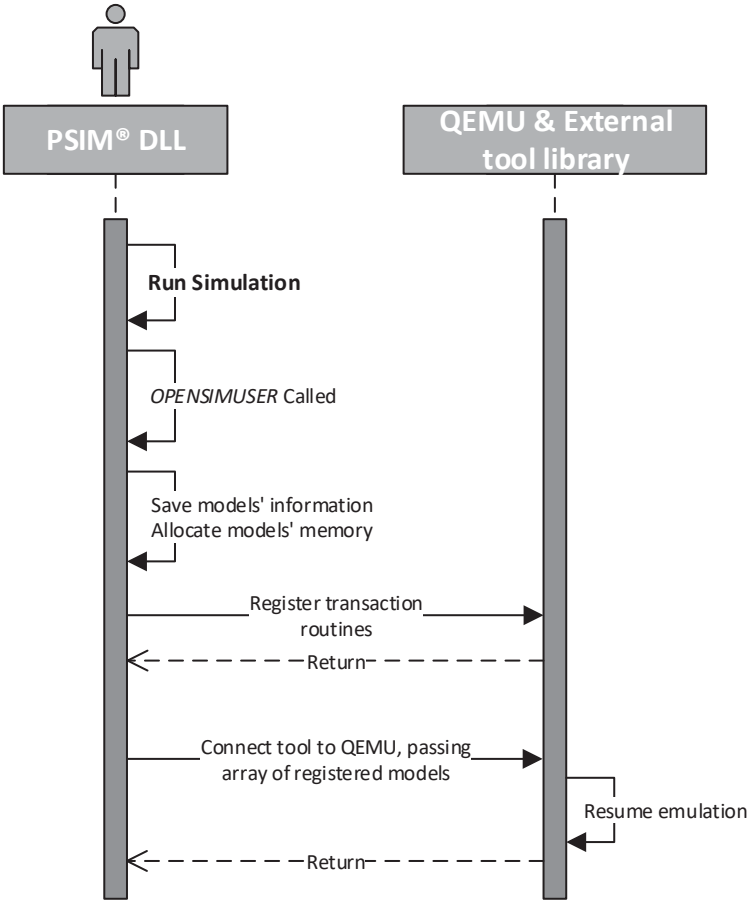


Figure 4.15: PSIM® DLL initialization sequence diagram

4.3.2 Library Structure

In order to allow the machine emulated on QEMU to interface with external models from PSIM®, a general DLL block (3.4.2) must be included in the PSIM® simulation.

As mentioned in subsection 3.4.2, the PSIM® DLL Blocks interface has strict rules: the *RUNSIMUSER* function must **always** be present, and the *OPENSIMUSER* and *CLOSESIMUSER* functions are optional.

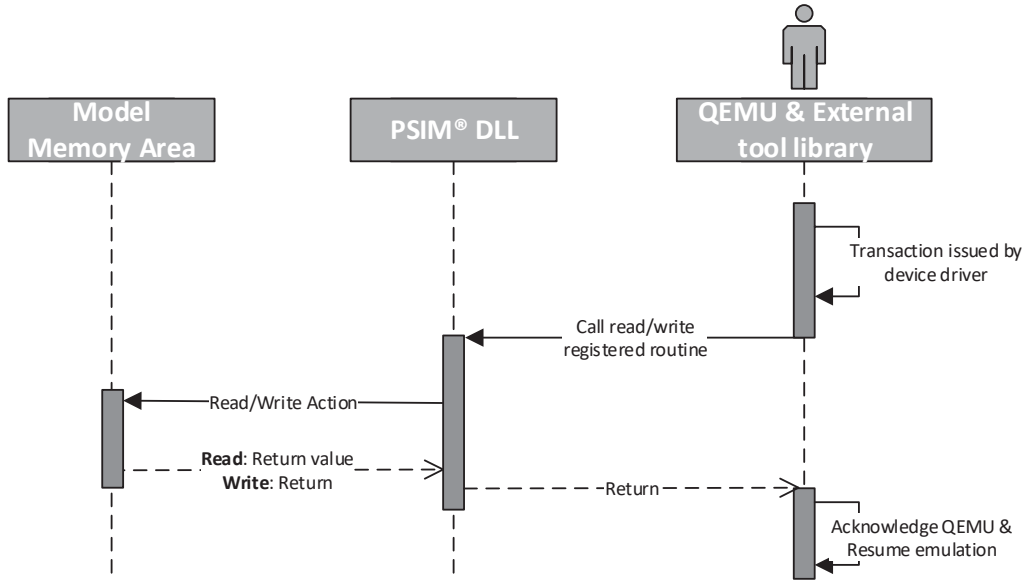


Figure 4.16: PSIM® DLL transaction sequence diagram

OPENSIMUSER Routine

In the co-simulation library, the *OPENSIMUSER* routine must always be present, as it will issue the initialization process seen in the last subsection. This routine is called only once, right before the simulation starts.

This functions' behavior is to register the transaction routines and connect PSIM® with QEMU, using the External Tool Library interfaces.

The following code shows an example of this routine:

```
1  void OPENSIMUSER(...)
2  {
3     qemu_register_model
4     (
5        model1.name,
6        model1.memory_mapped_address,
7        model1.mapped_area_size,
8        model1.interrupt_requests_size,
9        (const char*)  model1.interrupt_requests
10    );
```

```
11
12      qemu_register_model
13      (
14          model2.name,
15          model2.memory_mapped_address,
16          model2.mapped_area_size,
17          model2.interrupt_requests_size,
18          (const char*)  model2.interrupt_requests
19      );
20
21      if(qemu_connect()) printf("\nConnect OK.\n");
22      else printf("\nConnect failed.\n");
23
24      pthread_mutex_init(&in_mutex, NULL);
25      pthread_mutex_init(&out_mutex, NULL);
26  }
```

This functions' implementation should follow the flowchart present in Figure 4.17.

When the connecting routine (*qemu_connect_tool*) is called, information about the tool and models to be registered must be provided. The structs *tool_info_t* and *model_info_t* defined by the External Model Library will be used in order to do so.

This is the tool information struct that must be used for the PSIM® library:

```
1  static tool_info_t psim_info =
2  {
3          .name = "PSIM",
4          .domain = "Power Electronics",
5          .models = NULL,
6          .n_models = 0
7  };
```

This is a template of a struct to be used for the models that are going to be used as external models:

```
1  static model_info_t MODEL = {
2          .name = "Model_Name",
3          .memory_mapped_address = 0x54113000,
```

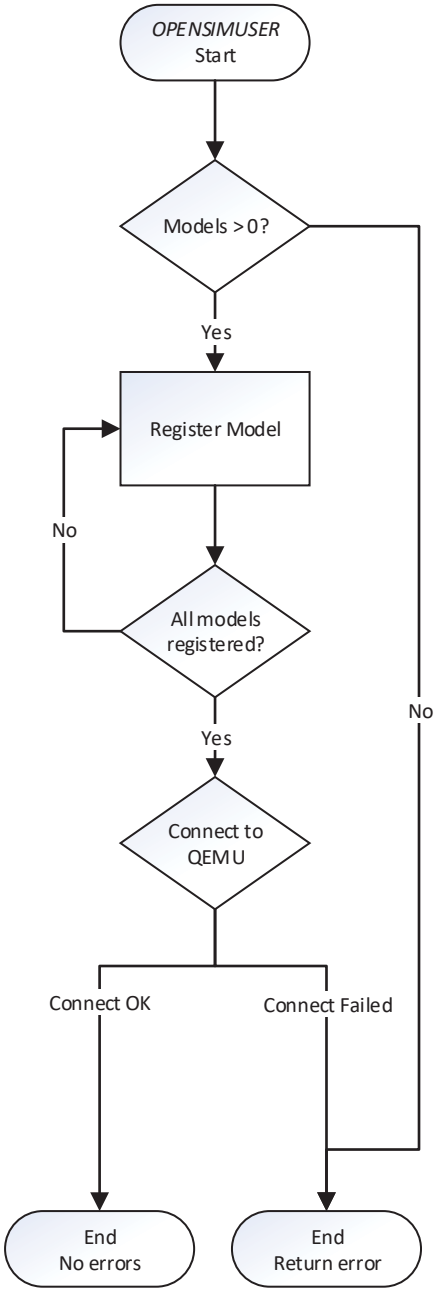


Figure 4.17: PSIM® DLL *OPENSIMUSER* flowchart

```
4         .mapped_area_size = 32
5 };
```

This function also has the duty to allocate the memory needed by every external model, given by the *mapped_area_size* parameter in the models' information

structure. This allocated memory will be accessed every time a read/write transaction is issued to the model.

CLOSESIMUSER Routine

This routine must be present in the co-simulation library, so the connection to QEMU can be closed properly when the PSIM® simulation ends, which is the only time this routine is called.

The following code shows an example of this routine:

```
1 void CLOSESIMUSER(...)
2 {
3         memory_cleanup();
4         qemu_close_connection();
5         printf("Simulation ending\n");
6 }
```

This function should only free any memory allocated by the DLL and then call the *qemu_close_connection* function from the External Tool Library.

Given this, its' flowchart is very simple, and is present in Figure 4.18.

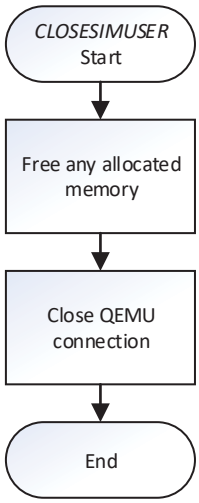


Figure 4.18: PSIM® DLL *CLOSESIMUSER* flowchart

RUNSIMUSER Routine

As mentioned in subsection 3.4.2, this routine is mandatory for any DLL being used on PSIM® DLL Blocks, and is called at each simulation time step.

The following code shows a possible implementation for this routine:

```
1  void RUNSIMUSER(...)
2  {
3     pthread_mutex_lock(&in_mutex);
4
5     model1_in[0] = in[0];
6     model1_in[1] = in[1];
7     model1_in[2] = in[2];
8     model1_in[3] = in[3];
9
10    model2_in[4] = in[4];
11    model2_in[5] = in[5];
12
13    pthread_mutex_unlock(&in_mutex);
14
15 //Synchronization goes here, if needed
16
17    pthread_mutex_lock(&out_mutex.trans_mutex);
18
19    out[0] = model1_in[0];
20    out[1] = model2_in[0];
21    out[2] = model2_in[1];
22
23    pthread_mutex_unlock(&out_mutex);
24 }
```

This functions' parameters include two arrays called **in** and **out**, respectively representing the input and output data. Given this, this functions' flowchart should always be close to the presented in Figure 4.19, which is pretty straightforward.

This functions' first task will always be loading the input data received via the **in** array of values, either into the external simulation models registers or other auxiliary variables. After that, the library must make sure it can proceed to the

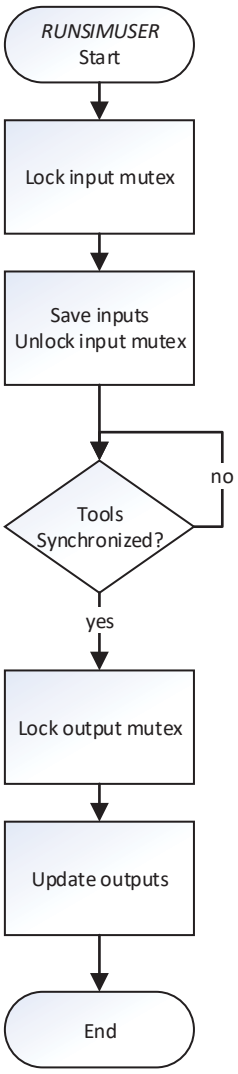


Figure 4.19: PSIM® DLL *RUNSIMUSER* flowchart

next time step. This will happen when other tools need to be synchronized with the PSIM® simulation. This synchronization can be very useful, even indispensable, in cases where the inputs must be used for processing in an external tool and the simulation has to wait for the processed values in order to write them on the output array, allowing the external tool to influence the simulation. The mechanisms used to do this may be very simple and can vary a lot, depending on the application scenario under test.

By updating the output values, using the **out** array, the simulation components

behavior such as switching semiconductor may be changed.

4.3.3 Library Transaction Handling

As above-mentioned, the QEMU Co-Simulation PSIM® library uses the External Tool Library (3.2.2) in order to register the transaction routines in order to handle the transactions issued by the QEMU emulated machine.

The transaction routines registered on the External Tool Library by the DLL will receive the transaction requests coming from the device drivers being emulated on QEMU.

In order to get the transaction information to the respective external model, the memory address present in the received message will be compared. The offset will also be checked, in order to perform the write/read action on the correct register amongst all the mapped memory area each model owns.

Unlike what happens with the QEMU Co-Simulation DPI Library (4.2), the registered transaction routines can directly access each models' allocated memory. This makes the implementation of the transaction handling mechanisms much easier.

Figure 4.20 presents a more precise sequence diagram for the the QEMU Co-Simulation PSIM® Library. As already mentioned, and is referred in the figure, read and write transactions may happen several times, as many as needed. This number is determined by the software application running on the QEMU emulated machine. That same application is what determines the flow of the transactions, along with the synchronization between QEMU and PSIM®.

As seen above, the DLL inputs are saved at the very start of the *RUNSIMUSER* function, so the PSIM® simulation always has the actual values stored and ready to be read by the software application.

The read transactions must always happen first, as the PSIM® simulation will have different input values at each time step. These transactions will allow the software application to receive updated values.

When they are issued by QEMU, coming from the device driver, the registered read routine will be called. There, the address received in the transaction message will be compared with the registered models on the PSIM® DLL, so the transaction

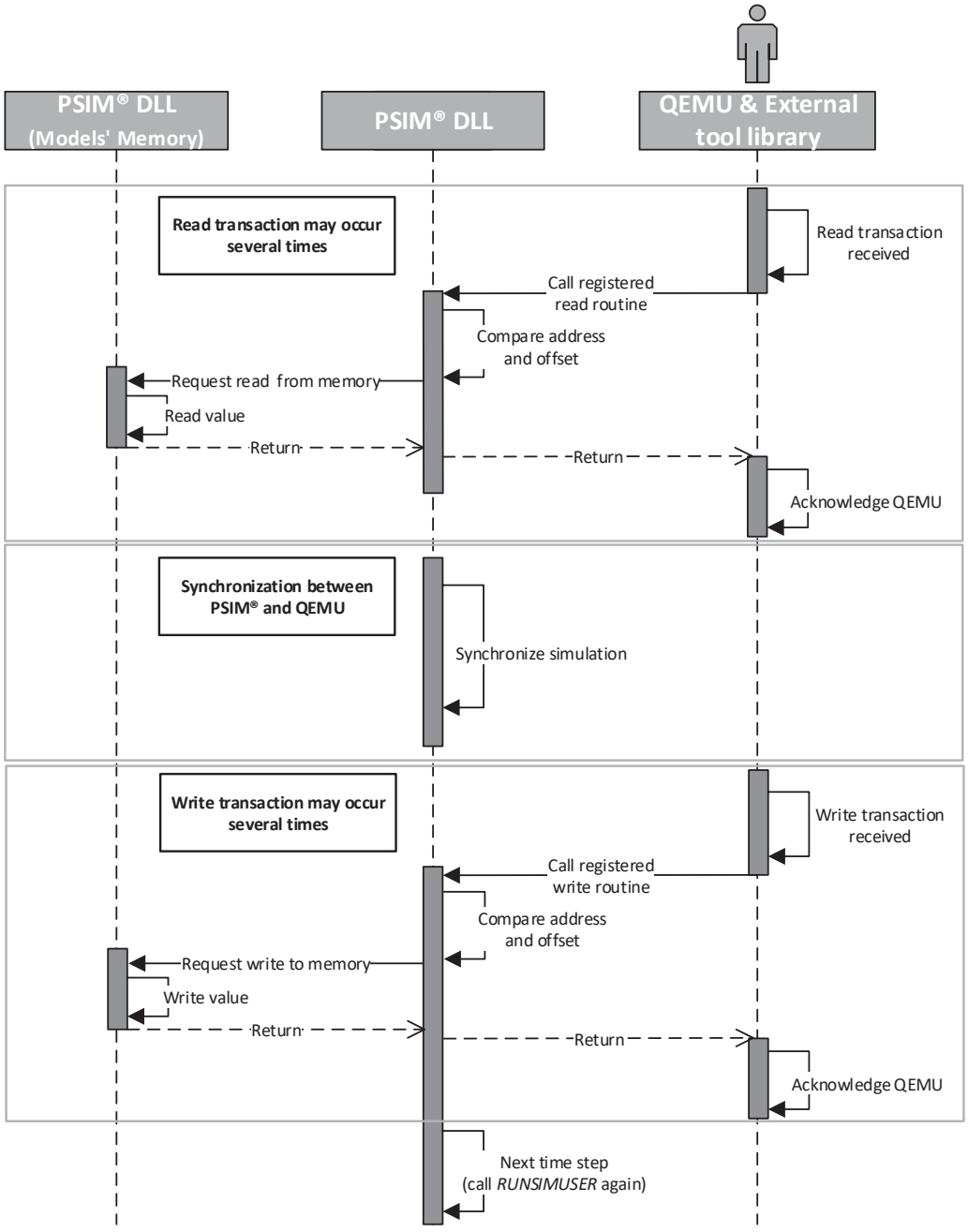


Figure 4.20: PSIM® Library transaction handling sequence diagram

is directed to the correct model. The received offset will also be checked, in order to access the correct register amongst the models' allocated memory.

After all the readings have been performed, the software application running on the emulated machine will use the received values in order to process them and perform

its' specific algorithms. When the data and algorithms have been processed, the outputs of the software application have to be sent back to the PSIM® simulation, which is waiting for that same data in order to proceed to the next time-step. This synchronization between the tools will be addressed below.

The write transactions are very similar to the read transactions: the registered routine is called and compares the received address and offset in order to write the received value into the correct register from the correct model allocated memory. When all the write actions have completed, the DLLs output values will be updated and the PSIM® simulation may proceed to the next time-step.

PSIM® & QEMU Synchronization

As mentioned in the last subsection, there may be a need for the software application running on the QEMU emulated machine to synchronize with the PSIM® simulation, for instance when the input values from the simulation are needed to the control algorithms and feedback must be given back to the PSIM® simulation in order to alter the hardware behavior, for example by changing the semiconductor driver signals.

The mechanisms used to achieve this synchronization can be as simple as counting the number of read/write transactions, and when all have been completed, move on to the next time-step on the PSIM® simulation. The software application will always be the trigger to the synchronization, since it controls the co-simulation flow.

In order to prevent race conditions and maintain data/transaction integrity, mutual exclusion mechanisms can and should be used, as they bring integrity to the system, even though good algorithms implemented in the software application can avoid using them.

Chapter 5

Application Scenario

In order to demonstrate the developed work in a practical context, an application scenario stimuli for the proposed design flow was selected. As already mentioned, this dissertation has a great focus on power electronics systems, so the chosen case belongs to that area, more precisely on the electrical grid and the power electronic devices used to study, evaluate and improve quality. The chosen scenario corresponds to a shunt active power filter that is being used with merit for compensating current harmonics, unbalances and installation power factor.

The instantaneous active and reactive power theory, usually referred to as "p-q Theory", was introduced in 1983 and is very useful when dealing with controllers of power *conditioners* (Akagi et al., 2007). The digital controller for a shunt active power filter, operating over harmonic currents along with current unbalance, following a constant power at source side strategy on power systems, based on the above-mentioned p-q Theory (Akagi et al., 2007), may be developed following the design flow adopted in this dissertation and described in section 4.1.

An overview diagram for such design flow was already presented, and can be seen in Figure 4.2. Figure 5.1 presents the procedures taken in order to move from one phase to the other, in order to help the reader understand this design flow. These steps will be presented during the course of the current chapter, where each step will be addressed.

In order to comply with that same design flow, the controller for the active power filter was first modeled and validated by developing a software application that corresponds to the desired system, using PSIM as stimuli. After that, the software application was parallelized using a software multi-threading paradigm. The multi-

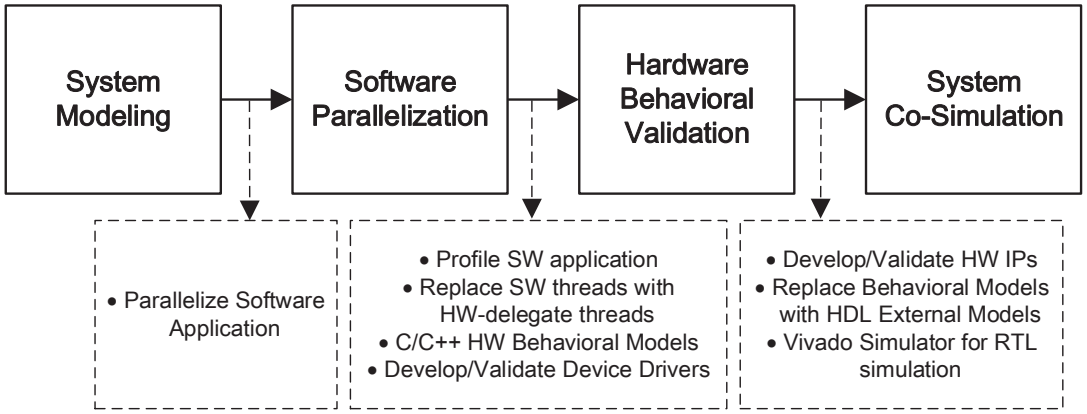


Figure 5.1: Design Flow Methodology

threaded software application was profiled in order to identify the most *CPU-heavy* tasks in the system, as they are the candidates for hardware acceleration. Then, the hardware accelerators behavior was validated using C/C++ behavioral models, and only then the HDL IPs were developed. At this time, the integrated co-simulation environment allowed the simulation of the system in it's three domains: embedded software, hardware accelerators and power electronics.

5.1 System Modeling

As above-mentioned, the chosen application scenario was Shunt Active Power Filter, and its' controller follows the p-q Theory by Akagi et al. (2007). The filter is connected to a three-phase, three-wire, electrical power grid where some non-linear loads are present, causing harmonic currents along with current unbalance.

Nowadays, since many of the loads present in electrical installations are non-linear, their power consumption consists of active and reactive powers. Only the active power contributes to energy consumption by the load, with reactive power being an undesired component. Still, the reactive power is provided by the electrical grid, and will therefore be taxed if significant. A Shunt Active Power Filter allows the compensation of the reactive powers, thus reducing their presence in an electrical installation.

5.1.1 System Overview

The system under study is composed of a three-phase, three-wire, electrical power source and the loads being supplied by it, which will be described later, but can either be linear or non-linear, single-phase or three-phase, with the The Shunt Active Power Filter being connected to this electrical installation. An overview diagram of the described system can be seen in Figure 5.2.

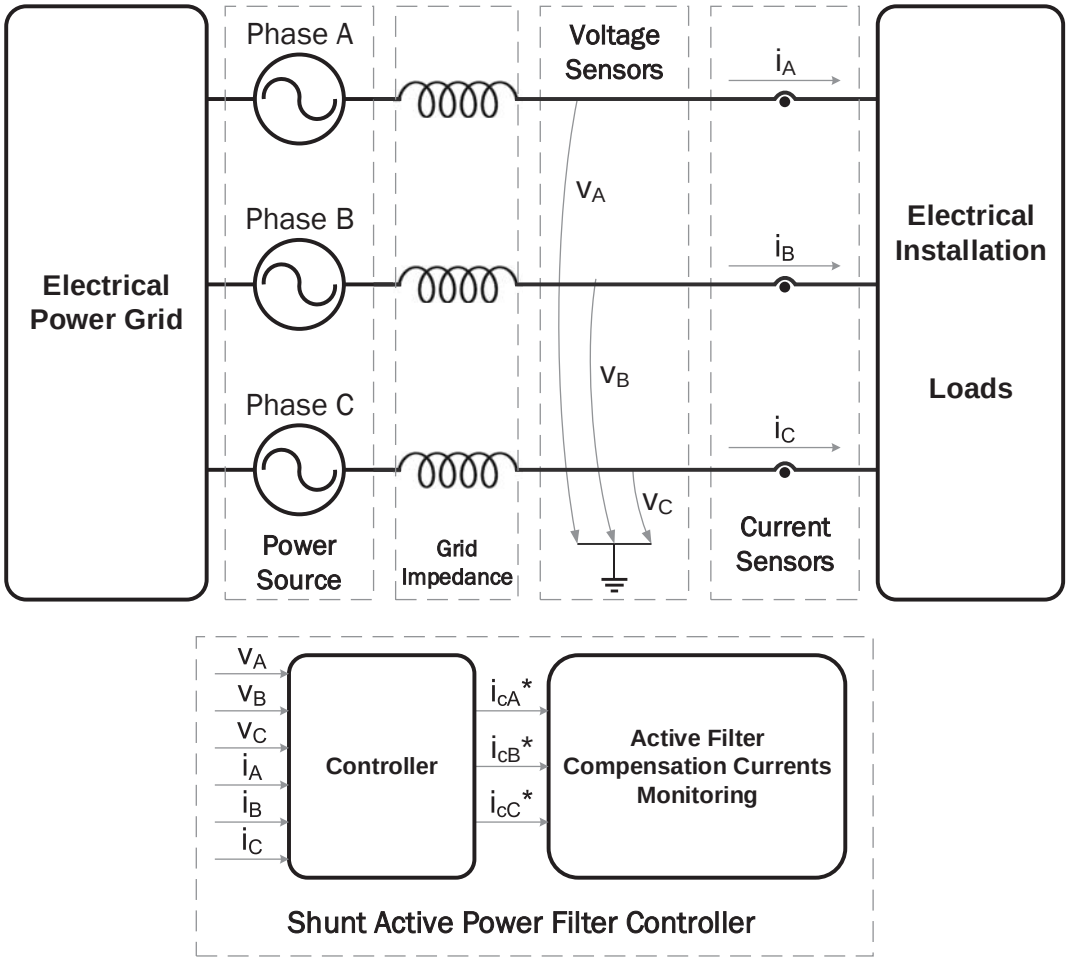


Figure 5.2: Three-phase, three-wire shunt active power filter overview diagram

Some hardware components other than the above-mentioned must also be present in such a system, like voltage and current sensors for the three phases, which are the inputs for the controller.

The controller must compute the reference currents that must be produced by a power inverter stage, in *real-time*, to force a power converter to synthesize them

accurately. The shunt active filter may then compensate the harmonic currents of non-linear loads, continuously tracking changes in its harmonic contents.

5.1.2 Controller Modeling

The p-q Theory by Akagi et al. (2007) defines instantaneous real and imaginary powers through the use of the Clarke transform. The controller for the shunt active power filter is based on that control theory, allowing the compensation of the unwanted power components, thus reducing their presence in the electrical installation. Appendix D contains the mathematical fundamentals of the p-q Theory.

The calculated real power (p) of the load can be separated into its average ($\bar{p}$) and oscillating ($\tilde{p}$) parts. Likewise, the load imaginary power (q) can be separated into its average ($\bar{q}$) and oscillating ($\tilde{q}$) parts. Then, undesired portions of the real and imaginary powers of the load that should be compensated are selected (Akagi et al., 2007).

Controller Description

The idea is to compensate all undesirable power components generated by non-linear loads that can damage or make the power system overloaded or stressed by harmonic pollution. This way, it would be desirable for a three-phase balanced power supply to deliver only the average real power ($\bar{p}$) of the load. Thus, all the other power components required by non-linear loads ($\tilde{p}$, $\bar{q}$, $\tilde{q}$, $\bar{p_0}$ and $\tilde{p_0}$) should be compensated by a shunt compensator (Akagi et al., 2007).

A particular characteristic of three-phase, three-wire systems is the absence of the neutral conductor, and, consequently, the absence of zero-sequence current components ($p_0 = \bar{p_0} + \tilde{p_0}$). Thus, the zero-sequence power is always zero in these systems ($p_0 = 0$) (Akagi et al., 2007). Since the system under study is a system like this (Figure 5.2), the zero-sequence power components are discarded.

The control algorithm implemented in the controller of the active filter determines the compensation characteristics of the system. The controller design is particularly difficult if the shunt active power filter is applied in power systems in which the supply voltage itself has been already distorted andor unbalanced (Akagi et al., 2007). The chosen control strategy, as already briefly mentioned, was the constant

instantaneous power. This technique guarantees that only the $\bar{p}$ portion of power is drawn from the source.

As stated by Akagi et al. (2007), when compensating both the $\tilde{p}$ and all q components, all the undesirable current components of the loads are being eliminated. In a scenario like this, and if the voltages are sinusoidal and balanced, the compensated current is sinusoidal, produces a constant real power and does not generate any imaginary powers. The source current has a minimum **R**oot **M**ean **S**quared (RMS) value that transfers the same energy as the original load current that produce the average real power ($\bar{p}$). This is the best compensation that can be made from the power-flow point-of-view, because it smooths down the power drawn from the electrical grid, and eliminates all the harmonic currents. However, it should be pointed out that this is a particular situation in which no unbalances or distortions are present in the system voltages.

Control Algorithm

Figure 5.3 presents the algorithm of a controller for a three-phase, three-wire shunt active power filter that compensates the oscillating real power ($\tilde{p}$) and the imaginary power (q) of the loads and follows the constant instantaneous power control strategy.

As can be seen in the block diagram of the controller (Figure 5.3) and the overview of the system (Figure 5.2), the controller inputs are the voltages and currents for the three phases *a*, *b* and *c*.

The first task performed by the controller is the *α-β-0 transformation*, also known as the Clarke Transformation. Basically, it maps the three-phase instantaneous voltages/currents in the abc phases (v_a, v_b, v_c) into the instantaneous voltages/currents on the *α-β-0* axes (v_α, v_β and v_0).

The next block calculates the instantaneous powers of the loads using the α-β voltages and currents, according to the mathematical operations described in the p-q Theory. As already stated, in a three-phase, three-wire system, the zero-sequence powers are always zero. Still, they are present in the diagram since calculation errors may cause the zero-sequence powers to exist. By introducing these powers into the instantaneous powers calculation may reduce errors in the compensation currents calculation.

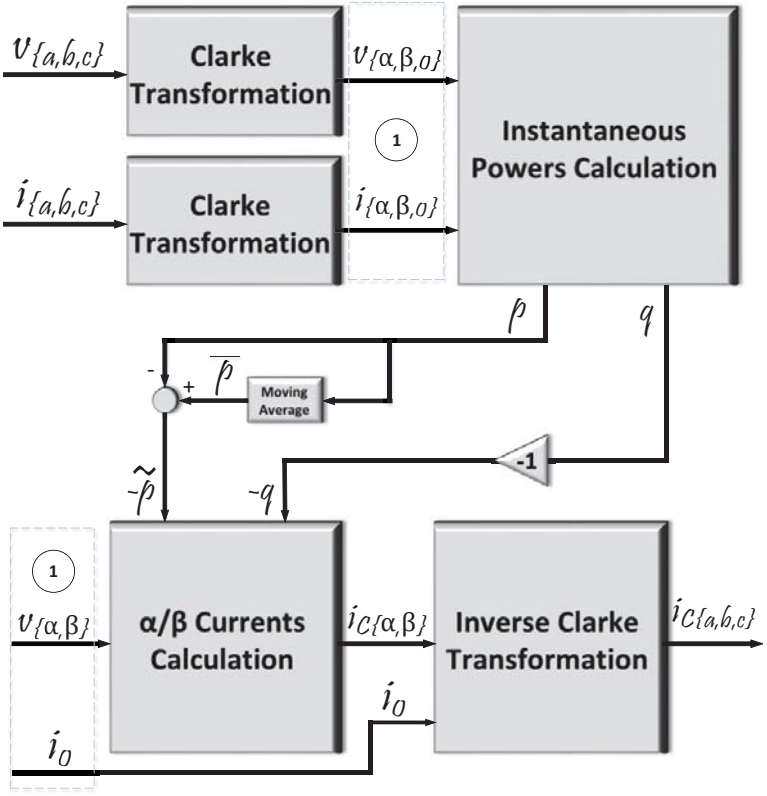


Figure 5.3: Block diagram for the shunt active power filter controller

Next, the power components to be compensated must be selected, and as already mentioned, the chosen control strategy for the controller states that only the constant portion of the real power ($\bar{p}$) should be supplied by the source to the loads. Given this, the alternate real power ($-\tilde{p}$) along with the total imaginary power ($-q = -\bar{q} - \tilde{q}$) should be compensated. In order to separate p into $\bar{p}$ and $\tilde{p}$, a moving average filter may be used in a digital implementation, being very simple and effective (Akagi et al., 2007).

Next, the reference compensation currents $i_{c\alpha}$ and $i_{c\beta}$ are calculated, based on the selected power components to be compensated and the α / β voltages previously calculated. After calculating the compensation currents in the α-β *reference frame*, they must be converted back to the *abc reference frame*. To perform this action, the Clarke Transformation will again be used, but this time in its' inverse form.

When applying the Inverse Clarke Transformation to the α-β compensation currents, the compensation currents for the a, b and c phases are obtained. These currents are used as references for the inverter control unit that will determine

when to turn on or off the power semiconductors in order to inject the compensation currents into the electrical installation. The ideal compensated current can be calculated simply by subtracting the eliminated current from the load current ($i_{\text{compensated}} = i_{\text{loads}} - i_{\text{compensation}}$) (Akagi et al., 2007).

A software implementation of the described algorithm for the controller of the shunt active filter was developed, following the p-q Theory mathematical fundamentals present in Appendix D along with the block diagram present in Figure 5.3. This software implementation was used to validate the behavior of the control system algorithms. As mentioned in subsubsection 4.1.1, during this phase PSIM is used in order to model the power electronics hardware along with the controller (the software application is compiled as a PSIM DLL).

5.1.3 PSIM® Simulation

The schematic of the circuit simulated in PSIM® and used as stimuli for the controller of the shunt active filter can be seen in Figure 5.4.

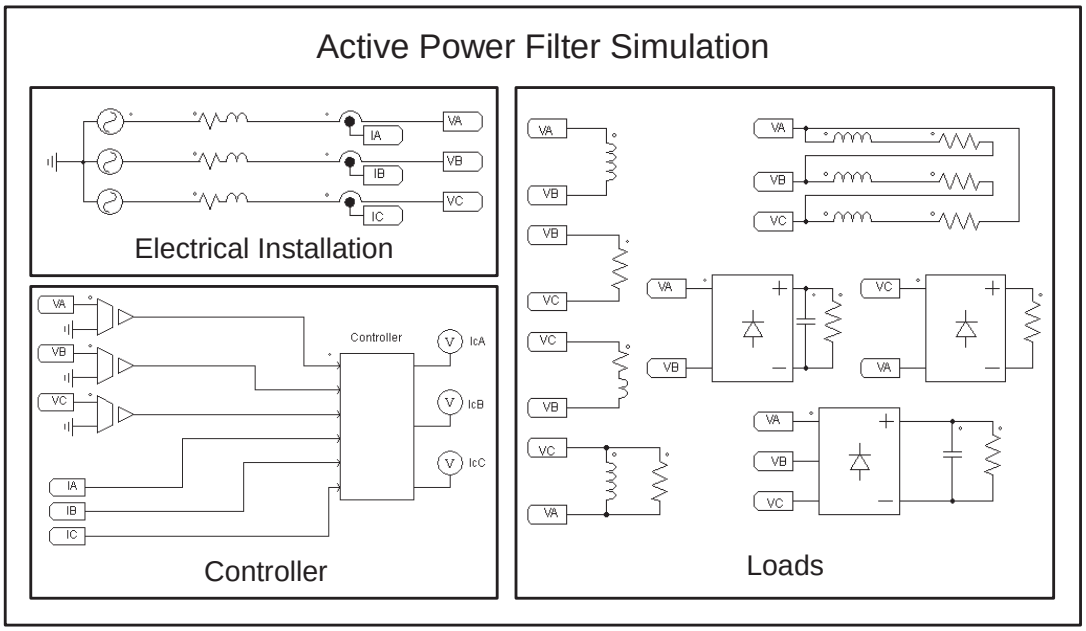


Figure 5.4: Schematic of the PSIM® circuit used in the simulation

The schematic follows the system overview presented in Figure 5.2, and is divided into three parts: the electrical installation, the active filter controller and the loads. The electrical installation is composed of a 400 V RMS three-phase power source

along with the grid impedance. There are also three voltage sensors sensing each phase voltage, along with three current sensors sensing the three phases currents. The controller is basically composed of the DLL Block, with its' inputs being the sensed voltages and currents for each phase, and the outputs being the reference compensation currents for the three phases A, B and C.

The loads used were the following:

- An 100 mH inductor connected from phases A to B;
- An 10 Ω resistor connected from phases B to C;
- An 1 Ω resistor in series with a 15 mH inductor connected from phases C to B;
- An 100 mH inductor in parallel with a 10 Ω resistor connected from phases C to A;
- An one-phase diode bridge with a parallel of a 2 mF capacitor and a 50 Ω resistor, connected from phases C to A;
- An one-phase diode bridge with a 50 Ω resistor connected from phases C to A;
- A three-phase 5 Ω resistive load in series with with a 100 mH inductor connected in delta;
- A three-phase diode bridge with a parallel of a 1 mF capacitor and a 10 Ω resistor.

Appendix E contains the results of the performed simulations, where the DLL Block is running the developed software application that mimics the control algorithm above described.

5.2 Software Parallelization

Following the design flow presented in section 4.1, and already refered in this chapter, when the algorithm for the software application is working properly, and as it becomes more and more complex, the system core tasks must be identified, in order to split it into smaller parts. Then, the software will be parallelized by means of a multi-thread programming model.

In this step, the multiple task algorithms are assigned to different threads. Since threads usually share data, synchronization mechanisms are mandatory to avoid race conditions, and are usually provided by the multi-threading API. Figure 5.5 presents the task graph of the system, where the core tasks of the system are identified and were already assigned to different threads, along with the synchronization mechanisms be used.

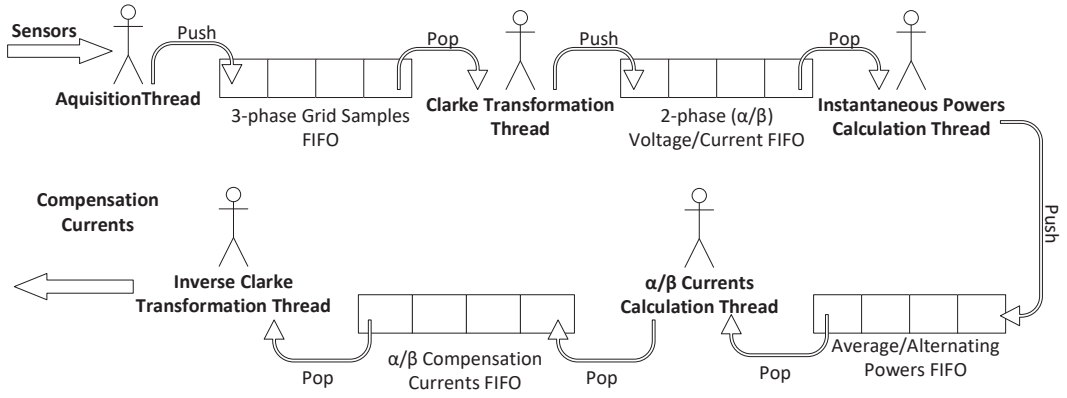


Figure 5.5: Active power filter controller application task graph

The modeled system was implemented using the C++ programming language, and implement each different task in software, POSIX threads were used. Similar blocks were grouped in the same thread, such as voltage and current acquisition, or the Clarke Transformations for the voltage and current values.

The programming technique used to achieve concurrency (topic discussed in subsection 2.2.1) was the *mesa-style semantics*, which is based on the producer-consumer synchronization problem. This was the chosen technique since it is more efficient, requiring less context switches than for example Hoare semantics, and signal broadcasting is also easier to implement.

As already mentioned, POSIX synchronization mechanisms were used, namely mutexes and condition variables. **F**irst **I**n **F**irst **O**ut (FIFO) queues were used to store data and allow threads to process data independently, breaking the sequentiality between them.

Figure 5.6 presents the first part of the **U**nified **M**odeling **L**anguage (UML) class diagram for the developed application, since only the "main" class, *CSystem*, along with the processing and queue classes are represented. The rest, namely the thread data classes, are to be presented later in the second part of the UML class diagram.

As can be seen in the first part of the class diagram, the *CSystem* class groups all

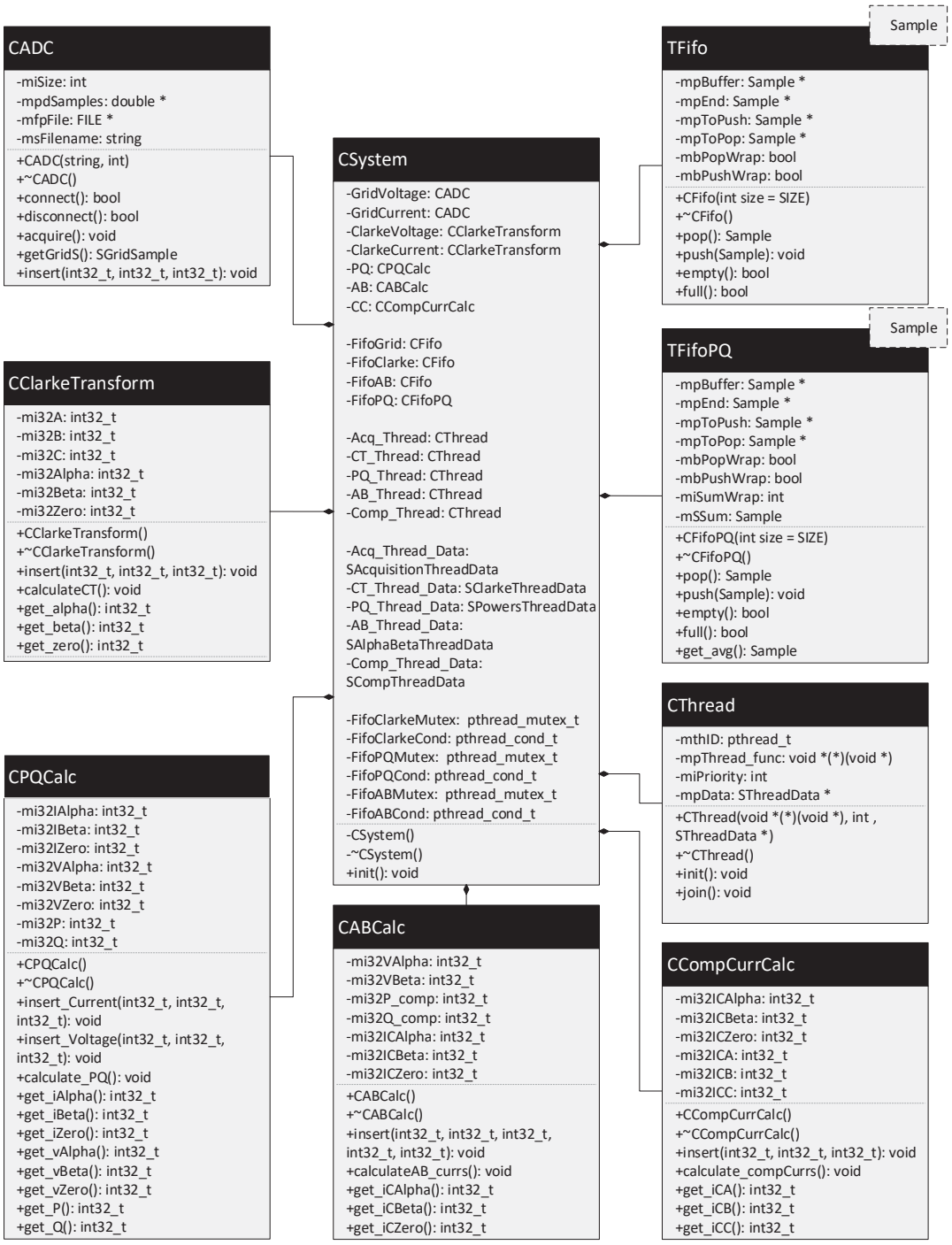


Figure 5.6: Active power filter controller application UML, part 1

the objects that will be used by the threads. Only 1 object of this type will be instantiated, that will initialize and join the threads.

The template classes *TFifo* and *TFifoPQ* are very similar, but *TFifoPQ* has special operators to deal with the mean real power calculation. These classes represent

the FIFOs to be used by the application in order to transfer data between threads.

The *CADC* class represents the "Aquisition" task in figure 5.3. It implements mechanisms to read values from the PSIM simulation and place them on a FIFO, acting as an **A**nalog to **D**igital **C**onverter (ADC).

The *CClarkeTransform* class represents the "Clarke Transformation" task in the task graph. This class grabs the values from the ADC and perform the Clarke Transformation, placing the results in the next FIFO.

The *CPQCalc* class represents the "Instantaneous Powers Calculation" task in the task graph. It uses the α/β voltage and current received values in order to calculate the real and imaginary powers. It will then push the values into a PQ-FIFO, as above-mentioned.

The *CABCalc* class represents the "α/β Currents Calculation" task in the task graph. This class will calculate the compensation currents in the α/β reference frame, and place them on the last FIFO.

The *CCompCurrCalc* class represents the "Inverse Clarke Transformation" task in the task graph. It will grab the α/β compensation currents from the last FIFO and perform the Inverse Clarke Transformation on the voltage and current values. These values will then be sent back to PSIM, so they can be shown in the integrated waveform viwer SimView.

The *CThread* class is present in the next class diagram, and will then be explained.

Figure 5.7 presents the second and last part of the UML class diagram for the developed application.

In this diagram, the *CSystem* class is again shown, but in a simplified representation, since it was already shown in the last one. The *CThread* is fully also replicated. The other represented classes are the thread data structures and the different sample types data structures.

The *CThread* objects represent the running threads and contain each thread identifier and priority along with the start routine and data container.

The *SThreadData* struct is an empty base-struct, so the *CThread* class can have a generic data container, represented by the derived-structs *SAcquisitionThreadData*, *SClarkeThreadData*, *SAlphaBetaThreadData* and *SCompThreadData*. Each one of these structs have different members, in order to provide their correspondent

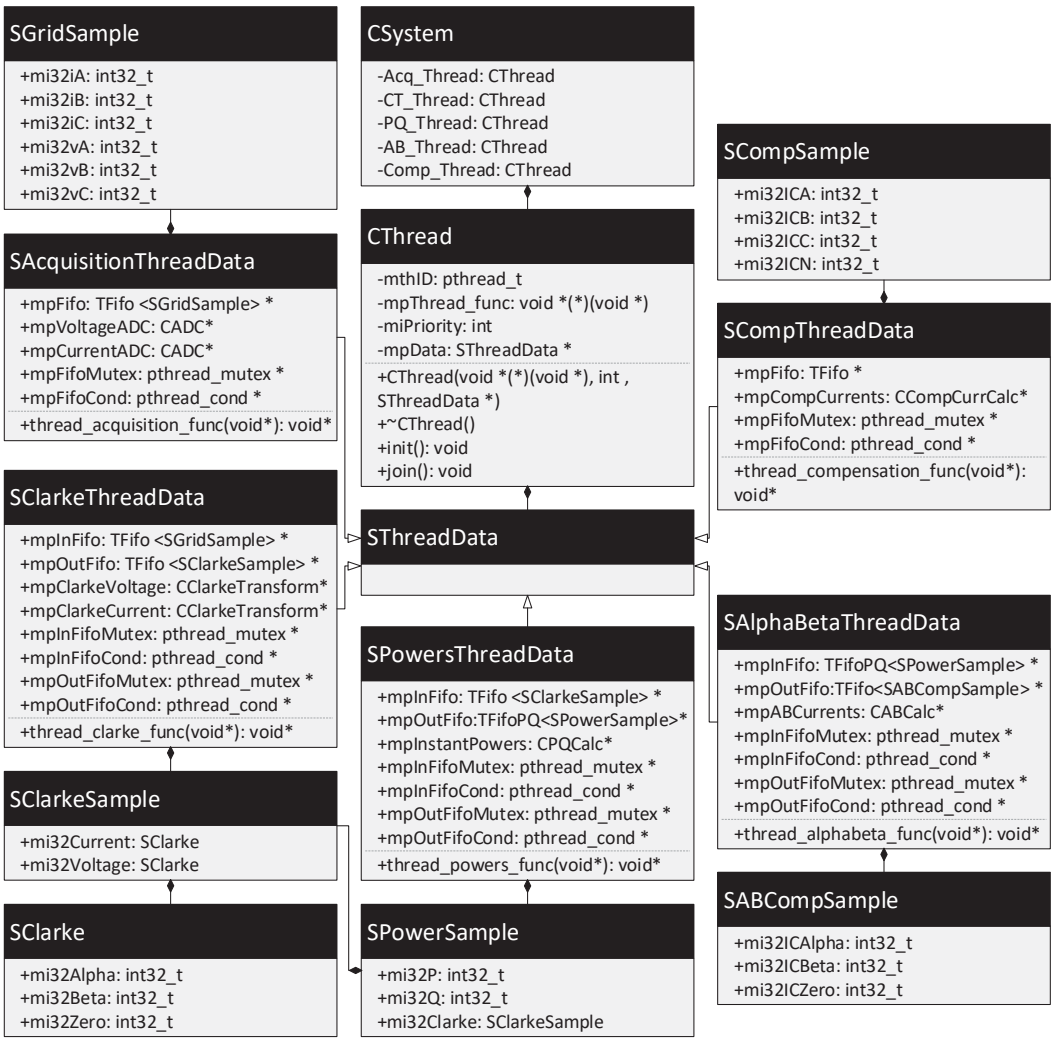


Figure 5.7: Active power filter controller application UML, part 2

thread with the needed data.

5.3 Hardware Behavioral Validation

The functionality of the developed software application for the shunt active power filter controller was proven in the last subsection, but as mentioned in the beginning of the current section, in order to identify the threads to be migrated to hardware the application has to be profiled.

5.3.1 Profiling

Oprofile was used to profile the software-only application. It is unobtrusive, which means there is no need for special recompilations or wrapper libraries when using it, instead being based on CPU performance counters (Oprofile, 2015).

The results from the software application profiling are dependent on the target architecture, so the profiling results can be misleading, since the hardware behavior can be very different from the expected. Even though, by profiling the software application, the developer can take some impressions: the most critical tasks as seen from the profiling results are good candidates for hardware acceleration. From the profiling results obtained, the Clarke Transformation thread is one of the most critical tasks in terms of processing, along with the Instantaneous Powers Calculation, the first due to its' calculus, and the second due to the number of memory accesses.

5.3.2 C/C++ Behavioral Models

Again following the design flow presented in section 4.1, before developing the accelerators, the hardware's behavior can be validated using the QEMU Plugin Extension developed by Naia (2015), with behavioral C/C++ models for the migrated hardware being integrated into the emulated machine. This allows testing the behavior of the hardware before developing the HDL IP's, allowing the development and validation of the device drivers, that may reveal a very lengthy task.

The hardware-software application will now run over the Target Linux Kernel, on the QEMU emulated target machine. As already mentioned, the software application must be modified, with delegate threads replacing the software threads identified as candidates for hardware acceleration.

Hardware Delegate Threads

Figure 5.8 shows how the Clarke Transformation Software Thread behaves as regards to the Clarke Transformation calculus: in order to perform it, the thread invokes the *calculate_CT* function from the *CClarkeTransformation* class.

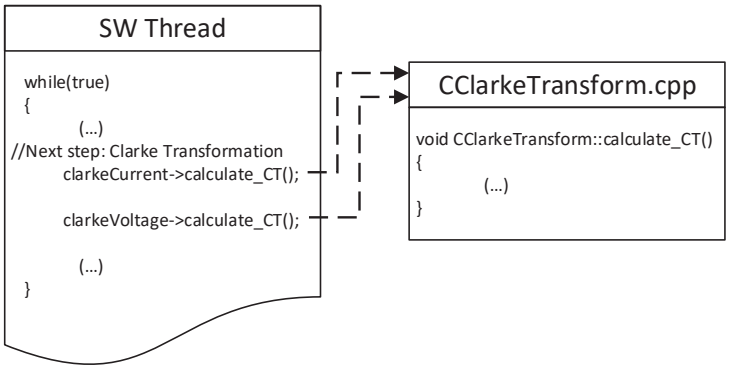


Figure 5.8: Clarke Transformation SW Thread Processing Overview

As already mentioned, the hardware delegate threads must entrust their processing to the C/C++ behavioral models. This procedure can be observed in Figure 5.9, and the main difference from the SW thread is that when the Clarke Transformation calculus must be performed, the application performs a read or write system call to a device driver. This device driver will perform a write/read on the hardware, that QEMU will grab and send to the Clarke Transformation Behavioral Model, which is a C/C++ QEMU Plugin Model. This plugin acts as a hardware representation for the Clarke Transformation, and will perform its' calculus.

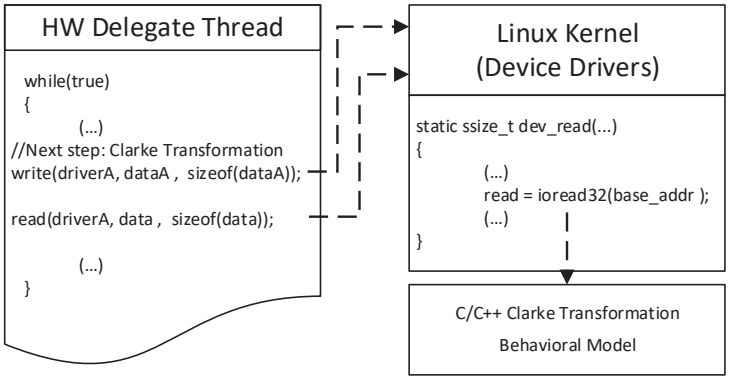


Figure 5.9: Clarke Transformation HW Delegate Thread Processing Overview

This way, the device drivers can be easily tested and validated, so that when passing to the RTL simulation phase, there are no issues related to them.

5.4 System Co-Simulation

Since the hardware's behavior and the device drivers were already validated in the last phase, the hardware accelerators can now be developed and tested, with the RLT simulation being performed on Vivado Simulator and the simulated IP's replacing the previously used C/C++ behavioral models. The hardware delegate threads remain, and will now entrust their processing to the external HW models present in the external HDL simulator.

After the hardware IP's have been developed, the system can be simulated in all it's domains: embedded software, hardware accelerators and electrical circuit. This is possible by using QEMU along with the simulation extensions developed during this work and the ones developed by Naia (2015), and Figure 4.6 presents an overview diagram of the co-simulation environment.

The hardware IPs being simulated on Vivado Simulator will communicate with QEMU using the simulation extensions described in chapter 4, via device driver system calls made from the hardware delegate thread present in the now altered software application described in the last subsection.

5.4.1 Hardware Acceleration

As above-mentioned, the chosen task for hardware acceleration was the *Clarke Transformation*. Given this, an hardware IP that performs the Clarke Transformation must be developed and tested. In this work, the used HDL was SystemVerilog, and the used RTL simulator was the Vivado Simulator, as already mentioned in the document.

The Clarke Transformation IP will behave the same as the Clarke Transformation Behavioral Model used in the last design flow phase. As the software thread for the Clarke Transformation task was already migrated into a delegate hardware thread, and the device driver was developed and validated, testing the developed HW IP along with the rest of system is now faster and less prone to errors.

Appendix B.6 contains the Clarke Transformation IP SystemVerilog code used to co-simulate the system.

Chapter 6

Conclusion

This chapter finalizes this document, summing up this dissertation by reviewing the developed work. Some interesting topics regarding future work will also be mentioned.

In order to achieve the objectives of this challenging project that brought together the Embedded Systems and Power Electronics domains, a broad set of skills was needed. The required knowledge areas are varied, including embedded system design and platform bring-up, Linux device driver development, dynamic library integration, HDL dedicated co-processor design, HDL simulation interface frameworks, HVL simulation interface frameworks, analog hardware simulation interfaces, harmonic current compensation from electrical installations and co-simulation environments.

Given that this work is framed with the ESRGs concept of co-simulation, and is a sequel to a previous work, the first stages were based on assimilating the already developed concepts and interfaces. This dissertation leaves "open doors" for further development and improvement of the current co-simulation framework for hardware accelerated embedded systems.

Given the above-mentioned reasons, along with the project's validation using the obtained results, it is concluded that the educational and project objectives were accomplished.

6.1 Developed Work

Concerning the developed work, the simulation extensions that had been completed in the past along with the ones developed during this dissertation enables the creation of an integrated co-simulation environment. The initial software application trials using rapid deployment of behavioral hardware IPs as plugins, that was made possible before this work, were now complimented with extensions that allow not only for co-simulation between software and hardware acceleration , but also involving the power systems domain.

The possibility of stimulating the system under test using simulation tools such as PSIM®, which also enables graphic and analytic verification of the outputs, is a great improvement and a step forward in the development of a framework for co-simulation purposes.

By using a real application scenario to validate the behavior of the co-simulation environment, the hardware acceleration and power systems related extensions were tested, along with the developed Linux device drivers. The presented design flow for hardware accelerated embedded systems was also validated, which allows for faster development of projects of this kind.

However, the HDL modules simulation used does not model any system bus interactions accurately, and when it comes to real system buses, the transactions will surely be slower. In order to prevent this shortcoming, every developed hardware co-processor must be wrapped in a slave wrapper for the specific architecture's system bus.

6.2 Future Work

Regarding possible future work, and as already stated, this project opens up many interesting possibilities for further improvements that may result in benefits to the design flow, minimizing development time and costs.

To begin with, support for commonly used SoC-based platforms such as the Xilinx Zynq family could be added to QEMU. This task may prove easier than it looks like, since Xilinx actively develops a QEMU tree for both Microblaze, Zynq and Zynq UltraScale+. By adding support for these architectures, the deployment of the developed hardware accelerators along with the simulated software

would become much easier and direct, requiring less changes in order to be fully functional.

As mentioned in section 2.4.1, HLS tools can be used in order to fasten up the hardware acceleration phase, since they use behavioral descriptions to create digital hardware that implements such behavior. By using HLS tools such as Xilinxs' Vivado High-Level Synthesis, the algorithms may be developed and verified in software languages like C/C++ and then converted to HW IP's.

Another enhancement would be to implement cycle-accurate emulation for system buses commonly used in target architectures, such as **A**dvanced e**X**tensible **I**nterface (AXI), **P**rocessor **L**ocal **B**us (PLB) or **P**eripheral **C**omponent **I**nterconnect **e**xpress (PCIe). Before performing platform deployment, mistakes made when mapping the hardware IP in the target system bus may go by unnoticed, since the interface between the hardware delegate threads and the co-processors is performed by the testbench. As it is, the currently supported design flow partially helps system wide validation, but this step would enable less changes to the simulated system when being deployed into the target platform.

Also, other simulation domains could be integrated in this co-simulation environment, by introducing domain-specific tools, like a SPICE simulator. This way, full system validation would become even closer to reality, which is a very appealing feature while developing power electronics scenarios.

By adapting the developed simulation extensions to follow the FMI standard presented in section 3.5 would allow for a more portable interface, not to mention the industry acceptance. The integration of other simulators and tools into the co-simulation environment would also be easier, since there are many simulation tools in other domains that already support this standard.

The last suggestion for future work is to improve the thread model used, in order to allow direct data exchange between hardware delegate threads and reduce the overhead introduced by the device driver system calls. The use of DMAs would be a good way to improve the real-time capabilities of the co-simulation environment, since it allows peripheral components to transfer their I/O data directly to and from main memory, greatly increasing throughput to and from a device and eliminating great part of computational overhead.

Bibliography

D. Abbott, *Linux for Embedded and Real-Time Applications*, ser. Embedded technology series. Newnes, 2003.

H. Akagi, E. W. Hirokazu, and M. Aredes, *Instantaneous Power Theory and Applications to Power Conditioning*, ser. IEEE Press Series on Power Engineering. Wiley-IEEE Press, 2007.

Altera, "What is an SoC FPGA?" Architecture Brief, 2014. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ab/ab1_soc_fpga.pdf

——, "Spectra-Q Engine," Backgrounder, 2015. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/backgrounder/spectra-q-engine-backgrounder.pdf

M. Armbruster, "QEMU interface introspection: From hacks to solutions," 2015. [Online]. Available: https://drive.google.com/file/d/0BzyAwvVlQckeWW5DRldRU2tKYlU/view

F. Balducci, "OpenRisc Verilog simulation of serial port communication," 2009. [Online]. Available: https://balau82.wordpress.com/2009/12/17/openrisc-verilog-simulation-of-serial-port-communication/

Barebox.org, "The Barebox Bootloader," 2016. [Online]. Available: http://www.barebox.org/

T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, a. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J. V. Peetz, and S. Wolf, "The Functional Mock-up Interface for Tool independent Exchange of Simulation Models," in *Proceedings of the 8th International MODELICA Conference in Dresden*, no. 063. Linköping

University Electronic Press, March 2011, pp. 105–114. [Online]. Available: http://www.ep.liu.se/ecp/063/013/ecp11063013.pdf

T. Blochwitz, M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel *et al.*, "Functional Mock-up Interface 2.0: The Standard for Tool Independent Exchange of Simulation Models," in *Proceedings of the 9th International MODELICA Conference in Munich*, no. 076. Linköping University Electronic Press, November 2012, pp. 173–184. [Online]. Available: http://www.ep.liu.se/ecp/076/017/ecp12076017.pdf

DENX Software Engineering, "Das U-Boot - the Universal Boot Loader," 2016. [Online]. Available: http://www.denx.de/wiki/U-Boot

C. Ebert and C. Jones, "Embedded software: Facts, figures, and future," *Computer*, vol. 42, no. 4, pp. 42–52, apr 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5054871

Free Electrons, "Buildroot," 2016. [Online]. Available: https://buildroot.org/

M. Gries and K. Keutzer, *Building ASIPS: The Mescal Methodology*. Springer, 2005.

L. Hansen, "Unleash the Unparalleled Power and Flexibility of Zynq UltraScale+ MPSoCs," Xilinx, White Paper, June 2016, v1.1. [Online]. Available: http://www.xilinx.com/support/documentation/white_papers/wp470-ultrascale-plus-power-flexibility.pdf

S. Hauck and A. DeHon, *Reconfigurable Computing - The Theory and Pratice of FPGA-Based Computing*. Morgan Kaufmann Publishers Inc., 2008.

T. Huffmire, C. Irvine, T. D. Nguyen, T. Levin, R. Kastner, and T. Sherwood, *Handbook of FPGA Design Security*. Springer, 2010.

IEEE, "IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language," *IEEE Std. 1800^{TM}-2012*, February 2013. [Online]. Available: https://standards.ieee.org/findstds/standard/1800-2012.html

S. P. Khatri and K. Gulati, *Hardware Acceleration of EDA Algorithms: Custom ICs, FPGAs and GPUs*, 1st ed. Springer US, 2010.

I. Lee, J. Y.-T. Leung, and S. H. Son, *Handbook of Real-Time and Embedded Systems*. Chapman and Hall/CRC, 2007.

J. Liaw, "QEMU Binary Translations," 2014. [Online]. Available: http://pt.slideshare.net/RampantJeff/qemu-binary-translation

M. Marazakis, "QEMU: Architecture and Internals Lecture for the Embedded Systems Course." [Online]. Available: http://www.csd.uoc.gr/~hy428/reading/qemu-internals-slides-may6-2014.pdf

C. Maxfield, *FPGAs: World Class Designs*. Newnes, 2009, vol. 1.

A. McHoes and I. M. Flynn, *Understanding Operating Systems*, 7th ed. Course Technology, 2013.

MODELISAR, *Functional Mock-up Interface for Co-Simulation*, October 2010, version 1.0. [Online]. Available: https://svn.modelica.org/fmi/branches/public/specifications/v1.0/FMI_for_CoSimulation_v1.0.pdf

——, *Functional Mock-up Interface for Model Exchange* , January 2010, version 1.0. [Online]. Available: https://svn.modelica.org/fmi/branches/public/specifications/v1.0/FMI_for_ModelExchange_v1.0.pdf

MODELISAR and Modelica Association, *Functional Mock-up Interface for Model Exchange and Co-Simulation*, July 2014, version 2.0. [Online]. Available: https://svn.modelica.org/fmi/branches/public/specifications/v2.0/FMI_for_ModelExchange_and_CoSimulation_v2.0.pdf

Modelon, *FMI Library: part of JModelica.org*, April 2016, version 2.0.2b3. [Online]. Available: http://www.jmodelica.org/api-docs/FMIL_docs/FMILibrary-2.0.2b3.pdf

N. Naia, "Real-Time Linux and Hardware Accelerated Systems on QEMU," Master Thesis, Universidade do Minho, 2015.

T. Noergaard, *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*, 2nd ed. Newnes, 2013.

P. Oliveira, "Sistema de Aquisição de Sinais em Tempo Real Baseado em Linux," Master Thesis, Universidade do Minho, 2013.

Oprofile, "Oprofile Overview and Features," 2015. [Online]. Available: http://oprofile.sourceforge.net/about/

T. Petazzoni, "Device Tree for Dummies," October 2013, presented at Embedded Linux Conference Europe 2013. [Online]. Available: https://events.linuxfoundation.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf

Powersim, "Help on the General DLL Block in PSIM®," User Guide, October 2004. [Online]. Available: http://caxapa.ru/thumbs/591244/Help_General_DLL_Block.pdf

——, "PSIM® User's Guide," User Guide, January 2016, version 10.0, Release 5. [Online]. Available: https://powersimtech.com/drive/uploads/2016/06/PSIM-User-Manual.pdf

Red Hat, Inc., "RedBoot," 2016. [Online]. Available: https://sourceware.org/redboot/

J. Sato, M. Imai, T. Hakata, A. Alomary, and N. Hikichi, "An integrated design environment for application specific integrated processor," *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 414–417, October 1991.

V. Silva, "Sistema de Aquisição de Dados Tempo Real baseado em Linux," Master Thesis, Universidade do Minho, 2011.

C. Spear and G. Tumbush, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, 3rd ed. Springer Science+Business Media, 2012.

W. Stallings, *Operating Systems: Internals and Design Principles*, 8th ed. Pearson, 2014.

S. Sutherland, "Transitioning to the New PLI Standard," March 1998, paper presented at the 1998 IEEE International Verilog HDL Conference in Santa Clara. [Online]. Available: http://sutherland-hdl.com/papers/1998-HDLCon-paper_transitioning_to_new_PLI.pdf

——, *The Verilog PLI Handbook: User's Guide and Comprehensive Reference on the Verilog Programming Language Interface*, 2nd ed. Kluwer Academic Publishers, 2002, no. 1.

——, "SystemVerilog 3.1: The Hardware Description AND Verification Language," March 2003, paper presented at Synopsys 2003 SNUG San Jose conference.

[Online]. Available: http://sutherland-hdl.com/papers/2003-SNUG-paper_SystemVerilog.pdf

——, "The Verilog PLI Is Dead (maybe). Long Live The SystemVerilog DPI!" March 2004, paper presented at Synopsys 2004 SNUG San Jose conference. [Online]. Available: http://sutherland-hdl.com/papers/2004-SNUG-paper_Verilog_PLI_versus_SystemVerilog_DPI.pdf

S. Sutherland and D. Mills, "Synthesizing SystemVerilog: Busting the Myth that SystemVerilog is only for Verification," November 2013, paper presented at Synopsys 2013 SNUG Silicon Valley 2013. [Online]. Available: http://sutherland-hdl.com/papers/2013-SNUG-SV_Synthesizable-SystemVerilog_paper.pdf

S. Tu, "Atom™-x5/x7 series processor, codenamed Cherry Trail," January 2015, Intel Developer Forum 2015. [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.25-Tuesday-Epub/HC27.25.70-Processors-Epub/HC27.25.740-Atom-CherryTrail-Tu-Intel.pdf

J. Turley, *The Essential Guide to Semiconductors*, 1st ed. Prentice Hall, 2002.

F. Vahid and T. Givargis, *Embedded System Design - A Unified Hardware/Software Approach*. University of California, 1999.

C. Wen-Ren, "QEMU Detailed Study," 2011. [Online]. Available: https://lists.gnu.org/archive/html/qemu-devel/2011-04/pdfhC5rVdz7U8.pdf

Xilinx, "Vivado Design Suite: Logic Simulation," User Guide, June 2016, v2016.2. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug900-vivado-logic-simulation.pdf

——, "Vivado Design Suite High-Level Synthesis," User Guide, June 2016, v2016.2. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug902-vivado-high-level-synthesis.pdf

——, "7 Series FPGAs Overview," Product Specification, May 2015, v1.17. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

K. Yaghmour, J. Masters, G. Ben-Yossef, and P. Gerum, *Building Embedded Linux Systems*, 2nd ed. O'Reilly Media, 2008.

W. M. Zabołotny, "Development of embedded PC and FPGA based systems with virtual hardware," in *Proc. of SPIE Vol. 8454 84540S-1, Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*, 2012.

Appendix A

Buildroot Support

A.1 Real-time Linux Patching and Compilation with Buildroot

In this appendix, the basics on how to compile a real-time Linux kernel using Buildroot will be shown.

A.1.1 Buildroot Installation

The latest stable buildroot version can be downloaded from:

https://buildroot.org/download.html

This location also maintains stable versions of Buildroot, as well as **R**elease **C**andidate (RC) versions, although their use is not encouraged. As of this document's elaboration, the latest stable release is 2016.08 and it will be the one used throughout this appendix.

To download Buildroot, use the following command:

```
$ wget https://buildroot.org/downloads/buildroot
   -2016.08.tar.gz
```

The downloaded context is compressed, so it should be later extracted to a desired directory using:

```
$ tar -xvf buildroot-2016.08.tar.gz -C "target folder"
```

The result of the extraction is the creation of a folder called *"buildroot-2016.08"* at the target directory, with a set of makefiles and scripts that constitute Buildroot.

This does not contain the toolchain, Linux sources and other services related to target system generation, as they will be downloaded later during the system compilation, with the produced output being located at *"buildroot-2016.08/output"*.

A.1.2 Real-time Linux Compilation

Buildroot provides a set of default configurations for a wide number of supported boards. To list the supported defconfigs run:

```
$ make list-defconfigs
```

For this example, the used *defconfig* will be the as follows:

```
$ make qemu_arm_vexpress_defconfig
```

To access kernel-specific configuration, a graphical menu can be prompted using:

```
$ make linux-menuconfig
```

By accessing the kernel features, it is possible to observe that no real-time preemption is available:

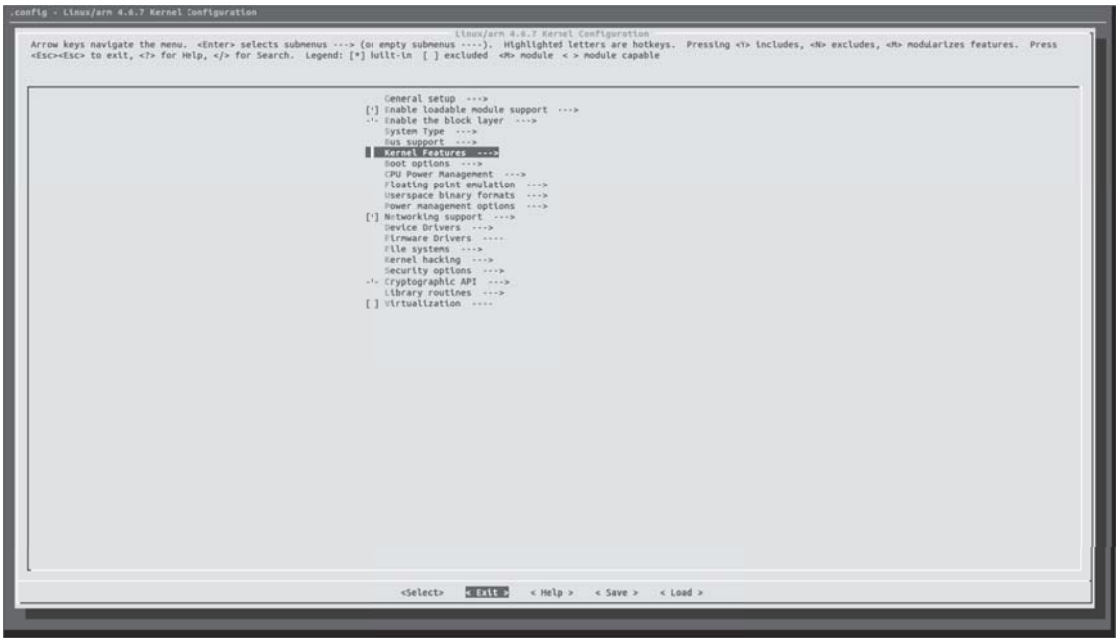


Figure A.1: Linux kernel configuration

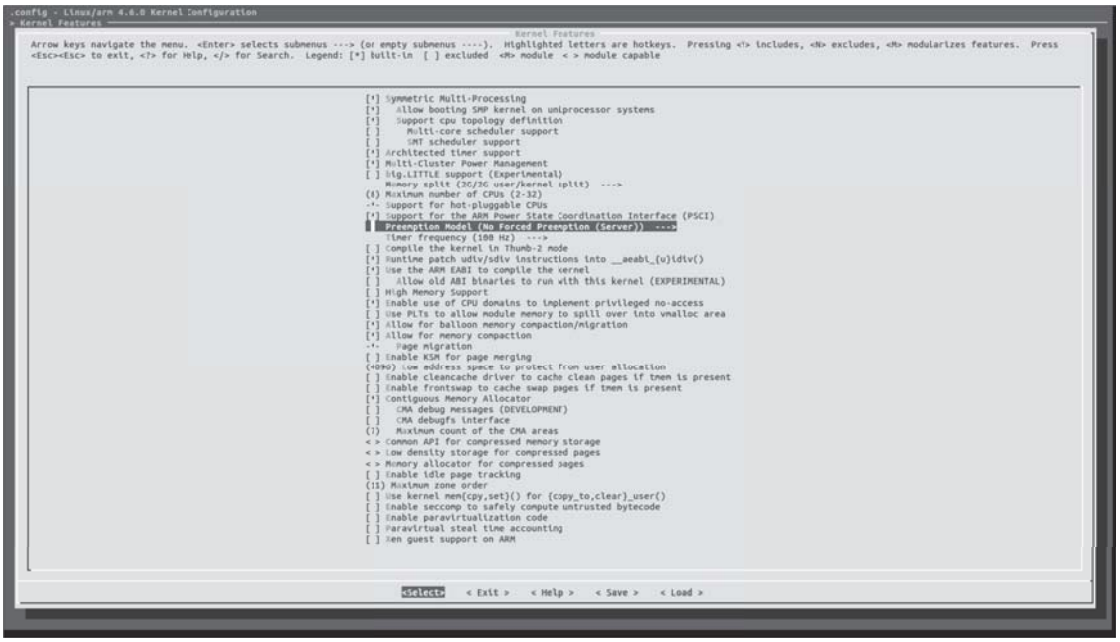


Figure A.2: Linux kernel features

Figure A.3: Linux kernel preemption mode selection

To enable real-time preemption, a patch is maintained in *kernel.org*, formally known as the RT-Preempt Patch. The latest stable version of the RT-Preempt patch can be found in:

https://www.kernel.org/pub/linux/kernel/projects/rt

Buildroot provides support for kernel patches, like the RT-Preempt Patch. To launch the Buildroot configuration menu just run:

```
$ make menuconfig
```

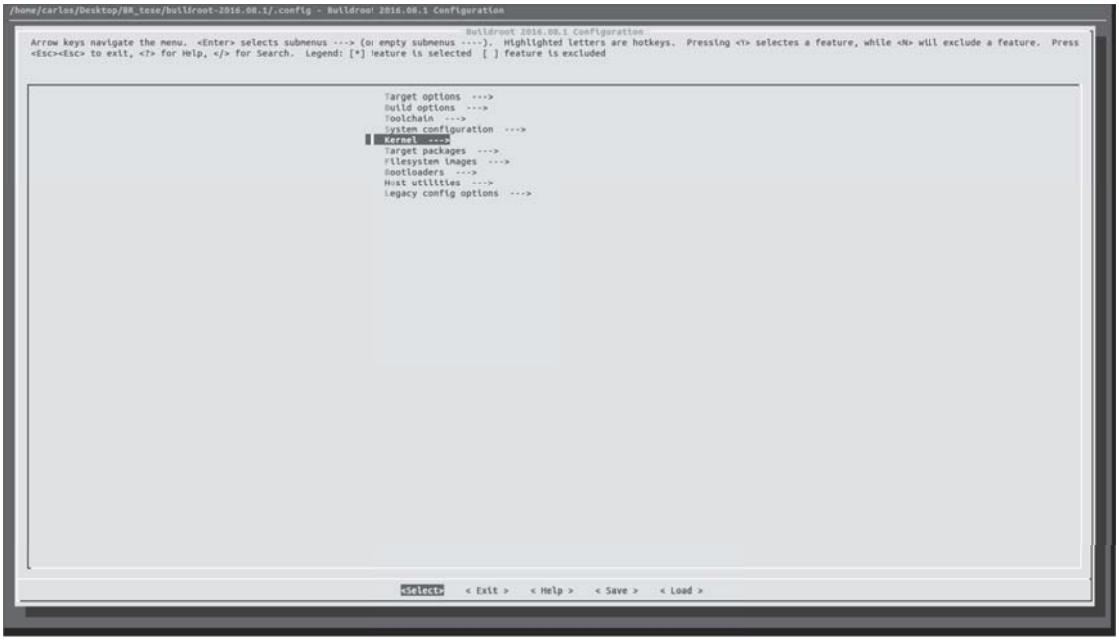


Figure A.4: Buildroot configuration

Fill the *Custom Kernel patch* field with the the *.gz* patch file URL, matching it with the Kernel version currently being used.

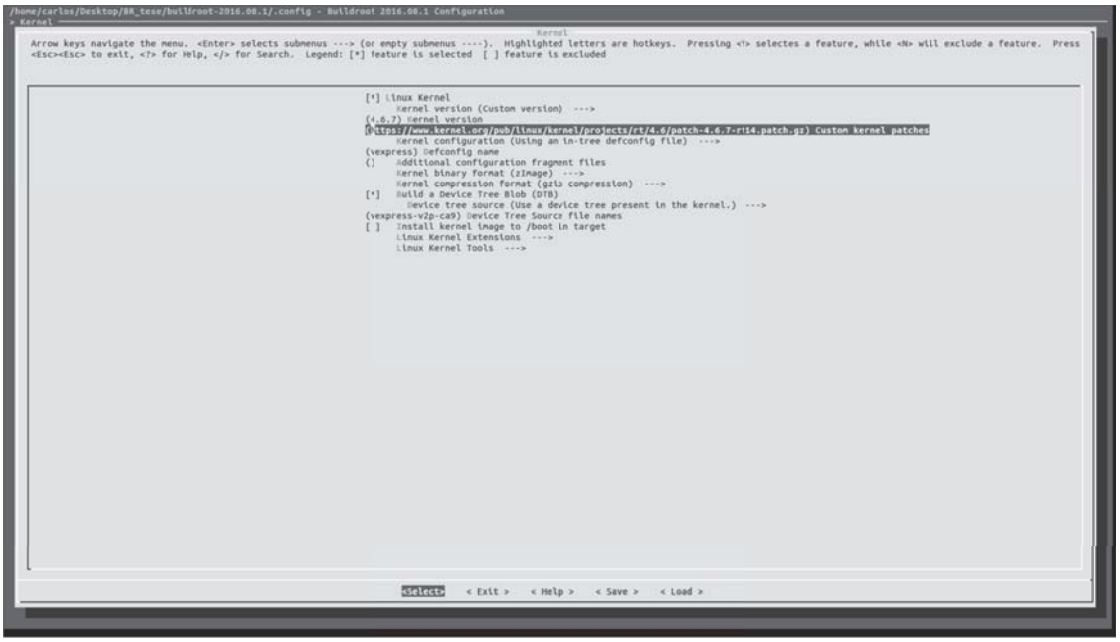


Figure A.5: Buildroot configuration of kernel patch

Afterwards, if the kernel configuration menu is accessed again, real-time preemption will be available.

Figure A.6: Linux kernel preemption mode selection: real-time preemption

Finally, to compile the kernel simply run:

```
$ make
```

Appendix B

DPI Co-Simulation Library Support Material

B.1 QEMU Monitor HW IP Property Output

```
dev: external-model, id ""
    model name = "exampleHW"
    simulation domain = "Hardware Acceleration"
    simulation tool = "Vivado Simulator"
    tool ip = "127.0.0.1"
    tool port = 34429 (0x867d)
    memory mapped address = 1409363968 (0x54013000)
    mapped area size = 32 (0x20)
    number of interrupts = 0 (0x0)
    mmio 0000000054013000/0000000000000020
```

B.2 Scripts for DPI Library: Compilation and Usage

Parsing design SystemVerilog files:

```
xvlog -prj vlog.prj -sv toplevel.sv wrapper.sv ip.v
```

Elaborate and generate design snapshot for RTL simulation along with generating the DPI header file to include in the DPI Library C layer:

```
xelab --relax --debug typical -m64 -L xil_defaultlib
       -L unisims_ver -L unimacro_ver -L secureip
       --snapshot toplevel xil_defaultlib.tb
       -log elaborate.log -dpiheader dpi.h
```

B.2.1 Script to Parse and Elaborate Design files & Compile the DPI C Layer Library

```
#!/bin/csh -xvf

xvlog -prj vlog.prj -sv toplevel.sv wrapper.sv ip.v

xelab --relax --debug typical -m64 -L xil_defaultlib
       -L unisims_ver -L unimacro_ver -L secureip
       --snapshot toplevel xil_defaultlib.tb
       -log elaborate.log -dpiheader dpi.h

make UNIX=1
```

The makefile used in this script is present in Appendix B.8.

B.3 Device Driver for Vivado Simulator HW IP External Model

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  #include <linux/device.h>
4  #include <linux/kernel.h>
5  #include <linux/fs.h>
6  #include <asm/uaccess.h>
7  #include <linux/wait.h>
8  #include <linux/interrupt.h>
9  #include <linux/irq.h>
10 #include <asm/io.h>
11 #include <linux/ioport.h>
12
13 #define  DEVICE_NAME "exampleHW"    // "/dev/DEVICE_NAME"
14 #define  CLASS_NAME  "hw-ip-example"
15 #define  BASE_ADDR   0x54013000
16 #define  MAPPED_BLOCK_SIZE  32
17
18 MODULE_LICENSE("GPL");
19 MODULE_AUTHOR("CarlosMesquita");
20 MODULE_DESCRIPTION("Qemu hardware interface device
      driver -> exampleHW");
21 MODULE_VERSION("1.1");
22
23
24 // Stores the device number - determined automatically
25 static int    majorNumber;
26
27 // The device-driver class struct pointerstatic
28 struct class*  hw_ip_class  = NULL;
29
30 // The device-driver device struct pointer
31 static struct device* hw_ip_device = NULL;
32
```

```
33 static int dev_open(struct inode *, struct file *);
34 static int dev_release(struct inode *, struct file *);
35 static ssize_t dev_read(struct file *, char *, size_t,
      loff_t *);
36 static ssize_t dev_write(struct file *, const char *,
      size_t, loff_t *);
37
38 static struct file_operations fops =
39 {
40    .open = dev_open,
41    .read = dev_read,
42    .write = dev_write,
43    .release = dev_release
44 };
45
46 static uint32_t *base_addr;
47
48 // IRQ handler
49 static  irqreturn_t irq_handler (int irq, void *dev_id)
50 {
51    return IRQ_HANDLED;
52 }
53
54 // LKM initialization function
55 static int __init dev_init(void)
56 {
57    printk(KERN_INFO "%s: Initializing LKM\n",
         DEVICE_NAME);
58
59    // Try to allocate a major number for the device
60    majorNumber = register_chrdev(0, DEVICE_NAME, &fops);
61    if (majorNumber < 0)
62    {
63       printk(KERN_ALERT "%s failed to register a major
            number\n", DEVICE_NAME);
64       return majorNumber;
65    }
```

```
66      printk(KERN_INFO "%s: registered correctly with major
            number %d\n",DEVICE_NAME, majorNumber);
67
68      // Register the device class
69      hw_ip_class = class_create(THIS_MODULE, CLASS_NAME);
70
71      if (IS_ERR(hw_ip_class))
72      {
73          unregister_chrdev(majorNumber, DEVICE_NAME);
74          printk(KERN_ALERT "Failed to register device class
                %s\n", CLASS_NAME);
75          return PTR_ERR(hw_ip_class);
76      }
77      printk(KERN_INFO "%s: device class registered
           correctly\n", CLASS_NAME);
78
79      // Register the device driver
80      hw_ip_device = device_create(hw_ip_class, NULL, MKDEV
           (majorNumber, 0), NULL, DEVICE_NAME);
81
82      if (IS_ERR(hw_ip_device))
83      {
84          class_destroy(hw_ip_class);
85          unregister_chrdev(majorNumber, DEVICE_NAME);
86          printk(KERN_ALERT "Failed to create device %s\n",
               DEVICE_NAME);
87          return PTR_ERR(hw_ip_device);
88      }
89
90      if(!request_mem_region(BASE_ADDR, MAPPED_BLOCK_SIZE,
           DEVICE_NAME))
91      {
92          device_destroy(hw_ip_class, MKDEV(majorNumber,0));
93          class_unregister(hw_ip_class);
94          class_destroy(hw_ip_class);
95          unregister_chrdev(majorNumber, DEVICE_NAME);
96          printk(KERN_ALERT "%s: Cannot acquire memory
```

```
            region\n", DEVICE_NAME);
97          return -EBUSY;
98      }
99
100     base_addr = (uint32_t *)ioremap(BASE_ADDR,
            MAPPED_BLOCK_SIZE);
101
102     if(request_irq(INTERRUPT_0, irq_handler,
            IRQF_IRQPOLL, DEVICE_NAME, NULL))
103     {
104         device_destroy(hw_ip_class, MKDEV(majorNumber,0));
105         class_unregister(hw_ip_class);
106         class_destroy(hw_ip_class);
107         unregister_chrdev(majorNumber, DEVICE_NAME);
108         iounmap(base_addr);
109         release_mem_region(BASE_ADDR , MAPPED_BLOCK_SIZE);
110         printk(KERN_ALERT "%s: Couldn't install interrupt
                handler\n", DEVICE_NAME);
111         return -EBUSY;
112     }
113     printk(KERN_INFO "%s: device created correctly\n",
            DEVICE_NAME);
114     return 0;
115 }
116
117 // LKM cleanup function
118 static void __exit dev_exit(void)
119 {
120     device_destroy(hw_ip_class, MKDEV(majorNumber, 0));
121     class_unregister(hw_ip_class);
122     class_destroy(hw_ip_class);
123     unregister_chrdev(majorNumber, DEVICE_NAME);
124     iounmap(base_addr);
125     release_mem_region(BASE_ADDR , MAPPED_BLOCK_SIZE);
126     printk(KERN_INFO "%s: Removing module\n", DEVICE_NAME
            );
127 }
```

```
128 // Device open function
129 static int dev_open(struct inode *inodep, struct file *
      filep)
130 {
131    return 0;
132 }
133
134 // Device read function, called whenever the device is
      being read from user space
135 static ssize_t dev_read(struct file *filep, char *buffer
      , size_t len, loff_t *offset)
136 {
137    int error_count, read;
138
139    char *to_send = kmalloc(len, GFP_KERNEL);
140    if(!to_send)
141    {
142       printk(KERN_ALERT "%s: could not allocate memory
         on read", DEVICE_NAME);
143       return -EFAULT;
144    }
145
146    read = ioread32(base_addr + offset);
147    sprintf(to_send, "%d", read);
148
149    error_count = copy_to_user(buffer, (char *)to_send,
      len);
150
151    kfree(to_send);
152
153    if(!error_count)
154       return len;
155    else
156       return -EFAULT;
157 }
158
159
```

```
160 // Device write function, called whenever the device is
       being written to from user space
161 static ssize_t dev_write(struct file *filep, const char
       *buffer, size_t len, loff_t *offset)
162 {
163     int value;
164     char *recv;
165 
166     recv = kmalloc(len, GFP_KERNEL);
167     if(!recv)
168     {
169         printk(KERN_ALERT "%s: could not allocate memory
              on write", DEVICE_NAME);
170         return -EFAULT;
171     }
172 
173     copy_from_user(recv, buffer, len);
174     kstrtoint(recv, 10, &value);
175     iowrite32(value, base_addr + offs);
176 
177     kfree(recv);
178     return len;
179 }
180 
181 static int dev_release(struct inode *inodep, struct file
       *filep)
182 {
183     return 0;
184 }
185 
186 module_init(dev_init);
187 module_exit(dev_exit);
```

B.4 SystemVerilog Top-Level Module Example

```
1  `timescale 1ns / 1ps
2
3  `default_nettype none
4
5  module top_level();
6
7  localparam WORD = 32;
8
9  reg clk;
10 reg rst;
11 reg write;
12 reg ready;
13 reg busy;
14
15 reg [63 : 0]addr;
16 reg [63 : 0]offs;
17 reg [WORD - 1 : 0]din;
18 reg [WORD - 1 : 0]dout;
19
20
21 int PORT = 0;
22 string IP = "";
23
24 wrapper #(.WORD(WORD))
25     ex_wrapper ( .addr(addr), .din(din), .offset(offs),
          .dout(dout), .write(write), .busy(busy), .ready(
          ready), .clk(clk), .rst(rst) );
26
27
28 /////      INTERFACE EXPORT DECLARATIONS      /////
29
30 export "DPI-C" function dpi_print;
31     function void dpi_print(input string msg);
32         $display("%t :: %s", $time, msg);
33     endfunction
```

```
34
35 export "DPI-C" function stop_sim;
36     function void stop_sim();
37         display("Stop simulation issued,stopping.");
38         $finish();
39     endfunction
40
41
42 /////     HW_ACCESS EXPORT DECLARATIONS     /////
43
44 export "DPI-C" function dpi_write;
45     function void dpi_write(input longint unsigned
          memory_mapped_address, input longint unsigned
          offset, input longint value, input int unsigned
          size);
46         //Write Transaction to IP's
47     endfunction
48
49 export "DPI-C" function dpi_read;
50     function longint signed dpi_read(input longint
          unsigned memory_mapped_address, input longint
          unsigned offset, input int unsigned size);
51         //Read Transaction from IP's
52     endfunction
53
54
55 /////     IMPORT DECLARATIONS     /////
56
57 import "DPI-C" function int qemu_register_ip(string NAME
     , longint unsigned MEMORY_MAPPED_ADDRESS, int
     unsigned MAPPED_AREA_SIZE, int unsigned
     INTERRUPTS_SIZE, string INTERRUPTS) ;
58
59 import "DPI-C" function int qemu_connect(string IP, int
     PORT);
60
61 import "DPI-C" function void qemu_raise_int(longint
```

```
    unsigned memory_mapped_address, int interrupt);

import "DPI-C" function void qemu_lower_int(longint
    unsigned memory_mapped_address, int interrupt);

import "DPI-C" context function void trans_handle();


/////     TESTBENCH     /////

initial begin
    qemu_register_ip
    (
        ex_wrapper.NAME,
        ex_wrapper.MEMORY_MAPPED_ADDRESS,
        ex_wrapper.MAPPED_AREA_SIZE,
        ex_wrapper.INTERRUPTS_SIZE,
        ex_wrapper.INTERRUPTS
        );

    qemu_connect(IP, PORT);

    #50
    rst = 1;
    #50
    rst = 0;
end


always
begin
    #100 clk = ~clk;
    trans_handle();
end

endmodule
```

B.5 SystemVerilog IP Wrapper Example

```
1  module wrapper#(parameter WORD = 32)
2  (
3      input clk,
4      input rst,
5      input [63 : 0]addr,
6      input [63 : 0]offset,
7      input [WORD - 1 : 0]din,
8      input write,
9      output [WORD - 1 : 0]dout,
10     output ready,
11     output busy
12 );
13
14     string NAME = "example_HW_IP";
15     longint MEMORY_MAPPED_ADDRESS = 64'h54013000;
16     int MAPPED_AREA_SIZE = 32'd32;
17     int INTERRUPTS_SIZE = 32'd2;
18     string INTERRUPTS = "41_20";
19
20     localparam MAPPED_REG_SIZE = MAPPED_AREA_SIZE/(WORD
         /8);
21
22     wire[WORD-1:0]mapped_registers_input[MAPPED_REG_SIZE
         -1:0];
23     wire[WORD-1:0]ip_write_enable[MAPPED_REG_SIZE-1:0];
24     reg[WORD-1:0]mapped_registers[MAPPED_REG_SIZE-1:0];
25
26      //Write access state machine
27      //states
28     localparam WAIT = 3'd0;
29     localparam READ = 3'd1;
30     localparam WRITE = 3'd2;
31     localparam READY_READ = 3'd3;
32     localparam READY_WRITE = 3'd4;
33     reg [2 : 0]state;
```

```
34      reg [2 : 0]next_state;
35
36       //Sequential state logic
37          always@(posedge clk)
38          begin
39             if(rst)
40                 state <= WAIT;
41             else
42                 state <= next_state;
43          end
44
45          //Next state logic
46          always@(*)
47          begin
48              case (state)
49                  WAIT:
50                  begin
51                      if((addr >= MEMORY_MAPPED_ADDRESS)
                           && (addr < (MEMORY_MAPPED_ADDRESS
                            + MAPPED_AREA_SIZE)))
52                          next_state = write ? WRITE :
                               READ;
53                      else
54                          next_state = WAIT;
55                  end
56                  READ:
57                     next_state = READY_READ;
58                  WRITE:
59                      next_state = READY_WRITE;
60                  READY_READ:
61                     next_state = MEM WAIT;
62                  READY_WRITE:
63                     next_state = WAIT;
64              endcase
65          end
66
67      assign dout = (state == READ || state == READY_READ)
```

```
        ? mapped_registers[offset] : 'hz;
68      assign ready = (state == READY_READ || state ==
          READY_WRITE) ? 'h1 : 'hz;
69
70
71      //IP instantiation
72      wire [WORD-1:0] _input;
73      wire [WORD-1:0] _output;
74      wire [WORD-1:0] _output_ready;
75
76      example_IP example_instance
77                  (
78                      .clock(clk),
79                      .reset(rst),
80                      .input(_input),
81                      .output(_output),
82                      .output_ready(_output_ready)
83                  );
84
85      //Port INPUT
86      assign _input = mapped_registers[0];
87      assign mapped_registers_input[0] = 'hz;
88
89      //PORT OUTPUT
90      assign mapped_registers_input[1] = _output;
91      assign ip_write_enable[1] = {WORD{_output_ready}};
92
93  endmodule
```

B.6 SystemVerilog Clarke Transformation IP

```
1  `define STATE1 4'b0000
2  `define STATE2 4'b0001
3  (...)
4  `define STATE9 4'b1000
5  `define RST    4'b1001
6
7  module clarkeTransform
8  (
9      input wire [31:0] sampleA,
10     input wire [31:0] sampleB,
11     input wire [31:0] sampleC,
12     input wire clk,
13     input wire rst,
14     input wire trigger,
15     output wire [31:0] outAlpha,
16     output wire [31:0] outBeta,
17     output wire [31:0] outZero,
18     output wire busy,
19     output wire ready
20 );
21
22 //State machine
23 reg [3:0] currState;
24 reg [3:0] nextState;
25
26 //Control flags
27 reg rstState;
28 reg saveAlpha;
29 reg saveBeta;
30 reg saveZero;
31 reg saveMul;
32 reg saveSom;
33 reg rBusy;
34 reg rReady;
35
```

```
36 // Add/Sub
37 reg [31:0] A;
38 reg [31:0] B;
39 reg add;
40 wire [31:0] S;
41
42 // Mult
43 wire [63:0] outputMul;
44 reg [31:0] inputMulA;
45 reg [31:0] inputMulB;
46 reg ce;
47
48 reg [31:0] ralpha;
49 reg [31:0] rbeta;
50 reg [31:0] rzero;
51
52 reg [31:0] auxSom;
53 reg [31:0] auxMul;
54
55 always @ (posedge clk) // update next state
56 begin
57     if(rst)
58         currState = `RST;
59     else
60         currState <= nextState;
61 end
62
63 always @(*)
64  begin
65     case(currState)
66         `STATE1:
67             nextState = (trigger) ? `STATE2 : `STATE1;
68         `STATE2:
69             nextState = `STATE3;
70         `STATE3:
71             nextState =`STATE4;
72         `STATE4:
```

```
73              nextState = `STATE5;
74          `STATE5:
75              nextState = `STATE6;
76          `STATE6:
77              nextState = `STATE7;
78          `STATE7:
79              nextState = `STATE8;
80          `STATE8:
81              nextState = `STATE9;
82         `STATE9:
83              nextState = `STATE1;
84         `RST:
85              nextState = `STATE1;
86           default:
87              nextState = `STATE1;
88     endcase
89  end
90
91 // add and sub block
92 addsub add_sub
93 (
94     .A(A),       // input wire [31 : 0] A
95     .B(B),       // input wire [31 : 0] B
96     .ADD(add),   // input wire ADD
97     .S(S)        // output wire [31 : 0] S
98  );
99
100 multiplier mult
101 (
102   .CLK(clk),       // input wire CLK
103   .A(inputMulA),   // input wire [15 : 0] A
104   .B(inputMulB),   // input wire [15 : 0] B
105   .CE(ce),         // input wire CE
106   .P(outputMul)    // output wire [31 : 0] P
107 );
108
109
```

```
110 always @(*)
111 begin
112     if(currState == `STATE1) //Wait for trigger to begin
113         begin
114             (...)
115         end
116     else if(currState == `STATE2) //C2(B+C)
117         begin
118             (...)
119         end
120     else if(currState == `STATE3) //C1(A - ResultLast)
121         begin
122             (...)
123         end
124     else if(currState == `STATE4) //save Alpha
125         begin
126             (...)
127         end
128     else if(currState == `STATE5) // C3(B-C)
129         begin
130             (...)
131         end
132     else if(currState == `STATE6) // save Beta + (A+B)
133         begin
134             (...)
135         end
136
137     else if(currState == `STATE7) // C4(A+B+C)
138         begin
139             (...)
140         end
141
142     else if(currState == `STATE8) //save Zero
143         begin
144             (...)
145         end
146
```

```
147     else if(currState == `STATE9) // finish
148         begin
149             (...)
150         end
151     else           // RST or default state
152         begin
153             (...)
154         end
155 end
156
157 //output
158 assign outAlpha = (ready == 1) ? ralpha : 0;
159 assign outBeta = (ready == 1) ? rbeta :0;
160 assign outZero = (ready == 1) ? rzero: 0;
161 assign ready = rReady;
162 assign busy = rBusy;
163
164 always @(posedge clk)
165 begin
166     if(rstState)
167     begin
168         ralpha = 0;
169         rbeta = 0;
170         rzero = 0;
171         auxSom = 0;
172         auxMul = 0;
173      // rst state
174     end
175     else
176     begin
177         if(saveAlpha) // save Alpha
178         begin
179                 ralpha = S ;
180         end
181
182         if(saveBeta) // save Beta
183         begin
```

```
184             if(outputMul[31] == 1)
185                 rbeta={10'b1111111111, outputMul
                        [31:10]};
186             else
187                 rbeta={10'b0000000000, outputMul
                        [31:10]};
188         end
189
190         if(saveZero) // save Zero
191         begin
192             if(outputMul[31] == 1)
193                 rzero={10'b1111111111, outputMul
                        [31:10]};
194             else
195                 rzero={10'b0000000000, outputMul
                        [31:10]};
196         end
197
198         if(saveSom)
199             auxSom = S;
200
201         if(saveMul)
202         begin
203             if(outputMul[31] == 1)
204                 auxMul={10'b1111111111, outputMul
                        [31:10]};
205             else
206                 auxMul={10'b0000000000, outputMul
                        [31:10]};
207         end
208     end
209 end
210
211 endmodule
```

B.7 DPI Library C Layer

```
1  static tool_info_t vivado_info = {
2     .name = "Vivado",
3     .domain = "Hardware Acceleration",
4     .models = NULL,
5     .n_models = 0
6  };
7
8  typedef struct transaction_struct
9  {
10    int state;
11    pthread_mutex_t trans_mutex;
12    pthread_cond_t trans_cond;
13    uint64_t address;
14    uint64_t offset;
15    int64_t value;
16    uint64_t size;
17 }transaction_struct;
18
19 transaction_struct trans_read;
20 transaction_struct trans_write;
21
22
23 /////     TRANSACTION HANDLER     /////
24
25 void trans_handle()
26 {
27    if(trans_write.state)
28    {
29       trans_write.state = 0;
30
31       pthread_mutex_lock(&trans_write.trans_mutex);
32
33       dpi_write(trans_write.address, trans_write.offset,
             trans_write.value, trans_write.size);
34
```

```
35        pthread_mutex_unlock(&trans_write.trans_mutex);
36    }
37
38    if(trans_read.state)
39    {
40        trans_read.state = 0;
41
42        pthread_mutex_lock(&trans_read.trans_mutex);
43
44        trans_read.value = dpi_read(trans_read.address,
              trans_read.offset, trans_read.size);
45
46        pthread_mutex_unlock(&trans_read.trans_mutex);
47    }
48 }
49
50
51 /////     HW ACCESS ROUTINES    /////
52
53 uint64_t hw_write(uint64_t memory_mapped_address,
       uint64_t offset, uint64_t value, uint32_t size,
       uint64_t time)
54 {
55    pthread_mutex_lock(&trans_write.trans_mutex);
56
57    trans_write.address = memory_mapped_address;
58    trans_write.offset = offset;
59    trans_write.value = value;
60    trans_write.size = size;
61    trans_write.state = 1;
62
63    pthread_mutex_unlock(&trans_write.trans_mutex);
64 }
65
66
67 uint64_t hw_read(uint64_t memory_mapped_address,
       uint64_t offset, uint64_t *value, uint32_t size,
```

```
    uint64_t time)
68 {
69     pthread_mutex_lock(&trans_read.trans_mutex);
70 
71     trans_read.address = memory_mapped_address;
72     trans_read.offset = offset;
73     trans_read.size = size;
74     trans_read.state = 1;
75 
76     pthread_mutex_unlock(&trans_read.trans_mutex);
77 
78     *value = trans_read.value;
79 }
80 
81 
82 /////      INTERRUPT HANDLING     /////
83 
84 void dpi_raise_interrupt(uint64_t memory_mapped_address,
       int interrupt)
85 {
86     qemu_raise_interrupt(memory_mapped_address, interrupt
        );
87 }
88 
89 void dpi_lower_interrupt(uint64_t memory_mapped_address,
       int interrupt)
90 {
91     qemu_lower_interrupt(memory_mapped_address, interrupt
        );
92 }
```

B.8 Makefile for the DPI C Layer Library and the PSIM® Library

```
1  NAME=dpi
2
3  ifdef UNIX
4  INC=$(MODEL_TECH)/include /usr/include/glib-2.0 /usr/lib
      /x86_64-linux-gnu/glib-2.0/include $(QEMU_SOURCE) $(
      QEMU_SOURCE)/include
5  INC_PARAMS = $(INC:%=-I%)
6
7  CC=gcc
8  CFLAGS = -c -fPIC -DUNIX
9  LDFLAGS = -shared -lpthread -lhash
10
11 SRC = $(wildcard *.c)
12 OBJ = $(SRC:.c=.o)
13
14 $(NAME).so: $(OBJ)
15          $(CC) $(OBJ) $(LDFLAGS) -o $@
16
17 %.o: %.c
18    $(CC) $(CFLAGS) $(INC_PARAMS) $< -o $@
19    $(CC) -MM $(INC_PARAMS) $< > $*.d
20    @cp -f $*.d $*.d.tmp
21    @rm -f $*.d.tmp
22
23 else
24
25 INC=$(QEMU_SOURCE) $(QEMU_SOURCE)/include
26 INC_PARAMS = $(INC:%=-I%)
27
28 ARCH =i686-w64-mingw32
29 CC = $(ARCH)-gcc
30 CXX = $(ARCH)-g++
31
```

```
32 CFLAGS = -shared -static -std=gnu99 -lpthread -lhash -
      lwsock32 -DWIN32
33 CXXFLAGS = -shared -std=c++11 -static -lpthread -lhash -
      lwsock32 -DWIN32
34 LDFLAGS = $(CFLAGS) -Wl,--output-def
35
36 SRC = $(wildcard *.c)
37 OBJ = $(SRC:.c=.o)
38
39 DLL = $(NAME).dll
40 DEF = $(NAME).def
41
42 $(DLL): $(OBJ)
43     @$(CC) $(CFLAGS) $(INC_PARAMS) $(LDFLAGS),$(DEF) $(
        OBJ) -o $(DLL) -lws2_32
44     @sed $(DEF) -i -e "s|_Z10RUNSIMUSERddPdS_PPvPiP|
        RUNSIMUSER = _Z10RUNSIMUSERddPdS_PPvPiP|"
45     @sed $(DEF) -i -e "s|_Z11OPENSIMUSERPKcS0_PPvPiPcS1_|
        OPENSIMUSER = _Z11OPENSIMUSERPKcS0_PPvPiPcS1_|"
46     @sed $(DEF) -i -e "s|_Z12CLOSESIMUSERPKcPPv|
        CLOSESIMUSER = _Z12CLOSESIMUSERPKcPPv|"
47     @sed $(DEF) -i -e "s|
        _Z15REQUESTUSERDATAiiiPPvPiS1_PcS2_|
        REQUESTUSERDATA =
        _Z15REQUESTUSERDATAiiiPPvPiS1_PcS2_|"
48     $(CC) $(CFLAGS) $(INC_PARAMS) $(DEF) $(OBJ) -o $(DLL)
         -lws2_32
49
50 %.o: %.c
51       $(CC) $(CFLAGS) $(INC_PARAMS) -c $< -o $@
52 endif
53
54 .PHONY: clean
55
56 clean:
57       rm -rf *.o *.def *.dll *.so *.d *.log *.pb *.dir *~
```

Appendix C

PSIM® Co-Simulation Library Support Material

C.1 QEMU Monitor Property Output for PSIM® Model

```
dev: external-model, id ""
  model name = "PSIM_HW_MODEL"
  simulation domain = "Power Electronics"
  simulation tool = "PSIM"
  tool ip = "127.0.0.1"
  tool port = 40711 (0x9f07)
  memory mapped address = 1091645440 (0x41113000)
  mapped area size = 16 (0x10)
  number of interrupts = 0 (0x0)
  mmio 0000000054113000/0000000000000010
```

C.2 Makefile for PSIM® DLL Creation

The makefile used to create the DLL is the same used for the DPI Library compilation, but should be executed using *make WIN32=1*.

The part used for the DLL compilation is presented next.

```
1  NAME=dpi
2
3  ifdef UNIX
4  (...)
5
6  else   \\ifdef WIN32
7
8  INC=$(QEMU_SOURCE) $(QEMU_SOURCE)/include
9  INC_PARAMS = $(INC:%=-I%)
10
11 ARCH =i686-w64-mingw32
12 CC = $(ARCH)-gcc
13 CXX = $(ARCH)-g++
14
15 CFLAGS = -shared -static -std=gnu99 -lpthread -lhash -
      lwsock32 -DWIN32
16 CXXFLAGS = -shared -std=c++11 -static -lpthread -lhash -
      lwsock32 -DWIN32
17 LDFLAGS = $(CFLAGS) -Wl,--output-def
18
19 SRC = $(wildcard *.c)
20 OBJ = $(SRC:.c=.o)
21
22 DLL = $(NAME).dll
23 DEF = $(NAME).def
24
25 $(DLL): $(OBJ)
26    @$(CC) $(CFLAGS) $(INC_PARAMS) $(LDFLAGS),$(DEF) $(
         OBJ) -o $(DLL) -lws2_32
27    @sed $(DEF) -i -e "s|_Z10RUNSIMUSERddPdS_PPvPiP|
         RUNSIMUSER = _Z10RUNSIMUSERddPdS_PPvPiP|"
```

```
28      @sed $(DEF) -i -e "s|_Z11OPENSIMUSERPKcS0_PPvPiPcS1_|
           OPENSIMUSER = _Z11OPENSIMUSERPKcS0_PPvPiPcS1_|"
29      @sed $(DEF) -i -e "s|_Z12CLOSESIMUSERPKcPPv|
           CLOSESIMUSER = _Z12CLOSESIMUSERPKcPPv|"
30      @sed $(DEF) -i -e "s|
           _Z15REQUESTUSERDATAiiiPPvPiS1_PcS2_|
           REQUESTUSERDATA =
           _Z15REQUESTUSERDATAiiiPPvPiS1_PcS2_|"
31      $(CC) $(CFLAGS) $(INC_PARAMS) $(DEF) $(OBJ) -o $(DLL)
            -lws2_32
32
33  %.o: %.c
34        $(CC) $(CFLAGS) $(INC_PARAMS) -c $< -o $@
35  endif
36
37  .PHONY: clean
38
39  clean:
40        rm -rf *.o *.def *.dll *.so *.d *.log *.pb *.dir *~
```

Appendix D

p-q Theory

The instantaneous active and reactive power theory, or "p-q Theory", was first proposed in 1983 by Akagi et al. (2007), and is mentioned in the beggining of chapter 5. It is based on the Clarke Transformation, which is a space vector transformation of time-domain signals from a natural three-phase coordinate system (ABC) into a stationary two-phase reference frame (α-β-0) (Naia, 2015).

The mathematical expressions used to perform the Clarke Transformation on the currents and voltages are the following:

$$\begin{bmatrix} v_0 \\ v_\alpha \\ v_\beta \end{bmatrix} = \sqrt{\frac{2}{3}} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} v_a \\ v_b \\ v_c \end{bmatrix} \tag{D.1}$$

$$\begin{bmatrix} i_0 \\ i_\alpha \\ i_\beta \end{bmatrix} = \sqrt{\frac{2}{3}} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} \tag{D.2}$$

Once the Clarke Transform is performed, the instantaneous real power (p), imaginary power (q) and zero-sequence power (p_0) may be calculated from the voltage and current transformed values using the following expression:

$$\begin{bmatrix} p_0 \\ p \\ q \end{bmatrix} = \sqrt{\frac{2}{3}} \begin{bmatrix} v_0 & 0 & 0 \\ 0 & v_\alpha & v_\beta \\ 0 & -v_\beta & v_\alpha \end{bmatrix} \begin{bmatrix} i_0 \\ i_\alpha \\ i_\beta \end{bmatrix} \tag{D.3}$$

Appendix E

PSIM Simulation Results

In this Appendix, the results of the PSIM simulation performed with a software-only controller are presented by means of the voltage/current waveforms before and after compensation, along with some measurements used for validation purposes. Since the grid frequency is 50 Hz and 500 samples per cycle were acquired, the sampling frequency used was 40 μs.

E.1 Voltage/Current Waveforms

Figure E.1 presents the waveforms of the voltages for the three-phases, which are distorted due to the non-linear current consumption by the loads.

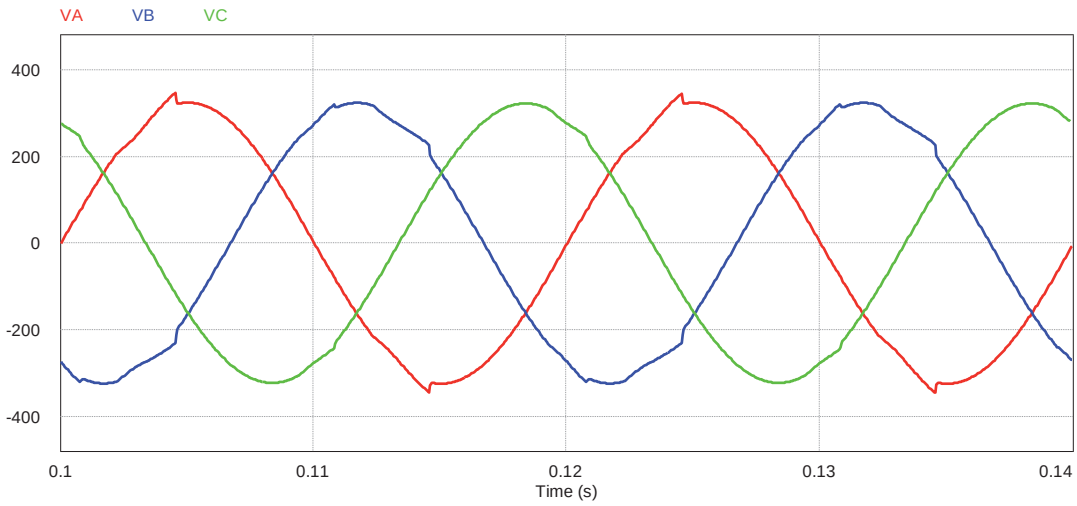


Figure E.1: Voltage waveforms for the three phases

Figure E.2 presents the waveforms of the currents for the three-phases.

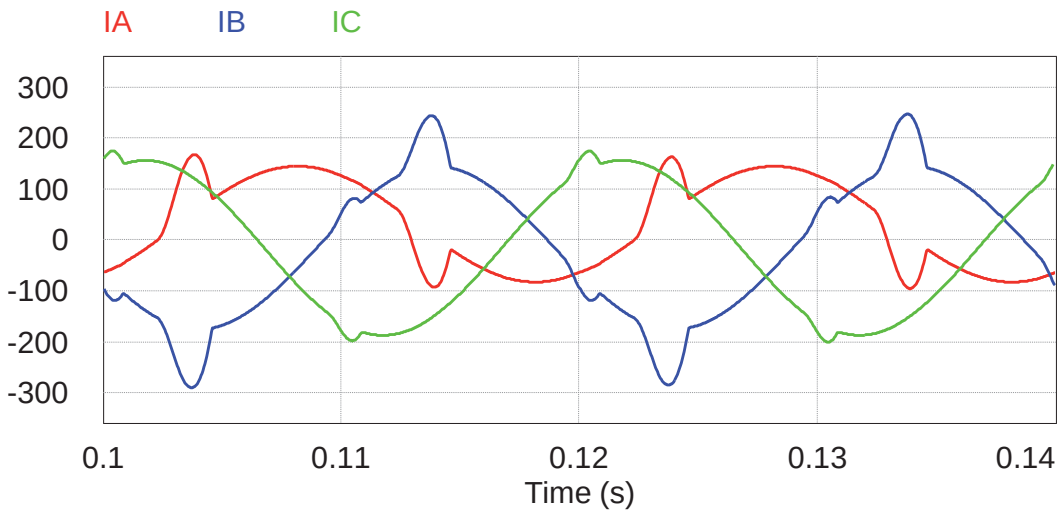


Figure E.2: Current waveforms for the three phases

In order to help the reader understanding the waveforms presented, the next graphs shown will refer only to phase A.

Figure E.3 presents the waveforms of the voltage and current for phase A, where the distortion of the current is noticeable.

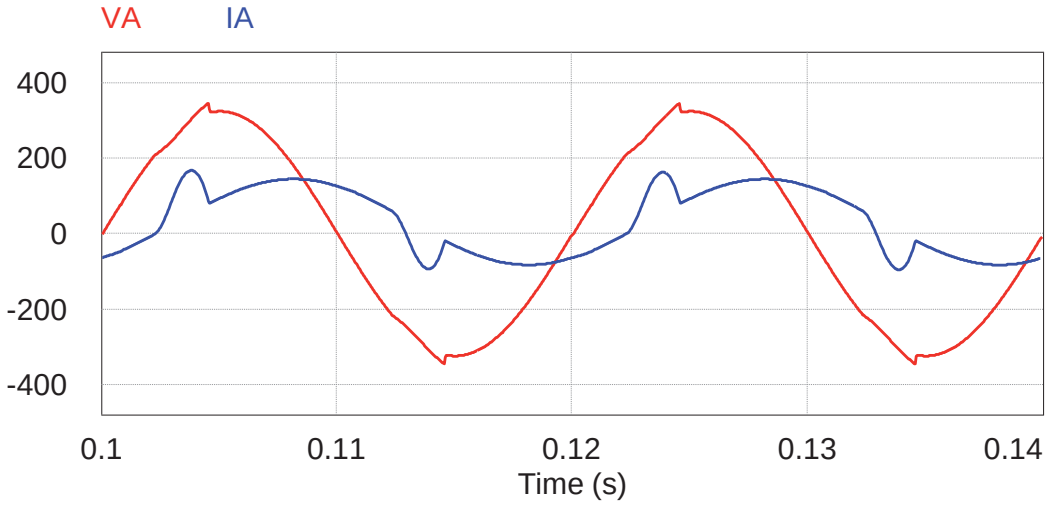


Figure E.3: Voltage/Current waveforms for phase A

Figure E.4 presents the waveforms of the voltage and current for phase A along with the calculated reference compensation current for phase A, in green.

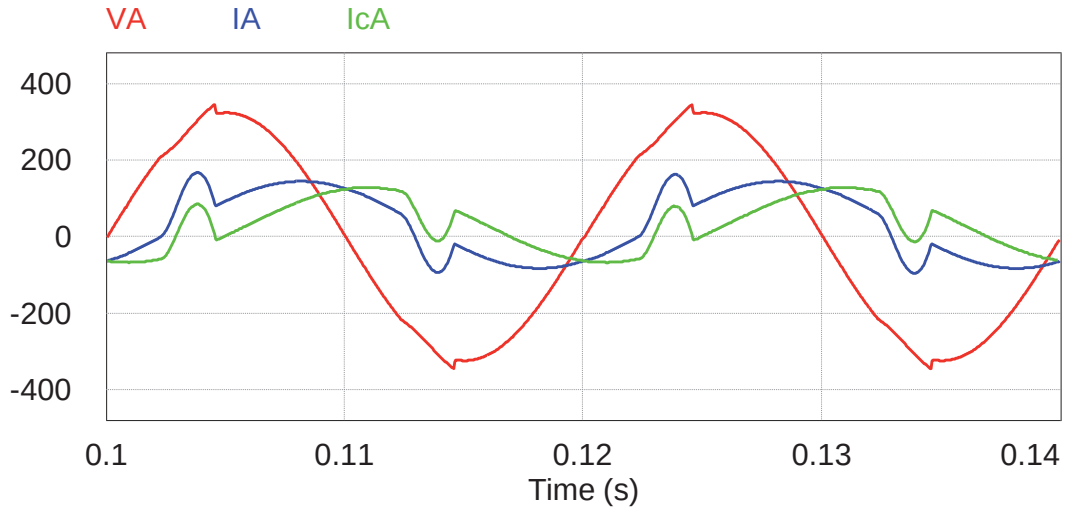


Figure E.4: Voltage/Current waveforms for phase A along with the reference compensation current for phase A

As stated by Akagi et al. (2007), the ideal compensated current can be calculated simply by subtracting the eliminated current from the load current ($i_{\text{Compensated A}} = i_{\text{Line A}} - i_{\text{Reference Compensation A}}$). Figure E.5 presents the waveforms of the voltage for phase A along with the ideal compensated current for phase A, following the above-mentioned statement.

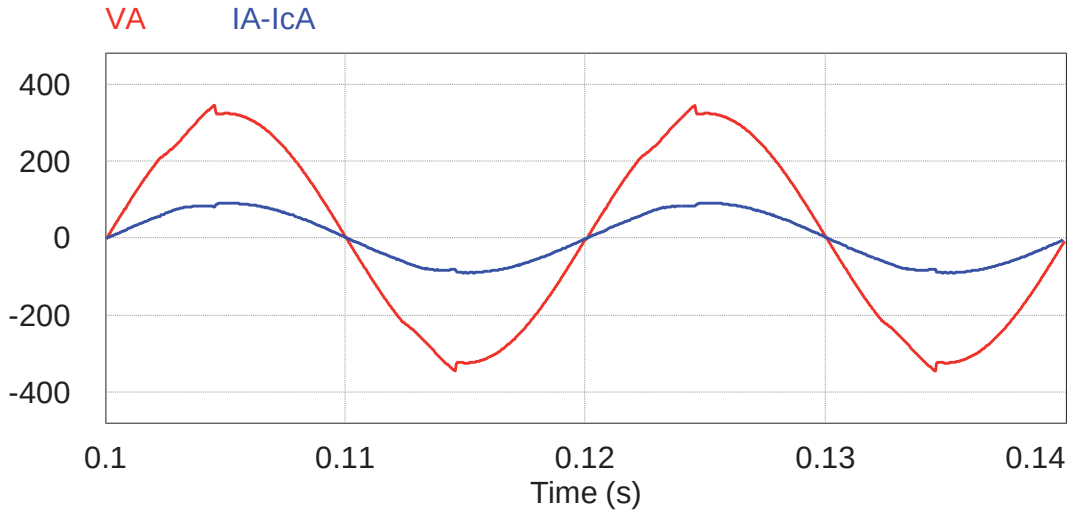


Figure E.5: Voltage and ideal compensated current waveforms for phase A

As can be seen from the current waveform in Figure E.5, it is now a lot less distorted, and apparently in phase with the voltage.

E.2 Measurements

In order to verify that the reference compensation current is being correctly compensated, and that this control algorithm actually reduces the harmonic currents presence in the installation, along with reducing the current RMS values to the possible lowest, some measurements are now presented:

Table E.1: Control Algorithm Validation

Measurement	**Before Compensation**	**After Compensation**
Line A Current THD	49,15 %	2,64 %
Line B Current THD	23,90 %	2,06 %
Line C Current THD	13,2 %	2,01 %
Line A RMS Current	91,0 A	62,0 A
Line B RMS Current	132,9 A	65,3 A
Line C RMS Current	128,8 A	64,0 A
Line A Power Factor	0,60	0,99
Line B Power Factor	0,77	0,99
Line C Power Factor	0,52	0,99

As can be verified from the table, the current THD values dropped to around 2%, which is a noticeable improvement.

The line current RMS values are now very close (62,0; 64,0; 65,3), and in some cases less than half of the values before compensation.

The power factor raised to 0,99 in the three-phases, which is also a noticeable improvement.