Hardware and Software Co-simulation

Manuel Muñoz  
Yu Cheng  
62157  61853  

Electronics Engineering  
Graduation dissertation  
May 31, 2012
HARDWARE AND SOFTWARE CO-SIMULATION

“Determination of feasibility of virtual platform usage in co-simulation techniques”

Graduation dissertation

BY

MANUEL MUÑOZ AGUIRRE

YU CHENG

Graduation dissertation submitted in partial fulfillment of the requirements for the bachelor degree of Engineering at HZ University of Applied Sciences

Date of publication: June 1, 2012

Place of publication: Venlo, Netherlands

Students: Manuel Muñoz Aguirre | 62157 muno0004@hz.nl

Yu Cheng | 61853 chen0030@hz.nl

Educational speciality: Mechatronics

Educational institute: HZ University of Applied Sciences

Academie voor Technologie & Innovatie

4380 AJ Vlissingen, NL

Company information: Océ-Technologies B.V.

5900 MA Venlo, NL

Internship mentors: J.J.W. Bastiaansen (Company)

jaccon.bastiaansen@oce.nl

C.D. de Haan (University)

cdeh@xs4all.nl
Hardware and Software Co-simulation

Document Information

Document type: Graduation report

Keywords: co-simulation, embedded system, hardware, software, virtual platform, OVP, SystemC, TLM

Date: May 31, 2012

Version: 1.0

Number of pages: 112

Revision history:

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>24-02-2012</td>
<td>Report skeleton, brief description about simulation environment and project concepts.</td>
</tr>
<tr>
<td>0.2</td>
<td>08-03-2012</td>
<td>Addition of general terminology, virtual platform structure and content related to concept implications.</td>
</tr>
<tr>
<td>0.3</td>
<td>12-03-2012</td>
<td>New content about concept implications, restructuration of first chapters, evaluation of different solution possibilities.</td>
</tr>
<tr>
<td>0.4</td>
<td>16-03-2012</td>
<td>General revision, improvements on text clarity.</td>
</tr>
<tr>
<td>0.41</td>
<td>25-03-2012</td>
<td>Grammar revision.</td>
</tr>
<tr>
<td>0.5</td>
<td>30-03-2012</td>
<td>Modification of concept evaluation chapter.</td>
</tr>
<tr>
<td>0.6</td>
<td>26-04-2012</td>
<td>Restructuration of complete report, new content added regarding technical details.</td>
</tr>
<tr>
<td>0.7</td>
<td>01-05-2012</td>
<td>General revision, improvements on text clarity.</td>
</tr>
<tr>
<td>0.8</td>
<td>22-05-2012</td>
<td>Added final chapters, references. Version for internal revision.</td>
</tr>
<tr>
<td>0.9</td>
<td>28-05-2012</td>
<td>Added TLM appendix.</td>
</tr>
<tr>
<td>1.0</td>
<td>31-05-2012</td>
<td>Final version.</td>
</tr>
</tbody>
</table>

Distribution overview:

<table>
<thead>
<tr>
<th>Student</th>
<th>Chapters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manuel Muñoz Aguirre</td>
<td>Summary, Preface, 1.1, 1.3, 1.4, 3, 3.1, 3.2, 4, 4.1, 4.2, 6, 6.1, 6.2, 6.4, 6.5, 6.6, 7, 7.2, 7.3, 8, 10, 11, 14</td>
</tr>
<tr>
<td>Yu Cheng</td>
<td>Index, Preface, 1, 1.2, 2, Figures of Ch.4, 5, 5.1, 5.2, 5.3, 5.4, 5.5, 6.3, 6.7, 7.1, 7.4, 8.1, 9, 12, 13, 14</td>
</tr>
</tbody>
</table>

The writers were enabled by Océ-Technologies B.V. to perform research that partly forms the basis for this report. Océ-Technologies B.V. does not accept responsibility for the accuracy of the data, opinions and conclusions mentioned in this report, which are fully for the account of the writers.
Summary

With the increasing complexity of functionality and architecture in electronic systems, design and verification tasks are demanding more time in a product’s cycle as much of the testing has to be done in a manual way: when working directly with a hardware platform, tracing errors in software is a slow and difficult process.

Hardware and software co-simulation is a technique that provides a way of virtualizing a hardware platform with the purpose of enabling a better behavioural visibility, thus providing an easier process of software verification. It allows the designer to have a clear overview of what is happening in the platform, and because of this, tracing and simulating errors to verify the software’s interaction with the hardware peripherals becomes easier. Additionally, co-simulation techniques can be used to design software when there is only a working specification of all the physical modules in a platform but the hardware implementation is not yet ready, so the software coding can be started without having to wait for the hardware release.

The main reason for effectuating this research is to prove the feasibility of using virtual platforms for error tracing and software debugging purposes, and because of this, the objective is to analyse the fundamental implications of a hardware and software co-simulation environment: the general structure of a virtualised platform is explained for later discussing various approaches for achieving this task, together with the considerations implicit in the construction process. To demonstrate the benefits and constraints that a designer has to face when deciding for a co-simulation approach, a proof of concept implementation of a virtual platform is developed with the intention of having a more realistic feel of this technique.
# Table of Contents

**DOCUMENT INFORMATION** ........................................................................................................ III

**SUMMARY** ............................................................................................................................... IV

**TABLE OF CONTENTS** ................................................................................................................ V

**PREFACE** .................................................................................................................................. IX

1 **INTRODUCTION** ...................................................................................................................... 1

1.1 CONCEPTUAL OVERVIEW ................................................................................................. 1

1.1.1 Platform structure .............................................................................................................. 1

1.1.2 Levels of abstraction ........................................................................................................ 1

1.1.3 Perception of time ............................................................................................................. 2

1.2 PROBLEM DEFINITION ....................................................................................................... 3

1.3 RELEVANCE ......................................................................................................................... 4

1.4 GUIDE FOR THE READER ................................................................................................. 4

2 **GENERAL TERMINOLOGY AND ENVIRONMENT STRUCTURE** ........................................... 6

3 **OPEN VIRTUAL PLATFORMS SIMULATOR** ........................................................................... 8

3.1 PLATFORM STRUCTURE ....................................................................................................... 8

3.1.1 ARM IntegratorCP from a general perspective ................................................................. 8

3.1.2 ARM IntegratorCP from the C perspective .................................................................. 10

4 **CO-SIMULATION APPROACHES** ...................................................................................... 14

4.1 OVPsim as main simulator .................................................................................................. 14

4.2 Non-OVPsim as main simulator ........................................................................................ 15

5 **PSE ORIENTED CONFIGURATIONS** .................................................................................. 17

5.1 INTERCEPTIONS ................................................................................................................... 17

5.2 MODELING APIs .................................................................................................................. 18

5.3 BASIC DESIGN ELEMENTS ................................................................................................. 19

5.4 INSTANCING PERIPHERALS FROM THE PLATFORM ......................................................... 20

5.5 GENERAL OVERVIEW .......................................................................................................... 20

6 **SYSTEMC TLM 2.0 ORIENTED CONFIGURATIONS** ............................................................. 21

6.1 A PRIMER ON SYSTEMC ...................................................................................................... 21

6.1.1 Structural design ............................................................................................................. 22

6.1.2 About concurrency ......................................................................................................... 23

6.1.3 Example ......................................................................................................................... 24

6.2 DESIGN LEVELS IN SYSTEMC .......................................................................................... 25

6.3 TLM 2.0 .................................................................................................................................. 28

6.3.1 Structure ......................................................................................................................... 28

6.3.2 Example ......................................................................................................................... 30

6.4 CONVERSION FROM HDL TO TLM .................................................................................... 33

6.4.1 Manual recoding ............................................................................................................. 33

6.4.2 HIFSuite ......................................................................................................................... 33

6.4.3 Carbon Model Studio .................................................................................................... 35

6.5 FORMAL EQUIVALENCE BETWEEN HDL AND TLM ......................................................... 35

6.6 SPEED CONCERNS RELATED TO TRANSLATION ALGORITHMS .................................. 35

6.7 GENERAL OVERVIEW ......................................................................................................... 35
List of tables

TABLE 5.1: BHM and PPM features .............................................................................................................. 18
TABLE 6.1: Design levels in SystemC ......................................................................................................... 25
TABLE 6.2: Basic attributes of a generic payload ...................................................................................... 29
TABLE 7.1: Datatype comparison ............................................................................................................. 40
TABLE 9.1: Cosimulation concepts EDS ................................................................................................... 45

List of figures

Figure 1.1: Levels of abstraction ................................................................................................................... 2
Figure 1.2: Typical Océ printer platform ..................................................................................................... 3
Figure 2.1: General co-simulation structure ................................................................................................. 6
Figure 3.1: ARM IntegratorCP simulated platform structure ..................................................................... 8
Figure 3.2: Simulation launch ..................................................................................................................... 9
Figure 3.3: Simulation statistics .................................................................................................................. 9
Figure 3.4: Time flow ................................................................................................................................ 10
Figure 4.1: Concept A ................................................................................................................................. 14
Figure 4.2: Concept B ................................................................................................................................ 14
Figure 4.3: Concept C ................................................................................................................................ 15
Figure 4.4: Concept D ................................................................................................................................ 15
Figure 4.5: Concept E ................................................................................................................................ 16
Figure 5.1: Concepts A and B .................................................................................................................... 17
Figure 5.2: The PSE toolchain ................................................................................................................... 17
Figure 5.3: Callbacks .................................................................................................................................. 18
Figure 6.1: Concepts C and D .................................................................................................................... 21
Figure 6.2: Abstraction levels language comparison ................................................................................. 21
Figure 6.3: Element summary ..................................................................................................................... 23
Figure 6.4: Execution phases ...................................................................................................................... 24
Figure 6.5: TLM 2.0 protocol ...................................................................................................................... 28
Figure 6.6: TLM elements .......................................................................................................................... 28
Figure 6.7: HIFSuite Translation Flow ....................................................................................................... 33
Figure 6.8: HIFSuite's construct support ................................................................................................... 33
Figure 6.9: HIFSuite VHDL to SystemC translation attempt ..................................................................... 34
Figure 7.1: Concept E .................................................................................................................................. 37
Figure 8.1: Cadence Virtual System Platform ............................................................................................ 42
Figure 8.2: Synopsis Virtualizer ................................................................................................................ 43
Figure 10.1: File structure ........................................................................................................................... 47
Figure 10.2: PSE UART execution process ................................................................................................. 48
Figure 10.3: PSE UART Callback triggering ............................................................................................. 48
Figure 10.4: Time flow in the PSE UART ................................................................................................... 49
Figure 11.1: ICM Platform execution ......................................................................................................... 50
Figure 11.2: Simulation launch ................................................................................................................... 51
Figure 11.3: UART driver ........................................................................................................................... 51
Figure 11.4: Overflow test case ................................................................................................................. 53
Figure 12.1: TLM Platform ........................................................................................................................ 54
Figure 12.2: Bus interface .......................................................................................................................... 55
Figure 12.3: Interrupt interface .................................................................................................................. 56
Figure 12.4: TLM UART execution flow ................................................................................................... 56
Figure 12.5: Blocking transport ................................................................................................................ 56
Figure 12.6: Overflow ................................................................................................................................. 57
Figure 13.1: Simulation execution flow ..................................................................................................... 58
Figure 13.2: Simulation launch ................................................................................................................... 59
List of Appendix figures

Figure A1: ARM Bare Metal platform simulation structure .............................................................. 63
Figure A2: Simulation statistics ...................................................................................................... 64
Figure A3: Cross-compiled test program ...................................................................................... 66
Figure A4: Enabling HDL UART driver ....................................................................................... 88

Abbreviations

API Application Programming Interface
BHM BeHavioural Modelling
EDA Electronic Design Automation
EDS Engineering Design Specification
FPGA Field-Programmable Gate Array
GCC GNU Compiler Collection
HDL Hardware Description Language
ICM Innovative CPUManager Interface
MIPS Millions of Instructions Per Second
MMC Memory Management Controller
MSYS Minimal SYstem
OS Operating System
OSCI Open SystemC Initiative Simultor
OVP Open Virtual Platform
PPM Peripheral Programming Modelling
PSE Peripheral Simulation Engine
RAM Random Access Memory
ROM Read Only Memory
RTL Register Transfer Level
TLM Transaction Level Modelling
UART Universal Asynchronous Receiver/Transmitter
VHDL VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
VHSIC Very High Speed Integrated Circuit
VMI Virtual Machine Interface
VSP Virtual System Platform
Preface

This dissertation was developed throughout a period of 4 months by Manuel Muñoz and Yu Cheng as the culmination of HZ University of Applied Sciences double degree programme of B.Eng.

The assignment was conducted at Océ-Technologies B.V., in the Research and Development facilities in Venlo, Netherlands. Océ develops, manufactures and sells printing and copying hardware and related software for the professional market. The product folio includes office printing and copying systems; production printers and wide format printing systems for both technical documentation and colour display graphics. The company was founded in 1877. With headquarters in Venlo, Netherlands, Océ is active in over 100 countries and employs more than 20,000 people worldwide.

We would like to express our gratitude to all the people who offered their kind advice and support during our assignment, to our in school mentor C.D. de Haan for advising us on graduation matters, and especially to our in-company mentor J.J.W. Bastiaansen who had a genuine interest in our learning process, providing us with very useful feedback and guidance on our research, ranging from technical issues to communication skills, and also by asking in depth questions on all the aspects of the project. This motivated us to keep a high quality in the content we developed.

Additionally, we state our thanks to the OVPworld.org team for supporting our project by providing an extended unconditional license for OVPsim during the assignment’s length.

Both of us have learnt a lot from this assignment. And we can look back to a successful and very educational period at Océ.

Yu Cheng and Manuel Muñoz
Venlo, May 2012
1 Introduction

With the increasing complexity of hardware systems, software engineers have to face the task of debugging their applications with limited behavioural visibility and few alternatives for error tracing. Additionally, there is a need for testing programs when the hardware modules are not yet in a finished state in order to make adequate changes for the applications to be ready at the moment of hardware deployment, reducing the overall time to market.

This research focuses on analysing how to tackle these issues via hardware and software co-simulation, which is a technique that virtualises a hardware platform at a high level of abstraction on which software can be implemented, without the latter realising that it is running on a simulation instead of the real hardware. An easier way to think of the co-simulation idea is a hardware board being simulated on its entirety by a computer at a certain level of detail. To fully comprehend the goals of this research, a general overview of the co-simulation technique is presented.

1.1 Conceptual overview

1.1.1 Platform structure

A platform can be defined, according to Chen et al. (2009) as: “an assembly of components that constitute a design, containing computational blocks that carry out appropriate operations but also communication components that are used to interconnect the functional components”. Platforms must include at least a processor, memory and data/instruction buses, but generally they also have other functional blocks such as UARTs, LCD/VGA controllers, interrupt controllers, etc.

Processor vendors are willing to provide models for their IP Cores, and there is a considerable set of emulators for executing the whole platform simulation. For the embedded design engineer, this means that the software application can be deployed on a co-simulated platform with the same functionality as if it was the real prototype. This leads to time savings on debugging issues as there is a broader visibility of the behaviour and relationship between each of the components on the platform; creating test cases becomes easier. Virtual platforms can be as simple as a bare-metal platform (no OS) making use of Instruction Set Simulator (ISS) features, or complex to the point of running an OS in a co-simulated way.

HW/SW co-simulation platforms generally do not address power issues (although methods for energy consumption are moderately documented). Instead, the current EDA industry efforts are focused on dealing with the abstraction levels used to implement a platform.

1.1.2 Levels of abstraction

When choosing an approach to build a co-simulation platform, it is vital to identify the abstraction levels that are involved in the design and analyse the payoff in speed. These vary in complexity and methods of data transfer. For example, Register Transfer Level (RTL) descriptions such as those designated by VHDL and Verilog components work at a lower level by declaring all the signals and behaviours necessary to achieve a workable design, namely clock signals, pin by pin definitions and related information. Other approaches such as Transaction Level Modelling (TLM) work at a system level by describing the functional behaviour without lowering to gate detail as RTL does. Mixed abstraction approaches that work at a level between RTL and TLM also exist, such as SystemC which is an event-driven set of classes that sit on top of C++.
The abstraction level will have a direct impact on the performance of a simulation; an implementation at low level demands more resources to execute, as it can be observed on Figure 1.1. These topics will be discussed in detail on upcoming chapters.

At the moment of choosing a simulation tool, the designer must also identify how to treat instructions: some environments are prepared for working with cycle accurate models which represent state by state changes in pipelines and signals. Instruction accurate models skip this thorough definition by just creating communication channels and models at a higher abstraction level, focusing on obtaining deterministic results.

Engineers that want to use co-simulation have to deal with issues regarding abstraction level mixture: most of the current designs in the technology industry are already deployed at a RTL level. And with the advantages that methodologies as TLM provide, it could be desirable to build a platform where models coded at different levels of abstraction can interact.

The inquisitive reader may wonder how the simulation speed is affected by mixing several of the design approaches. It would be reasonable to say that the whole simulated platform will be slowed down at the speed of the RTL designs. If the latter contains TLM modules, their working speed, even though they are defined at a functional level, will be slowed down to the frequencies of RTL components.

1.1.3 Perception of time
Because of all the different aspects that must be taken into account for assessing the performance of a simulation, it is convenient to have more than one perception of time. As Black and Donovan (2010) suggest, it is easy to identify three different ways of viewing elapsed time:
Hardware and Software Co-simulation

- **Wall clock time**: refers to the delta of time between the first call in the simulated platform and its termination at the moment that execution stops.
- **Processor time**: time that the actual simulation runs, which of course must be less than or equal to wall clock time.
- **Simulated time**: notion of time inside the simulation, which can vary respect the wall clock time (may be less or more).

To get a clearer understanding, consider the following example: a small test program calculates $n$ numbers of the Fibonacci series in 10 ms of simulated time, where the user perceives this as 2 seconds on a conventional clock, but it actually takes the processor only 1 second to execute this because of other application calls that must be addressed inside the simulation and that are actually not part of it.

The current challenge lies exactly at this point: how feasible is it to mix instruction accurate models with cycle accurate models and still have an acceptable time performance? Is it convenient to co-simulate an environment where different levels of abstraction are integrated, and at what orders of magnitude are the designers able to scale their simulations? With the usage of open source tools and suites from the large EDA companies, it is possible to delve into these promising but relatively unexplored areas of technology design.

### 1.2 Problem definition

For controlling an Ocè printer, it is necessary to have a certain amount of hardware, software and VHDL code working together. The typical Ocè printer platform (Figure 1.2) is composed of a CPU board, I/O board and FPGA.

![Figure 1.2: Typical Ocè printer platform](image)

Because such configurations have many features, it is necessary to perform quite a lot of tests to check if the system is working successfully after each hardware, software or VHDL change, therefore this process can prove to be a very tedious task.
For this reason, the objective of this assignment is to determine the feasibility of speeding up these tests via co-simulation by constructing a proof of concept platform.

1.3 Relevance
Because testing different features requires a lot of manual effort after each hardware, software or VHDL change, it is desired to find a way to accelerate the rate of effectuating these tests. It might be possible to model the combination of hardware and VHDL code and then simulate these models. With a successfully co-simulated platform, time and cost investments can be expected to decrease for testing the combinations of elements.

Main research question:
“Is it feasible to model company hardware and VHDL designs as well as their interaction with software and run these models on existing simulation tools? If not, what other alternatives are available?”

Sub questions:
- Does the initially proposed simulation tool, namely “Open Virtual Platform (OVP) simulator” meet with all the requirements to be able to simulate every combination of hardware, software and VHDL code?
- Is it possible to use already existing RTL models in a co-simulated platform? If not, what alternatives are available for implementing these descriptions?
- Are simulations repeatable, including erroneous behaviour?
- What costs does the simulation tooling involve?

1.4 Guide for the reader
It is assumed that the reader possesses fundamental knowledge about software and hardware design, computer architecture and terms related to the electronics design industry. This research was effectuated under the following methodology:

- Introductory phase: description of the assignment and related information.
  - Chapter 1: definition of hardware and software co-simulation. Presents the assignment’s objectives and research questions.

- Definition phase: overview on the structure of the co-simulation methodology.
  - Chapter 2: general terminology and environment structure necessary to understand how a virtual platform is composed and executed.
  - Chapter 3: basic insight of how the OVP simulator works.

- Research phase: different possibilities on how to structure a co-simulation platform.
  - Chapter 4: presentation of six different concepts on how to achieve a co-simulated platform.
  - Chapter 5: analysis of two approaches based on the Peripheral Simulation Engine (PSE).
  - Chapter 6: discussion of two SystemC oriented approaches.
  - Chapter 7: implications of a mixed language approach using ModelSim.
  - Chapter 8: solutions involving simulator suites from EDA companies.
• **Concept definition phase:** evaluation of the configurations discussed on the previous phase.
  ○ Chapter 9: summarization of the advantages and disadvantages of each approach via an Engineering Design Specification (EDS).

• **Implementation phase:** technical information on how to deploy a proof of concept virtual platform using an in-company peripheral specification.
  ○ Chapter 10: implementation of a UART peripheral using PSE.
  ○ Chapter 11: implementation of a virtual platform with OVP’s native APIs.
  ○ Chapter 12: implementation of a UART peripheral using TLM.
  ○ Chapter 13: implementation of a virtual platform with SystemC TLM 2.0.

• **Review phase:** final discussion of the outcomes of the research.
  ○ Chapter 14: conclusions and recommendations of the implemented platforms, discussion of general implications in co-simulations and determination of current feasibility for adoption in a large project.

On some chapters the reader can find boxes with code demonstrations, such as the following. The colour scheme used is:

- **Green** for reserved words.
- **Red** for string sequences and libraries.
- **Blue** for numerical constants and escape sequences.
- **Gray** for comments.

In-text notes are provided in a numbered way, and resolved at the end of the chapter.
2 General terminology and environment structure

For understanding the scope of the research and how hardware and software co-simulation works, it is of utmost importance to distinguish between the following concepts. Simulation construction-related definitions are, in order of encapsulation (from lowest to highest):

- **Simulator**: application which provides an environment in which a simulation can be executed based on a platform description.

- **Semihosting**: interception by the simulator of application I/O function calls and the passing of these calls to the host operating system. This feature provides an easy way of using the host computer’s resources as if they were part of the simulation per se, for example, using the host computer keyboard and LCD as the simulation’s keyboard and LCD.

- **Platform**: software abstraction of a complete hardware board comprising a collection of model instances such as processors, memory, buses, peripherals and their attributes.

- **Model**: software abstraction of a processor, processor family or other system components such as peripherals. Processors execute object code instructions and peripherals provide functionality that interacts with the OS and applications. They are instanced in the platform.

- **Instance**: running copy of a model that holds its state.

A distinction must be made between the types of processor models:

- **Cycle accurate processor model**: represents the implementation details of a processor including its pipelines and cycle by cycle state changes.

- **Instruction accurate processor model**: represents the functionality of a processor’s instruction execution without regard to artefacts like pipelines, only instruction boundaries are visible.

Modelling can be done on different levels of abstraction, the two most popular approaches on the world of simulation and verification are:

- **RTL**: Register Transfer Level refers to the level of abstraction for modelling gate level operations and digital circuits with data flow between hardware registers.

- **TLM**: Transaction Level Modelling, high level of abstraction modelling that focuses on functionality and speeds up simulation time because it does not define unnecessary low level details of communication. There are different ranges of speed advantage vs. RTL models depending on the timing model used. This is discussed in detail on Chapter 6.
In this research, OVP\textsuperscript{1} (Open Virtual Platforms) technology will be used as a base tool for co-simulating hardware and software. Four main elements make up the OVP suite:

- **OVP simulator (OVPsim):** provides the ability to run platforms which have processor and peripheral model definitions using an instruction-accurate approach, generating speed measurements in MIPS (Millions of Instructions Per Second). OVPsim dynamically translates target instructions to x86 host instructions as it is a binary translation simulator engine (Just-In-Time Code Morphing). OVPsim can load a model once and then concurrently simulate several instances of it without interference between them; it is accurate enough for software development as it is an instruction-accurate environment, for this reason it is not intended for power analysis.

- **Open source models:** a considerable collection of models are available for download on the OVP website, which include pre-compiled object code as well as source files. Examples of these are processor families (ARC, ARM, MIPS, etc.) and different types of system components such as RAMs, ROMs and caches, among others. Also available for download are compilations of pre-configured platforms that can run application software on them. One of the advantages of OVP models is that they can be connected to other environments through a SystemC wrapper, providing capabilities for a mixed-language implementation.

- **Modeling APIs:** four main APIs are provided, ICM (Innovative CPUManager Interface, for platform and simulation definition), VMI (Virtual Machine Interface, for processor modelling), BHM (BeHavioural Modelling, for peripheral modelling), and PPM (Peripheral Programming Modelling, for making connections between peripherals and the hardware platform). The platform is designed by using functions provided by these C-based APIs, with possibilities available to use C++ for defining a SystemC platform.

- **Peripheral Simulation Engine (PSE):** allows the implementation of peripheral models and interception libraries to enable interaction with the native host. Each instance of a peripheral model runs with its own “PSE processor”, so this processor and its memory are separated from other elements and buses in the actual platform. In other words, this PSE processor exists only to execute the code of the peripheral model, but its resources are not visible to other members of the platform.

Other general concepts that are important to bear in mind are:

- **ModelSim:** Simulator from Mentor Graphics that provides a mixed debug and simulation environment via a unified workspace that allows instances of VHDL, SystemC and Verilog components.
- **SystemC:** Collection of C++ libraries that enable event-driven simulation.
- **OSCI Simulator:** Open SystemC Initiative simulator, open source proof-of-concept simulator for executing SystemC implementations.

**Notes**

\textsuperscript{1}http://www.ovpworld.org
3 Open Virtual Platforms simulator

To be able to identify the magnitude and implications of a simulator, an overview about the way OVPsim works is briefly presented in this chapter before proceeding to examine approaches on how to build a complete co-simulation solution. In-detail explanations and an installation log of OVPsim are available on Appendix I.

3.1 Platform structure

3.1.1 ARM IntegratorCP from a general perspective

As the assignment requires running a Linux kernel on the processor model, exploring an example platform proves to be very useful in order to understand how a general simulation setup is structured. IntegratorCP is a development board that has one core and provides connectors for certain peripherals. Three files are required to launch a simulation that boots Linux:

- **Platform file**: defines how the models are connected inside the platform, defines address spaces for each model and sets attributes for the processor and paths for file locations. Compiled and linked into an executable file.
- **zImage**: compressed Linux kernel.
- **fs.img**: Root filesystem required by the Linux kernel.

Applications can be compiled and saved into the root filesystem. OVP provides two utilities, `packImage.sh` and `unpackImage.sh` to aid in this task. After building the required files, the simulation can be run by:

```
./platform_name.exe zImage fs.img
```

Because this is a more complicated setup than a bare-metal platform, the required peripherals to boot an OS need to be defined in the platform. It is important to note that these must be created by the user or downloaded from OVP’s site if available. The reference to location of these peripherals must be done inside the platform’s definition. The figure below represents the conceptual organization of the ARM IntegratorCP simulated platform from the user’s point of view.

![Figure 3.1: ARM IntegratorCP simulated platform structure](image)
Each peripheral is running with its own PSE processor, so memory-mapped I/O connections between them and the platform’s processor must be defined. This image shows an LCD screen peripheral with the boot process output.

After the simulation exits, statistics about instruction count and time are presented on the host system. The time results are related to the concepts presented on section 1.1.3. Elapsed time is a synonym of wall clock time. OVP splits the concept of processor time in two: system time is the time that the host computer spends executing instructions related to the simulation execution but that are actually not part of the simulation and user time is the time that the simulation executes instructions on the host computer. Real time ratio refers to the division of simulated time by elapsed time.
The diagram below summarizes the definitions introduced about the time perceptions. The small green line represents a period of time that the processor attends other tasks; this makes the distinction between the processor time and the elapsed time.

![Figure 3.4: Time flow](image)

### 3.1.2 ARM IntegratorCP from the C perspective

The platform file is described in a similar way as the previous example, with the difference that all the peripherals required to boot Linux must be instanced and their interrupt nets must be defined. The following example is an abridged version of the platform description:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "icm/icmCpuManager.h"   //File that includes other headers needed by OVP to start a
//simulation

//Variables to store the arguments that are used, if any, when launching the platform.
static    Bool  noGraphics        = False;
static    Bool  wallClock         = False;
...

//Parse the arguments that are used when launching the platform, if any.
static void parseArgs(int firstarg, int argc, char **argv)
{
    int i;
    for(i=firstarg; i < argc; i++) {
        if (strcmp(argv[i], "wallclock") == 0) {
            wallClock = True;
        } else if (strcmp(argv[i], "nographics") == 0) {
            noGraphics = True;
        }
    }
}

void createPlatform(char *kernelFile, char *ramDisk, Uns32 icmInitAttrs, Uns32 icmAttrs)
{
    //Set general attributes for the platform
    icmInit(icmInitAttrs, 0, 0);
    icmSetSimulationTimeSlice(0.001000);
    if(wallClock)
        icmSetWallClockFactor(3.0);
```

Hardware and Software Co-simulation

```cpp
icmSetPlatformName("ArmIntegratorCP");

//Create bus instances
icmBusP bus1_b = icmNewBus("bus1_b", 32);
icmBusP membus_b = icmNewBus("membus_b", 32);

//Processors

//Create a string pointing to the directory where the compiled model is stored
const char *arm1_path = icmGetVlnvString(0, // path (0 if from the product directory)
                                        "arm.ovpworld.org", // vendor
                                        0, // library
                                        "arm", // name
                                        0, // version
                                        "model"); // model

//Define attribute list for the processor instance
icmAttrListP arm1_attr = icmNewAttrList();
icmAddStringAttr(arm1_attr, "variant", variant);
icmAddStringAttr(arm1_attr, "compatibility", "ISA");
icmAddStringAttr(arm1_attr, "showHiddenRegs", "0");
icmAddDoubleAttr(arm1_attr, "mips", 200.000000);
icmAddStringAttr(arm1_attr, "endian", "little");

//Create processor instance, feeding the path string and the attribute list as part of the arguments.
icmProcessorP arm1_c = icmNewProcessor("arm1", // name
                                        "arm", // type
                                        0, // cpuId
                                        0x0, // flags
                                        32, // address bits
                                        arm1_path, // model
                                        "modelAttrs", // symbol
                                        0x20, // procAttrs
                                        arm1_attr, // attrlist
                                        0, // semihost file
                                        0 // semihost symbol
                                        );

//Define bus1_b to be used as data and instruction bus.
icmConnectProcessorBusses( arm1_c, bus1_b, bus1_b );

//PSE Core module

//Same methodology as the processor model.
const char *cm_path = icmGetVlnvString(0, 0, 0, "CoreModule9x6", 0, "pse");
icmAttrListP cm_attr = icmNewAttrList();
icmPseP cm_p = icmNewPSE("cm", cm_path, cm_attr, 0, 0);

//Define the size that the peripheral will take in the bus. This size must agree to the size defined in the peripheral's implementation.
icmConnectPSEBus( cm_p, bus1_b, "bport1", 0, 0x10000000, 0x10000fff );
```
Instance of all other peripherals such as interrupt controller, timer, LEDs, keyboard, mouse, real time clock, UARTs, etc., has to be made in the same way as this peripheral, indicating the corresponding path for the PSE model location, setting their attributes and connecting them to the main bus.

```c
icmMemoryP ram1_m = icmNewMemory("ram1_m", 0x7, 0x7ffffff);
icmConnectMemoryToBus( membus_b, "sp1", ram1_m, 0x0);

icmMemoryP ambaDummy_m = icmNewMemory("ambaDummy_m", 0x7, 0xffff);
icmConnectMemoryToBus( bus1_b, "sp1", ambaDummy_m, 0x1d000000);
```

//Create a bus bridge between two buses so that a window of the slave bus is visible from a certain address onwards on the master bus.
icmNewBusBridge(bus1_b, membus_b, "ram1Bridge", "sp", "mp", 0x0, 0x7ffffff, 0x0);
icmNewBusBridge(bus1_b, membus_b, "ram2Bridge", "sp1", "mp", 0x0, 0x7ffffff, 0x80000000);

//Define fast and normal interrupt request nets for the processor.
icmNetP irq_n = icmNewNet("irq_n");
icmConnectProcessorNet( arm1_c, irq_n, "irq", ICM_INPUT);
icmConnectPSENet( pic1_p, irq_n, "irq", ICM_OUTPUT);
icmNetP fiq_n = icmNewNet("fiq_n");
icmConnectProcessorNet( arm1_c, fiq_n, "fiq", ICM_INPUT);
icmConnectPSENet( pic1_p, fiq_n, "fiq", ICM_OUTPUT);

//Define interrupt net for UART peripheral
icmNetP ir1_n = icmNewNet("ir1_n");
icmConnectPSENet( pic1_p, ir1_n, "ir1", ICM_INPUT);
icmConnectPSENet( uart1_p, ir1_n, "irq", ICM_OUTPUT);

/*Define interrupts for all the peripheral in the same way as the previous example.*/
```

//Simulation’s entry point
```c
int main(int argc, char *argv[])
{
    if(argc<3) //Check for at least zImage and fs.img arguments
        return -1;

    char *kernelFile = argv[1];
    char *ramDisk = argv[2];
    parseArgs(3, argc, argv);

    Uns32 icmAttrs = ICM_ATTR_SIMEX; // simulate exceptions
    Uns32 icmInitAttrs =   ICM_STOP_ON_CTRLC // Ctr-C stops simulation
        | ICM_VERBOSE          // Print out statistics on simulation finish
        | (wallClock ? ICM_WALLCLOCK : 0);  // Do not move time forward
        // when blocked
```
// Construct the platform
createPlatform(kernelFile, ramDisk, icmInitAttrs, icmAttrs);

// Start simulation and keep running until it is finished or interrupted
icmSimulatePlatform();

// Clean-up
icmTerminate();

return 0;
4 Co-simulation approaches

After understanding the role of the platform and models in the simulation environment, it becomes clearer that it is not easy to hook up VHDL directly to the platform, as OVP does not have native VHDL support. A workaround needs to be found to be able to complete the whole hardware, software and VHDL co-simulation. The following six configurations (Figures 4.1 – 4.5) have been devised as possible solutions for completing the task, divided in two big categories. All the options require the usage of an OVPsim license except when indicated otherwise.

4.1 OVPsim as main simulator

a) Create each VHDL model as a PSE device, hook directly to the platform.

• Pros: Fastest simulation alternative (instruction-wise), native OVP APIs are used to interact with the peripheral models, simulation is not slowed down as opposed to hooking up the OVP platform with an outer HDL simulator. No problems between instruction-accurate and cycle-accurate models. Probably the fastest and most cost-effective solution.

• Cons: comprehension of the PSE API is required to rewrite each VHDL into a PSE model, can be a time consuming task and is not easily modifiable by persons who are not familiar with OVP environment. Each peripheral must be memory-mapped, losing fidelity to the original platform where the models are in the FPGA. When a change is made to a VHDL model, the same change must be implemented in the other model description (this will be referred as formal equivalence onwards), in this case, PSE.

b) Create an FPGA model in PSE and translate VHDL models into C files that can be loaded by the FPGA model.

• Pros: The same as the previous approach, the final platform simulation is a more accurate representation as an FPGA model is implemented, just as in the physical platform.

• Cons: The same as the previous approach, creating an FPGA peripheral is time consuming and not really necessary for a proof of concept platform.
4.2 Non-OVPsim as main simulator

c) Encapsulate OVP ARM model in a SystemC TLM wrapper, rewrite the VHDL models in SystemC TLM (or use a converter) and hook everything together without using an FPGA model.

- **Pros:** No need to master PSE API, usage of C/C++ language. SystemC is recognized a standard by the IEEE. Provides better performance than mixed-language approaches such as concept E.

- **Cons:** New libraries regarding SystemC functions need to be understood to rewrite the VHDL. Converters do not always give a correct output, manual intervention is required. Using a commercial converter (such as Carbon Model Studio) can imply additional cost. Formal equivalence must be kept between models.

d) Encapsulate OVP ARM model in a SystemC TLM wrapper, build an FPGA in SystemC and implement the VHDL models into SystemC files that can be read by the simulated FPGA.

- **Pros:** The same as the previous approach, the final platform simulation is a more accurate representation as an FPGA model is implemented, just as in the physical platform.

- **Cons:** The same as the previous approach, creating an FPGA peripheral is time consuming and not really necessary for a proof of concept platform.
e) **Encapsulate OVP ARM model in a SystemC TLM wrapper, use the mixed-language platform support from ModelSim (or another commercial HDL mixed language simulator) and directly load VHDL models.**

- **Pros:** No need to translate VHDL models as ModelSim can deal with direct HDL simulation and has SystemC support, thus hooking up the ARM model and the VHDL models should be an easier task. Actual implementation time might be shorter because FPGA design is not needed. No need to master PSE API, usage of C/C++ language.

- **Cons:** New libraries regarding SystemC functions need to be understood to hook the processor model in ModelSim (or another commercial tool), additional cost for licensing the mixed language simulator, simulation speed will be severely slowed down because of mixture of cycle-accurate and instruction-accurate environments.

![Figure 4.5: Concept E](image)

f) **Usage of a commercial EDA integrated solution.** (This concept does not necessarily imply the use of an OVP processor model).

- **Pros:** Design input can be in SystemC TLM 2.0 or legacy RTL as VHDL/Verilog/SystemVerilog, C, C++, assembly. Most software suites include a library of basic models and provide bug tracing features. Virtual prototyping is fast. These solutions are specialized on platform visibility when the software is already deployed.

- **Cons:** A maintenance contract might be needed, software suites are highly priced and training might be needed to use software efficiently.

To comprehend these benefits and drawbacks and decide which solution is the best for designing a proof of concept platform, it is necessary to spend some time learning the fundamental parts that make up each approach, which is the focus of the next chapters.
5  PSE oriented configurations

This chapter presents some discussion on approaches A and B, which make use of the Peripheral Simulation Engine (PSE).

From chapter 2, we can recall that PSE allows the implementation of peripheral models and interception libraries to enable interaction with the native host. These two co-simulation setups propose to rebuild into PSE models the companies’ VHDL such as capture and compare units, data loggers, interrupt controllers, UARTs, stepper controllers among other devices.

When a PSE peripheral executes, time flow inside the simulator does not advance and the platform waits for the peripheral response, so it is non-pre-emptive code. PSE models are written as C files, which are then compiled into .o files and linked into .pse files using the PSE toolchain. Peripherals are then instantiated from the main platform definition.

5.1 Interceptions

The PSE processor runs in its own memory space which is isolated and protected from the host environment, access to the latter is restricted. Even so, the user can create a peripheral model comprising both behaviourial code (running on a PSE) and functional code (running natively on the host) by using interception functions, which are defined directly inside the peripheral model. Interception functions are intended for semihosting. When a function that has been registered as semihosted is triggered within the PSE, it will be intercepted by the simulator and an alternative native function is executed. Using the VMI API, information from the simulation environment can be transferred into the native environment and viceversa.
5.2 Modelling APIs

There are two main OVP-native APIs that are used when creating PSE peripheral models and when instancing them from the platform:

- **Behaviour Modelling (BHM)**: Defines the peripheral’s behaviour with constructs such as delays and events. Refers to process control and time management.
- **Peripheral Platform Modelling (PPM)**: Defines interfaces between peripherals and the platform bus ports & net ports.

Peripherals written with BHM and PPM need to be compiled by using the PSE toolchain. The table below shows a comparison between BHM and PPM:

<table>
<thead>
<tr>
<th></th>
<th>BHM</th>
<th>PPM</th>
</tr>
</thead>
</table>
| header         | ```
#include "bhm.h"
```                                     | `#include "ppm.h"`                                 |
| give access to | • Threads                                         | • Connectivity of peripherals in platforms         |
|                | • Events and named events                       | • Create and control of ports, nets and address spaces |
|                | • Simulated delays                               | • Create behaviour on memory region accesses       |
|                | • Simulator control                              |                                                  |
|                | • Simulator message stream and diagnostics control |                                                   |
| general function| create and control threads of execution          | access to the platform hardware; interact with other components on the platform |
| operations     | • Create and delete tasks for the peripheral     | • Read and write memory regions on the platform    |
|                | • Allow a task to wait for a period of time      | • Define interrupt nets                           |
|                | • Allow a task to wait for an event in another task |                                                   |
|                | • Perform basic host Operating System I/O functions | • Arrange communication between tasks              |

Table 5.1: BHM and PPM features

Communication between the processor and peripherals is made through registers. A register can be modelled by specifying an area of memory, which is defined as a port within the peripheral. The peripheral grants a certain amount of size to the port, and once the peripheral is instantiated in the platform, the port is located in the memory space on the platform.

In order to read/write data in the peripheral’s internal registers, a callback function should be defined (for external access to the register), which can be applied to all or part of the memory region occupied by the register. A callback will be triggered when a bus access to the address bound to the callback occurs.

![Figure 5.3: Callbacks](image_url)
5.3 **Basic design elements**

To have a more concrete overview on how the abstraction discussed on the previous section works, the following code snippets are presented to get a feeling of the basic structure for a generic peripheral. In-depth description of functions and their arguments are available on OVP’s Doxygen-based documentation system\(^1\).

To install a memory-mapped user-viewable register, the function `ppmCreateRegister` is used:

```c
static void installRegisters(void *base_addr)
{
    ppmCreateRegister("type", UART_TYPE, handles.bport1, 0, 2, read_16, write_16, view16, &bport1_ab_data.type.value, True);
}
/*The parameters, are, in order:
Register name, short description, base address in the exposed window, offset address in the
exposed window, width of the register in bytes, callback to implement bus-read, callback for
bus-write, callback for debug (non-disturbing read), data passed to the callback functions,
whether to write to the register even if the value has not changed.
*/
```

For defining the callbacks used with the previous register, the functions `PPM_VIEW_CB`, `PPM_READ_CB` and `PPM_WRITE_CB` are used:

```c
static PPM_VIEW_CB(view16) { *(Uns16*)data = *(Uns16*)user; }
static PPM_READ_CB(read_16) { return *(Uns16*)user; }
static PPM_WRITE_CB(write_16) { *(Uns16*)user = data; }
```

The preferred method for defining the registers that make up the peripheral and store its state is via a `struct` with `unions`. Defining bit fields is a common practice on the available open source peripherals for ease of access when implementing its behaviour, but the designer must be aware of that bit fields can be dangerous due to compiler and endianness dependencies.

```c
typedef struct bport1_ab_dataS
{
    union
    {
        Uns16 value;
    } type;
    union
    {
        Uns16 value;
        struct
        {
            unsigned bit0 : 1;
            unsigned bit1 : 1;
            unsigned bit2 : 1;
            unsigned bit3 : 1;
            unsigned bit4 : 1;
        } bits;
    } status;
} bport1_ab_dataT, *bport1_ab_dataTP;
```
5.4 Instancing peripherals from the platform

As the objective of this chapter is to provide an overview of the features available on PSE to create a peripheral, further technical explanations about the functions available will be omitted as the necessary amount of information is available on OVPsim documentation system and user manuals. Instead, the following example is presented on how to instance a peripheral from the platform. A peripheral can be configured using attributes which may be either built into the platform description or passed by as arguments when invoking a simulation. When a peripheral model has functionalities created as both PSE and native codes, the attributes are all passed through the same attribute list to the instantiation of the PSE. This snippet shows how to instance a keyboard peripheral from the main platform description.

```c
// Create the peripheral location path
const char *keyboard = icmGetVlnvString(vlnvRoot, "directory_of_model", "peripheral",
"variant", "1.0", "pse");
const char *keyboardIntercept = icmGetVlnvString(vlnvRoot, "directory_of_model",
"peripheral", "variant", "1.0", "intercept");
{
    //Create variable for storing attributes
    icmAttrListP keybAttrs = icmNewAttrList();
    //Add peripheral attributes
    icmAddStringAttr(keybAttrs, "keyboardModel", "TestSimulation");
    //Create instance of peripheral
    icmPseP useKeyboard = icmNewPSE("useKeyboard", keyboard, keybAttrs,
    keyboardIntercept, "modelAttrs");
}
//Connect PSE peripheral to "bus", where start and end address
//specify the point in memory where the peripheral is mapped
icmConnectPSEBus(useKeyboard, bus, "useKeyboard", False, start_address, end_address);
```

5.5 General overview

Concepts A and B propose a native way to get peripherals to communicate with the processor model in a fast way. Although the learning curve of PSE is not very steep, a considerable amount of time is still required to analyse the documentation and comprehend how PSE-designed peripherals interact with the rest of the system. Available function documentation is good but extensively commented peripheral model examples are scarce, it might be necessary to work all the way through a briefly-commented peripheral source code to understand how to model its behaviour. Peripheral structure is good overall as it is coded in an intuitive way using traditional C programming. The discussion forums available on OVP’s website are a good starting point for reviewing common issues and clearing up questions as the staff is quite responsive, though there is not much expertise from part of the user community. A very important aspect that must be taken into account is that PSE is not an industry standard, also a formal equivalence between the PSE model and a HDL description must be kept and currently there is no way to automate this process, thus requiring a manual recoding. For this reason, OVP is mainly focused on general software visibility and debugging rather than hardware design. Attribute lists allow easy implementation of test cases. Overall, concepts A and B are good for providing a proof-of-concept platform, but maybe not the best for a sustainable long term implementation because of the model equivalence issue.

Notes

1 API documentation is available on `<installdir>/Imperas/doc/api/`. User manuals are available on `<installdir>/Imperas/doc/ovp/`.
6 SystemC TLM 2.0 oriented configurations

This chapter presents the background of concepts C and D, which suggest a complete high-level abstraction using SystemC TLM and its OSCI simulator as the base kernel.

In these configurations, VHDL has to be rewritten into SystemC TLM descriptions and the OVP models have to be wrapped in a TLM interface. By accomplishing these two steps, everything can be bound together in a SystemC TLM platform. For developing this solution, it is necessary to have a good understanding of the following elements:

1. How SystemC works in order to create the main platform and get to know the basis upon which TLM is built.
2. Basic structure of the TLM library, for implementing peripherals at a high level of abstraction.
3. Formal equivalence between HDL and SystemC implementations.

The aim of this chapter is to provide an insight on these topics.

6.1 A primer on SystemC

SystemC is a set of classes and macros that sit on top of C++ and provide an event-driven methodology for high level hardware modelling. As the list of functions that SystemC implements is quite long, it is often considered as a language on its own.

It was born when the design community realised the problems of using different languages, as hardware engineers worked on VHDL and software engineers on C/C++. SystemC intends to provide an adequate level of abstraction for integrating these parts. It became a standard on December of 2006, receiving approval as the IEEE 1666.

![Figure 6.1: Concepts C and D](image)

![Figure 6.2: Abstraction levels language comparison](image)
6.1.1 Structural design

SystemC allows creating and communicating a design via modules, processes, events, ports, channels and interfaces.

- **Module:** components are encapsulated inside “modules”, which are classes that inherit from the `sc_module` base class. SystemC provides some macros to lessen the burden of typing class dependencies at the moment of defining constructors. As in VHDL, SystemC separates the interface specification from the implementation in .h and .cpp files respectively (entity/architecture in VHDL). Modules can contain instances of other modules.

- **Process:** They correspond to member functions of modules that can be called by the simulation kernel. From a software perspective, processes translate to threads thus allowing concurrent execution; but from the hardware point of view their use is to model timed circuits. Processes can communicate using channels and events. Two types of processes can be defined within SystemC:
  - `SC_THREADs` are invoked only once by the simulation kernel and can suspend themselves allowing a notion of “time flow”, once they quit they cannot be called again.
  - `SC_METHODs` which can be invoked multiple times, and where time does not pass between the simulator call and the function return.

- **Event:** Their use is to wake up or send to sleep `SC_THREADs` and `SC_METHODs` to allow the illusion of concurrency. For threads, this can be accomplished with `notify(arguments)` and `wait(arguments)`, and for methods only the function `next_trigger(arguments)` is used.

- **Ports and channels/signals:** To transfer data between modules a channel (also known as signal) is used, which is an analogue to the HDL signal. A set of predefined channels such as FIFOs are included with SystemC. A port, which is a more sophisticated version of the HDL port, is used to receive/send information through the channel, and forms part of a module.

- **Interface:** Provides the means for modules to be independent from the mechanisms implemented in the communication channels. They are implemented as abstract base classes that define a set of access methods that can be used by all the derived classes, although this class itself will never create objects.

Channels are defined as classes that inherit from interfaces. The interface enables the module to use a channel via ports. In other words, interfaces do not provide any implementation but are mere virtual declarations of methods that will be referenced by channels and ports. The channel will implement all the methods inherited from the interface class.

SystemC introduces the concept of time with a 64 bit range within the `sc_time` class. It also provides its own data types to model information such as hardware signals, registers and four-state logic (0, 1, X, Z) which are designed for improving simulation speed. It does not currently support analogue hardware although a reasonable model can be obtained using floating-point representations, which are compatible with SystemC. Native C++ data types can be used and they affect the overall module performance in a positive way, but they contribute to fidelity loss towards the original specification.
There is a special type of port called *sc_export*, its idea is based on moving a channel inside of a module with the purpose of hiding connectivity details, and to use this *sc_export* from the outside as if it was behaving like a channel. For example, a designer may want to make publicly available only some specific channels for protection reasons. In order to visualize the previous abstractions easily, the picture below presents a summary of all the elements described.

![Element summary diagram](image)

**Figure 6.3: Element summary**

There are some guidelines which must be followed when designing in SystemC (which will be discussed later for restrictions related to running a SystemC platform in the ModelSim kernel). The most important are:

- Processes can communicate with each other in the same module using a channel or event.
- Processes that are not the same level of design hierarchy can communicate each other using the interfaces provided by ports.
- A module port may connect directly to a port of a submodule.
- A port can connect to a process via an interface.

Doing otherwise can generate concurrency problems, for this reason it is convenient to spend some time understanding the basics of this property in SystemC.

### 6.1.2 About concurrency

The use of concurrency allows simulating hardware signals in a realistic way by introducing the concept of *delta cycle*, which is a small step of time where the user-visible time is not increased in the simulation. This is used to determine the order in which processes have to execute when simulated time is not running, thus guaranteeing deterministic behaviour.

SystemC also uses a cooperative multitasking model as VHDL and Verilog. In order to be able to identify the way this affects the whole platform implementation in the ModelSim, it is necessary to have some insights on how the SystemC kernel (OSCI simulator) works, because ModelSim’s SystemC support is based on the latter. There are four elementary phases in the execution of a SystemC simulation:

- *Elaboration phase*: refers to instantiation of objects and assignment the corresponding connections in *sc_main()* (which replaces the typical C/C++ *main()*), via the *module.port1(port2)* syntax. Immediately after, the *sc_start()* function must be called, because the processes are already registered with the kernel at the moment of object creation.
**Initialization phase:** the kernel creates two queues, “runnable” and “waiting”. Depending on the setup, most processes are inserted directly into the runnable queue.

**Evaluation phase:** this phase resembles a state machine as the kernel runs each process until it returns, or executes a `wait()` so at this point the process is moved to the waiting queue together with its timing information. When all the processes are dispatched and the runnable queue is empty, the kernel “advances” time and continues evaluation.

**Cleanup:** when all processes yield or one of them executes `sc_stop()`, the simulation is terminated allowing to destroy objects and print simulation information.

### 6.1.3 Example

The following basic code snippet is presented to get a better overview of the information presented on this chapter. It implements an isolated module that counts from 1 to 10.

```cpp
//main.cpp
#include “counter.h”
int sc_main(int argc, char* argv[])
{ 
    counter my_counter(“my_counter”);
    sc_start();
    if(not sc_end_of_simulation_invoked())
        sc_stop();
    return 0;
}

//counter.h
#ifndef COUNTER(my_thread_process)
#define COUNTER
#include <systemc.h>
SC_MODULE(counter) //class macro
{ 
    void my_thread_process(void);
    SC_CTOR(counter) //constructor macro
    {
        SC_THREAD(my_thread_process);
    }
}; #endif

//counter.cpp
#include “counter.h”
void counter::my_thread_process(void)
{
    for(int i=1; i<=10; i++)
        std::cout<<i;
    std::cout<<” printed on “<< name()<<endl;
}
```

Output: 12345678910 printed on my_counter
6.2 Design levels in SystemC

One of the features of SystemC is its event-based model of computation, which provides a quite flexible design base. This allows designing modules at different levels of abstraction, where not all of these have to be at the same hierarchy. For example if a detailed model is already implemented, a more abstract one can be created for IP protection or simple simulation purposes, which is what we are looking for in this research. It can also happen the other way around: constructing an RTL model from a high level specification. The following levels can be defined with SystemC:

- **Functional Level Model** is focused on model interfaces, mainly used for algorithm proof purposes.

  - **Untimed functional** uses a delta cycle approach with no time concept (referring to execution and data transport), sequential processes communicate via FIFOs with blocking read and write functions and synchronization is implicit. They are based on SC_THREADS.

    ```
    SC_MODULE(ADD_EXAMPLE)
    {
        sc_fifo_in<int> in1, in2; //FIFO channels for data input
        sc_fifo_out<int> out; //and output
        void proc()
        {
            while(1)
                out.write(in1.read() + in2.read());
        }
        SC_CTOR(ADD_EXAMPLE)
        {
            SC_THREAD(proc);
        }
    }
    ```

  - **Timed functional** uses time concept (models latencies), data transport takes time using wait(sc_time). Can coexist with an untimed model.

    ```
    SC_MODULE(ADD_EXAMPLE)
    {
        sc_fifo_in<int> in1, in2;
        sc_fifo_out<int> out;
        void proc()
        {
            while(1) {
                int val = in1.read() + in2.read();
                wait(100, SC_NS); out.write(val);
            }
        }
    }
    ```

- **Transactional-Level Model** is loosely-timed.

- **Behavioural Hardware Model** is approximately timed.

- **Register Transfer Level Model** is cycle accurate.

<table>
<thead>
<tr>
<th>Design Level Model</th>
<th>Untimed functional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Level Model</td>
<td>Timed functional</td>
</tr>
<tr>
<td>Transaction-Level Model</td>
<td>Loosely-timed</td>
</tr>
<tr>
<td>Behavioural Hardware Model</td>
<td>Approximately-timed</td>
</tr>
<tr>
<td>Register Transfer Level Model</td>
<td>Cycle accurate</td>
</tr>
</tbody>
</table>

Table 6.1: Design levels in SystemC
• **Transaction-Level Model (TLM)** is about functionality, makes a clear distinction between communication and implementation. Uses blocking and non-blocking I/O. Provides the best accuracy-speed ratio for platform modelling. Further explanations will be reserved for the next section.

  o **Loosely-timed** is intended for early software development because partial hardware verification with functionality purposes is possible. Abstraction level is sufficient to boot an OS. Supports temporal decoupling (threads keep their own time and synchronization is only effectuated when necessary. Typically very complex bus protocols and a big number of processes are needed.

  o **Approximately-timed** is intended for architectural exploration with a notion of performance. Processes are lock-stepped with simulation time. Uses non-blocking calls.

• **Behavioural Hardware Model** is pin and functionally accurate, but does not represent cycle accurate behaviour. The implementation of the specification is still abstracted at some degree.

```vhdl
SC_MODULE(fibonacci)
{
    //Control
    sc_in_clk clk;
    sc_in<bool> reset;
    sc_out<bool> ready;
    //Data
    sc_in<unsigned> a = 0, b = 0;
    sc_out<unsigned> c = 1;
    void calculate();
    SC_CTOR(fibonacci)
    {
        SC_CTHREAD(calculate, clk.pos());
        watching(reset.delayed() == true);
    }
}
void calculate()
{
    unsigned next, prev, actual;
    next = a.read();
    prev = b.read();
    actual = c.read();
    while(true)
    {
        // IO Cycle 1,3,5,7...
        a.write(next);
        b.write(prev);
        c.write(actual);
        ready.write(true); //Assert that cycle finished
        wait();
    }
}
```
Hardware and Software Co-simulation

```cpp
next = a.read();
prev = b.read();
actual = c.read();
ready.write(false);
wait();

// At this point, IO will not take place.
// This illustrates the separation of implementation and communication
next = prev + actual;
prev = actual;
actual = next;
```

- **Register Transfer Level Model** is the clock-cycle implementation of the actual specification at pin level including all the states, registers, etc. Represents a complete description including communication channels (as a VHDL design).

```cpp
SC_MODULE(2bit_4to1_mux) {
  sc_in<sc_uint<2>> selector;
  sc_in<sc_int<2>> in3;
  sc_in<sc_int<2>> in2;
  sc_in<sc_int<2>> in1;
  sc_in<sc_int<2>> in0;
  sc_out<sc_int<2>> out;

  void mux();

  SC_CTOR(2bit_4to1_mux) {
    SC_METHOD(mux);
    sensitive << selector << in0 << in1 << in2 << in3;
  }
}

void mux() {
  switch(selector.read()) {
    case 3:
      out.write(in3);
      break;
    case 2:
      out.write(in2);
      break;
    case 1:
      out.write(in1);
      break;
    case 0:
      out.write(in0);
      break;
  }
}
```

OVPsim is stable when working with the TLM transport mechanism as it fits best with the instruction accurate performance the simulator provides. An important remark is that OVPsim does not provide native co-simulation of SystemC TLM 2.0 models (SystemC TLM models cannot be instantiated from an OVP platform); it provides models that may be used within a SystemC TLM 2.0 simulator through a wrapper. Because of this, the next section focuses on the basics of TLM.
6.3 TLM 2.0

Transaction Level Modelling 2.0 is supported by the Open SystemC Initiative (OSCI) and it is also part of the IEEE 1666 standard. TLM is a set of libraries that aid in modelling digital systems based on SystemC at a high level of abstraction, focusing mainly on interface and communication aspects rather than the algorithm performed by the process. TLM makes a clear distinction between implementation and communication between modules. The latter is modelled by using channels while hiding unnecessarily complex details. Its flexibility varies by layers: the high layers are more specific to bus modelling while the lower ones are more flexible.

In the user layer modules and sockets are defined. In the protocol layer, initiator and target sockets are implemented and callbacks are registered. In the transport layer, a payload is transported through channel. Before talking about the general structure of a TLM model, it is necessary to introduce these concepts in more detail.

6.3.1 Structure

- **Initiator and target**: the first has the task of generating new transactions while the latter will respond to these transactions. The transaction by itself is a data structured passed between initiator and target via function calls. When it is passed from initiator to target, the route is called *forward path* and the opposite is the *backward path*. A module that can perform both of the tasks of an initiator and target receives the name of *interconnect component*.

```
struct Initiator: sc_module
{...};

struct Target: sc_module
{...};
```

- **Socket**: holds the function of both a port and an export. An initiator socket has a port for the forward path and an export for the backward path. The target socket is the opposite. They inherit from the SystemC port binding functions to bind the port and export to the port and export respectively. With the usage of sockets the binding is simplified, and the underlying implementation where the transaction is transported will remain hidden.

In the user layer modules and sockets are defined. In the protocol layer, initiator and target sockets are implemented and callbacks are registered. In the transport layer, a payload is transported through channel. Before talking about the general structure of a TLM model, it is necessary to introduce these concepts in more detail.

![Figure 6.5: TLM 2.0 protocol](image)

![Figure 6.6: TLM elements](image)
- **Core interfaces:** there are four core interfaces. Blocking and non-blocking transport interfaces are basically used to transport transactions between initiators, targets and interconnect components. The Direct Memory Interface (DMI) is used by the initiator to access a block of memory that belongs to the target. The Debug Transport Interface provides non-intrusive means to read and write from/to a target. The combined interfaces are implemented as methods of a model. The prototype functions are:

```
//Blocking transport
b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& delay);

//Non-blocking transport (forward path)
nb_transport_fw(tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_core::sc_time& delay);

//Direct memory interface
socket.register_get_direct_mem_ptr(this, &Memory::get_direct_mem_ptr);

//Debug transport
transport_dbg(tlm::tlm_generic_payload& trans);
```

- **Generic payload:** Generic payload is the type of object used for transactions between socket classes; their use is to provide interoperability between modules. Each transaction has 10 bus attributes that have to be initialized by the initiator while the target can modify some of these attributes. In general practice, at least 8 of the 10 attributes are set.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Values and functions</th>
<th>Methods</th>
<th>Modifiable by target?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>TLM_READ_COMMAND: data copied from the target to the initiator; TLM_WRITE_COMMAND: data copied from the initiator to the target.</td>
<td>Initiator: set_command</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target: get_command</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>the lowest address to which data is read or written.</td>
<td>Initiator: set_address</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target: get_address</td>
<td></td>
</tr>
<tr>
<td>Data pointer</td>
<td>pointer to a data buffer within the initiator.</td>
<td>Initiator: set_data_ptr</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target: get_data_ptr</td>
<td></td>
</tr>
<tr>
<td>Data length</td>
<td>the length in bytes of data array in the initiator.</td>
<td>Initiator: set_data_length</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target: get_data_length</td>
<td></td>
</tr>
<tr>
<td>Streaming width</td>
<td>the width of a streaming burst; quals to data length for non-streaming transaction.</td>
<td>Initiator: set_streaming_width</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target: get_streaming_width</td>
<td></td>
</tr>
<tr>
<td>Byte enable pointer</td>
<td>default value 0 which indicates byte enables are not used.</td>
<td>Initiator: set_byte_enable_ptr</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target: get_byte_enable_ptr</td>
<td></td>
</tr>
<tr>
<td>Byte enable length</td>
<td>if the byte enable pointer is set to 0, the value of the byte enable length</td>
<td>Initiator: set_byte_enable_length</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>attribute should be ignored.</td>
<td>Target: get_byte_enable_length</td>
<td></td>
</tr>
<tr>
<td>DMI allowed</td>
<td>mandatory to be initialized to be false, and then reset by the target.</td>
<td>Initiator: set_dmi_allowed</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target: is_dmi_allowed</td>
<td></td>
</tr>
<tr>
<td>Response status</td>
<td>initialized to TLM_INCOMPLETE_RESPONSE indicating that the transaction has not been executed by the target, then reset by the target.</td>
<td>Initiator: set_response_status</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target: get_response_status</td>
<td></td>
</tr>
<tr>
<td>Extension pointers</td>
<td>permit any number of extensions to be defined to a transaction object; default value 0; normally not used.</td>
<td>/</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 6.2: Basic attributes of a generic payload

```
trans->set_command(command);
trans->set_address(0x17000000);
trans->set_data_ptr(reinterpret_cast<unsigned char*>(data));
trans->set_data_length(1);
trans->set_streaming_width(1);
```
trans->set_byte_enable_ptr(0);
trans->set_dmi_allowed(false);
trans->set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);

- **Base protocol**: compendium of rules for guaranteeing the transport of a transaction to a memory-mapped bus. It is pre-defined in the `tlm_base_protocol_types` class.

- **Top level connectivity**: One initiator socket is always bound to a target socket on the main platform.

```cpp
class Top: public sc_module
{
    public:
    Initiator *initiator_module;
    Target *target_module;

    SC_TOR(Top)
    {
        initiator_module = new Initiator("initiator_module");
        target_module = new Target("target_module");
        initiator_module->socket.bind( target_module->socket );
    }

    ~Top (); // Destructor
};
```

To summarize, a TLM2.0 modelling contains the following phases:
1. Create a new `sc_module`.
2. Define the socket(s) for the module.
3. Implement the sockets.
4. In the target, register callbacks for the incoming method calls related to the transport.
5. For initiators, set the generic payload attributes.
6. Instantiate processes inside the module.
7. Connect the modules on the top level.

### 6.3.2 Example
The example below includes the elements mentioned in this chapter. It represents two modules: the initiator module generates random data read and write commands, while the target module responds to these commands by sending data from its memory to the initiator, or writing data received from the initiator, respectively.

```cpp
#define SC_INCLUDE_DYNAMIC_PROCESSES

//SystemC Inclusions
#include "systemc"
using namespace sc_core;
using namespace sc_dt;
using namespace std;

//TLM inclusions
#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/simple_target_socket.h"
```
// Initiator module
struct Ini: sc_module
{
  // Initiator socket, defaults to 32-bits width.
  tlm_utils::simple_initiator_socket<Ini> ini_socket;

  // Constructor. The parameter passed to ini_socket is an internal string used by the module
  // and it should always correspond with the name of the instance.
  SC_CTOR(Ini) : ini_socket("ini_socket")
  {
    SC_THREAD(thread_process);
  }

  // Actual process of the module
  void thread_process()
  {
    // Create the generic payload transport object and define
    // a time delay
    tlm::tlm_generic_payload* trans = new tlm::tlm_generic_payload;
    sc_time delay = sc_time(10, SC_NS);

    // Data buffer
    int data;
    for (int i = 0; i < 96; i += 4)
    {
      // Randomly assign a TLM_READ_COMMAND or TLM_WRITE_COMMAND
      tlm::tlm_command cmd = static_cast<tlm::tlm_command>(rand() % 2);
      if (cmd == tlm::TLM_WRITE_COMMAND)
        data = 0xCC000000 | i;  // Modify the data to write

      // Initialize 8 out of the 10 attributes for the generic payload, byte_enable_length
      // and extensions are not used
      trans->set_command( cmd );
      trans->set_address( i );
      trans->set_data_ptr( reinterpret_cast<unsigned char*>( &data ) );
      trans->set_data_length( 4 );
      trans->set_streaming_width( 4 );  // when equal to data length, means no streaming
      // transaction
      trans->set_byte_enable_ptr( 0 );  // 0 for ignoring this attribute
      trans->set_dmi_allowed( false );  // always initialized to false
      trans->set_response_status( tlm::TLM_INCOMPLETE_RESPONSE );  // always initialized to
      // TLM_INCOMPLETE_RESPONSE

      // Register the socket to the blocking transport interface
      ini_socket->b_transport( *trans, delay );

      // Check if the transaction was OK
      if (trans->is_response_error())
        SC_REPORT_ERROR("TLM-2", "Error in b_transport!");

      // Print the results
      cout << "trans = [ " << (cmd ? "WRITE" : "READ") << ", " << hex << i << " ]
           data = " << hex << data << " at time " << sc_time_stamp() << " with a delay of "
           << delay << endl;

      // Yield control to other processes
      wait(delay);
    }
  }
};
// Create a new target module Tar
struct Tar : sc_module
{
// Target socket, defaults to 32-bits wide
tlm_utils::simple_target_socket<Tar> tar_socket;

// Internal memory for the module
enum {SIZE = 256};
int mem[SIZE];

// Constructor
SC_CTOR(Tar) : tar_socket("tar_socket")
{
  // Register the target with the b_transport interface
tar_socket.register_b_transport(this, &Tar::b_transport);

  // Initialize the target’s memory with random data
  for (int i = 0; i < SIZE; i++)
    mem[i] = 0xFF000000 | (rand() % 256);
}

// Blocking transport
virtual void b_transport(tlm::tlm_generic_payload& trans, sc_time& delay )
{
  tlm::tlm_command cmd = trans.get_command();
  sc_dt::uint64 adr = trans.get_address() / 4;
  unsigned char* ptr = trans.get_data_ptr();
  unsigned int len = trans.get_data_length();
  unsigned char* byt = trans.get_byte_enable_ptr();
  unsigned int wid = trans.get_streaming_width();

  // Implement received command
  if ( cmd == tlm::TLM_READ_COMMAND )
    memcpy(ptr, &mem[adr], len); // copy the data from the target to initiator
  else if ( cmd == tlm::TLM_WRITE_COMMAND )
    memcpy(&mem[adr], ptr, len); // copy the data from the initiator to target

  // If everything completed successfully set response status to OK
  trans.set_response_status( tlm::TLM_OK_RESPONSE );
}
};

// Top-level connectivity
SC_MODULE(Top)
{
  Ini *initiator;
  Tar *target;

  // Instantiate components
  initiator = new Ini("initiator");
  target = new Tar("target");

  // Bind the modules’ sockets.
  initiator->ini_socket.bind( target->tar_socket );
};

// Simulation's entry point
int sc_main(int argc, char* argv[])
{
  Top top("top");
  sc_start(); return 0;
}
6.4 Conversion from HDL to TLM

A design process that starts from zero benefits from the usage of SystemC to build a model prototype and then convert it into an RTL model, but the opposite is also possible which is what the two approaches presented by this chapter propose. This implies that it is necessary to recode the VHDL into SystemC TLM models. The main platform would be also written in TLM and the processor model from OVP connected to the latter via a TLM wrapper. There are different possibilities to accomplish the migration between languages such as manual recoding and automated translation.

6.4.1 Manual recoding

Represents the most time consuming option but probably the safest regarding program structure. Requires a good knowledge of SystemC constructs, processes and events. SystemC libraries are the only requirement for this; they can be freely downloaded from the official website of the initiative and imported in a project created in any C++ IDE.

6.4.2 HIFSuite

HIFSuite is a set of HDL tools that enable IP reuse by providing conversions between different languages. Two main tools are of interest:

- Translation tools: they rely on a proprietary intermediate language “HIF” (Heterogeneous Intermediate Format). The translation process is integrated by two types of translators. The conversion flow is as follows:

![HIF-Based Translation Flow](image)

Where the front-end represents the translator from VHDL/Verilog/SystemC source code to HIF files and the back-end is the translator from HIF files to the target language. According to claims by HIFSuite’s official site, version 2011.05 of the tools has the following level of construct support:

![HIFSuite’s construct support](image)
• Abstraction tools: their use is to convert RTL models into TLM descriptions, with the intention of preserving the simulation speed that is typical to the latter type of models. In a working-case scenario, these tools would provide an excellent way of converting VHDL descriptions into SystemC TLM models, as translation from a VHDL description to a pure SystemC RTL one would not take full advantage of SystemC’s high abstraction level speed benefits.

A demo version was downloaded from HIFSuite’s website for corroborating how efficient and successful the translation is. Installation of the tools is straightforward, but the results of conversions are not very encouraging. Translation of four different VHDL behaviours (a simple latch, register, counter, and a RAM module) was attempted, obtaining a complete whole translation only for the simple latch.

```
Figure 6.9: HIFSuite VHDL to SystemC translation attempt
```

All the tests except the simple latch crashed on conversion because of syntax errors on HIF files or assertion problems on HIF to SystemC conversion. The Abstractor tools also have low rate of success on conversion, showing also errors related to HIF syntax.

The complete overview that could be obtained by testing the tools is that they are still on a primitive stage and not feasible for usage in a large project, but they are a promising concept (especially because of the Abstractor tools) when development is advanced enough to provide stable translations.
6.4.3 **Carbon Model Studio**

Carbon Design Systems provide a solution\(^3\) related to cycle-accurate models and their validation. Included in the suite is a tool to make conversions from RTL models to C/C++/SystemC models. No further research has been possible for this toolkit as no demo version is available.

6.5 **Formal equivalence between HDL and TLM**

Proving the formal equivalence between VHDL and TLM models is still an open problem in the electronic design industry because their internal structure is organized differently. In order to make a comparison consider the two main branches of equivalence checking methodologies for RTL: **combinatorial checking** relies heavily on interface and temporal similarities to prove gate-level correctness, and **sequential checking** instead focuses on verification of two designs whose state encodings are different. For TLM, a small amount of literature is available at the time of this research but there are some recently documented techniques which focus on **event checking**: Nicola et al. (2007) suggest a methodology similar to sequential checking, in which the equivalence between RTL and TLM is intended to be proved by splitting the task into functional analysis (correctness of event sequence) and model performance (distribution of events over time). Tools such as HIFSuite are based on this technique. Model equivalence can be verified to a certain degree just by comparing model simulations: the output should be the same regarding the implementation. Of course that this approach may not be able to identify hidden bugs or glitches originated while coding the specification if extensive testing is not effectuated.

6.6 **Speed concerns related to translation algorithms**

Direct VHDL to SystemC RTL has a big downside: the simulation speed obtained by making a direct translation will be generally slow because SystemC can indeed model RTL behaviors (pin level), but going for this technique causes overhead due to the big number of events needed to simulate the signals, slowing down the whole performance. For this reason, its main usage in the industry is focused on higher level abstractions as TLM, for example. VHDL to SystemC TLM is the desired type of translation, but no reliable tools are yet available.

Additionally, it is important to consider that translation tools provide a non-optimal and mechanical way of conversion, generating relatively inefficient code. But from a software perspective, a TLM model design is just enough for application verification and testing.

6.7 **General overview**

These approaches might currently not be the best in relation to implementation time because open source or low cost description language translation tools are still not available. But on the other side, SystemC provides a flexible work environment which is being used more and more\(^4\) in the industry.

In comparison with pure OVPsim solutions (concepts A and B) which require mastering a program-specific C-based API for designing peripherals and the platform, SystemC is an industry standard heavily based on C/C++ providing more community support, although a good understanding of the scheduling system and TLM transport mechanisms is needed (the learning curve is not soft, as compared to OVP’s PSE).

The flexibility of the SystemC model of computation allows to easily modify peripherals to create test cases and for error injection purposes.
A good amount of research is on-going in the design community regarding TLM techniques, which at the present are not very widely documented. In a few words, a platform implemented in SystemC TLM is a sustainable solution for the long term when translation tools become available, but at the moment the best option (cost and reliability wise) is manual translation which is enough for creating a proof-of concept platform and documenting the implications of the whole process.

Notes
1 http://www.accellera.org
2 http://www.hifisuite.com
3 http://www.carbondesignsystems.com/carbon-model-studio/
7 A mixed language configuration in ModelSim

Using ModelSim as main kernel for structuring the virtualised platform, which is the solution proposed by concept E demands attention on the following elements:

1. TLM Wrapper for using OVP Processor and peripheral models inside ModelSim.
2. Concerns about abstraction mixing: instruction-accurate and cycle-accurate models.
3. Restrictions in mixed-language platform implementation.

As this solution also involves the usage of SystemC TLM 2.0, the main focus of this chapter will be on binding everything together through ModelSim, as SystemC and TLM have been already discussed on the previous chapter.

7.1 ModelSim as main kernel

The main reason for using ModelSim as an integration environment is because of its support for various standards such as Verilog, VHDL and SystemC. It has six basic tools that are needed to build the main test bench which is able to link a VHDL and SystemC model. They are divided on three main sets:

- **Work environment setup tools**:
  - *vlib*: sets the path where the result of .vhd/.c/.cpp file compilation is stored. This is with the purpose of preventing unnecessary compilation each time the simulation is run, so pre-compiled data will be stored in this directory.
  - *vmap*: used to create a logical library that references to the pathname of where the actual library is located. This can be practical when third party libraries are going to be used.

- **Compiler tools**
  - *vcom*: ModelSim’s compiler for VHDL.
  - *sccom*: ModelSim’s compiler for SystemC files.

- **Other tools**
  - *scgenmod*: automates the process of foreign module declaration in SystemC. (ie. counter.vhd > counter.h)
  - *vsim*: start simulation

There is an immediate drawback to this approach: to be able to run *vsim* when a SystemC file is involved, ModelSim requires a special SystemC license which implies an additional cost to the original software suite. Additionally, there is no support for SystemC on Windows 7 x64 at the time of finishing this research and a special version of the gcc compiler including MingW utilities (which can be obtained from Mentor’s SupportNet) is also needed for setting up a working environment.
On the other hand, ModelSim provides a very easy way to integrate all the model descriptions as it is a mixed-language kernel. To illustrate this, a shortened example is presented with the basic conceptual descriptions and structure needed to instance a VHDL entity from a SystemC test bench.

Consider a simulation started by typing the following commands on the ModelSim console:

```
vsim1> vlib work
vsim2> vcom hdlcomponent.vhd
vsim3> scgenmod hdlcomponent > hdlcomponent.h
vsim4> sccom launch.cpp
vsim5> sccom -link
vsim6> vsim -c testmodule
```

- **vsim1** sets up the working directory.
- **vsim2** compiles `hdlcomponent.vhd` which contains the peripheral description on VHDL.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE std.textio.all;
ENTITY hdlcomponent IS
    generic (------
    PORT (------
    END hdlcomponent;
ARCHITECTURE RTL OF hdlcomponent IS
BEGIN
    action: PROCESS
    BEGIN
        --do something
    END PROCESS;
END RTL;
```

- **vsim3** launches the `scgenmod` utility which automatically constructs a foreign module declaration (to provide an interface for the VHDL to interact with the SystemC platform) by generating the “`hdlcomponent.h`” file.

```c
// hdlcomponent.h
#ifndef INCLUDED_HDLCOMPONENT
#define INCLUDED_HDLCOMPONENT

class hdlcomponent: public sc_foreign_module
{
public:
    sc_in<bool> clk;
    ...

    //Constructor
    counter(sc_module_name name, char* hdl_name, int paramNumber, const char* params)
    :sc_foreign_module(name, hdl_name, paramNumber, params),
    {};
};
#endif
```
vsim4 compiles the “launch.cpp” file with the sccom utility, providing the entry point for the platform. It creates a SystemC constructor for the test module where the foreign VHDL module is instantiated.

```cpp
//launch.cpp
#include "testmodule.h"
SC_MODULE_EXPORT(testmodule)

//testmodule.h
#ifndef INCLUDED_TESTMODULE
#define INCLUDED_TESTMODULE
#include "hdlcomponent.h"

SC_MODULE(testmodule)
{
    sc_signal<T> iclock;
    ...
    ...

    //Allocate memory for VHDL component
    hdlcomponent* myVHDLmodule;

    //SystemC constructor
    SC_CTOR(testmodule) : iclock("iclock"), ...
    {
        const char* params[4];
        params[0] = strdup("TEST STRING OUTPUT START");
        ...
        params[3] = strdup("TEST STRING OUTPUT FINISH");

        //Instance VHDL component
        myVHDLmodule = new hdlcomponent("myVHDLmodule", "hdlcomponent", 4, params);
    }
    #endif
```

vsim5 and vsim6 are in charge of the linking and simulation starting, respectively.

7.2 Restrictions in mixed-language implementations
Boundaries in a mixed-language design need to be clearly defined in order to avoid configurations that are error prone. It would be natural to assume that interfaces that mix unsupported datatypes, events, classes and other structures between languages should be avoided in order to prevent writing tricky wrappers. The table on the next page has been compiled in order to have a reference of compatibility. Because SystemC extends all of C++’s functionality, there is support for function overloading. When there is need to call a function between boundaries, equivalent datatypes can be used as arguments. For example:

```vhdl
package vhdl_ex is
procedure word_32_and(driver1,driver2:in word_32; en: in bit:=0; and_out: out bit);
end vhdl_ex;

//SystemC
SC_IMPORT(vhdl_ex);
void module_ex(); word_32_and(data1,data2,0,&r_and); end;
```
Hardware and Software Co-simulation

For convenience, SystemC should be in the top level of the test bench hierarchy as it provides the most versatile structure for component instantiation and platform design. Inside it, foreign module declarations indicating the existence of VHDL components in the design need to be done.

An important fact on this whole ModelSim-based implementation approach is that the instruction-accurate environment (OVP model) will be forced to work at the speed of the HDL simulator, which is much slower. This creates an overhead on SystemC’s scheduler described on the previous chapter. But, is this slow-down considerable enough to discard ModelSim integration as a feasible solution? A study by Bernard et al. (2003) showed that a pure SystemC test simulation using ModelSim’s kernel ran at 18.2 kcycles/s while a SystemC-VHDL simulation ran at 2.5 KCycles/s, which represents a 9x slowdown. Comparison with the cycles/s of a pure VHDL implementation of the same system was not effectuated.

7.3 Usage of mixed-language environments in the industry
Designers generally need to implement these types of platforms when they have to use an IP core or another piece of code that is not available in the language where the main project is. In this specific case, SystemC is intended to be used as an interface between OVP’s processor model and the VHDL blocks. From this, it can be inferred that the quantity of reused components (in order to evade changing all the existing structures) can have an impact in the total implementation time if they are in different languages. Currently, there is no documented standard or procedure on how to target mixed-language designs as each particular case implies different requirements that might need writing situation-specific schedulers, wrappers and buffers, among others.

7.4 General overview
ModelSim uses a mixed-language environment that seems very useful for the co-simulation attempts as it provides an easy way to interface between languages. Unfortunately, mixing different levels of abstraction will slow down the simulation to the level of the lowest-abstraction model in the chain.

<table>
<thead>
<tr>
<th>SystemC</th>
<th>VHDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>integer or character</td>
</tr>
<tr>
<td>int</td>
<td>integer</td>
</tr>
<tr>
<td>short</td>
<td>integer</td>
</tr>
<tr>
<td>long long</td>
<td>integer</td>
</tr>
<tr>
<td>enum</td>
<td>enum</td>
</tr>
<tr>
<td>double</td>
<td>real</td>
</tr>
<tr>
<td>float</td>
<td>real</td>
</tr>
<tr>
<td>struct</td>
<td>record</td>
</tr>
<tr>
<td>class</td>
<td>------</td>
</tr>
<tr>
<td>union</td>
<td>------</td>
</tr>
<tr>
<td>sc_bit</td>
<td>bit</td>
</tr>
<tr>
<td>sc_logic</td>
<td>std_logic</td>
</tr>
<tr>
<td>bool</td>
<td>boolean</td>
</tr>
<tr>
<td>sc_bv&lt;T&gt;</td>
<td>bit_vector</td>
</tr>
<tr>
<td>sc_lv&lt;T&gt;</td>
<td>std_logic_vector</td>
</tr>
<tr>
<td>sc_int, sc_uint, sc_signed, sc_unsigned, sc_bigint, sc_biguint</td>
<td>integer (w = 32)</td>
</tr>
</tbody>
</table>

Table 7.1: Datatype comparison
After consulting some experienced in-company HDL designers, we confirmed that involving VHDL components with SystemC TLM descriptions means a very significant slowdown in general simulation speed and this reduces the feasibility of this approach as the general intention is to test software on the embedded system simulation and be able to trace errors, yielding a slow approach unacceptable. Additional to this, an extra cost must be considered for a ModelSim SystemC license.
8 Commercial EDA tool configurations

There are some commercial solutions that are oriented to virtual platform modelling. Acquiring these suites implies a high cost despite the fact that they are heavily based on TLM. They enable a faster platform design than the currently available open source alternatives. Some of the most popular non-open source hardware and software co-simulation frameworks are:

- **Virtual System Platform (VSP):** Developed by Cadence, it is intended for virtual prototyping and software debug within the platforms created. Thus, designers can create a simulated platform and begin with software development at an early phase in the product’s cycle. It is optimized for fast software debugging. It can read IP-XACT specifications and automatically produce a TLM2.0 template. Comes pre-loaded with a library of TLM models that include a UART, keyboard, mouse controller, interrupt controllers, programmable LEDs, an LCD, Ethernet controllers, serial interfaces, touch screen inputs, among others. Provides support for ARM fast processor models and it is possible to instance OVP’s processor models too. It is compatible with third party software debug tools. A mixed-language platform optimized for HW/SW debugging and replay can be designed as there is support for SystemC TLM, C, C++, VHDL, Verilog and Assembly constructs. VSP is specially intended for providing platform visibility, generating test cases and tracing bugs.

- **Simics:** Developed by Wind River, also intended for virtual platform creation. Optimized for scalable simulations. What-if scenarios can be developed for architectural analysis. Provides a snapshot functionality where the state of the whole system can be saved, and this snapshot can be transferred to other designers for analysis or for continuing the simulation in another workstation. Features breakpoints and single stepping in forward and reverse directions is possible. Can inject faults to test how the system will react. C, C++, SystemC, Python and DML models are supported.
Hardware and Software Co-simulation

- **Vista**: Developed by Mentor Graphics, is a prototyping and verification solution based on TLM. Has a model builder for helping in the implementation of TLM modules. It can graphically show the relationship between components in the platform. Although this toolkit can run an OS with software deployed as-is, it emphasizes more in power analysis.

- **Virtualizer**: Developed by Synopsis, has SystemC/TLM authoring tools for assembling a virtual prototype: a peripheral modelling tool generates templates and documentation automatically. Processor, interconnects, peripherals and reference designs are included in the suite. The official website states that more than 500 models are available. Supports SystemC/TLM and hybrid modelling with HDL. Provides an aggregated view of software and hardware behaviour. A demo of the Virtualizer toolkit was requested and a 2-hour full version trial was granted through an online virtual machine. The interface is clean and intuitive; a pre-assembled virtual platform using an ARM Cortex-A15 MPCore was quickly launched. It is possible to view the individual state of the components (memory contents) while the simulation is running. The suite appears to be more user friendly compared to the other commercial toolkits.

Figure 8.2: Synopsis Virtualizer
8.1 General overview
Commercial solutions offer integrated environments that speed up the process of building a virtual prototype to focus the efforts on analysing the behaviour of software and hardware. All of the mentioned suites include peripheral models that can be used to build the virtual prototype or as a reference for generating new models. They also have a GUI that presents information in a more intuitive way than the approaches mentioned on the previous two chapters. Fault injection and software analysis is mentioned as a key feature in almost all of the packages. The downside is that prolonged trials are not available and the cost that implies acquiring a toolkit from a commercial EDA company is high.
Concept evaluation

After researching the positive and negative aspects of each one of the proposed solutions, it is easier to make a general evaluation and decide which concept to implement in order to produce a proof-of-concept simulation. The table below represents an Engineering Design Specification (EDS) describing scores for all the six concepts and the important aspects to consider when choosing a solution. Weights are given according to the requirements of the assignment.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Weight (1-5)</th>
<th>A: Pure PSE</th>
<th>B: PSE + FPGA model</th>
<th>C: Pure SystemC</th>
<th>D: SystemC + FPGA model</th>
<th>E: ModelSim</th>
<th>F: Cadence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform concept</td>
<td></td>
<td>Points</td>
<td>Score</td>
<td>Points</td>
<td>Score</td>
<td>Points</td>
<td>Score</td>
</tr>
<tr>
<td>Synchronization between the actual VHDL model and the implemented solution.</td>
<td>3</td>
<td>5</td>
<td>15</td>
<td>5</td>
<td>15</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>Platform approximation to the physical board</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Available documentation and support</td>
<td>3</td>
<td>7</td>
<td>21</td>
<td>6</td>
<td>18</td>
<td>7</td>
<td>21</td>
</tr>
<tr>
<td>Ease of design</td>
<td>4</td>
<td>8</td>
<td>32</td>
<td>7</td>
<td>28</td>
<td>7</td>
<td>28</td>
</tr>
<tr>
<td>Need of understanding new APIs for building platform</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Standards compliance</td>
<td>4</td>
<td>5</td>
<td>20</td>
<td>5</td>
<td>20</td>
<td>7</td>
<td>28</td>
</tr>
<tr>
<td>Architectural strategies</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ease of deployment</td>
<td>3</td>
<td>6</td>
<td>18</td>
<td>6</td>
<td>18</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>Error detection and recovery</td>
<td>4</td>
<td>7</td>
<td>28</td>
<td>7</td>
<td>28</td>
<td>7</td>
<td>28</td>
</tr>
<tr>
<td>Memory management policies</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Compatibility of language mixing</td>
<td>2</td>
<td>6</td>
<td>12</td>
<td>6</td>
<td>12</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>Performance and resources</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulation speed and scalability</td>
<td>5</td>
<td>8</td>
<td>40</td>
<td>8</td>
<td>40</td>
<td>7</td>
<td>35</td>
</tr>
<tr>
<td>User friendly final solution</td>
<td>2</td>
<td>7</td>
<td>14</td>
<td>7</td>
<td>14</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>Licensing cost</td>
<td>5</td>
<td>9</td>
<td>45</td>
<td>9</td>
<td>45</td>
<td>9</td>
<td>45</td>
</tr>
<tr>
<td>Total from 390 possible points</td>
<td></td>
<td>270</td>
<td>264</td>
<td>281</td>
<td>275</td>
<td>253</td>
<td>293</td>
</tr>
<tr>
<td>Percentage</td>
<td></td>
<td>69%</td>
<td>68%</td>
<td>72%</td>
<td>71%</td>
<td>65%</td>
<td>75%</td>
</tr>
</tbody>
</table>

Table 9.1: Co-simulation concepts EDS

Concept 6 provides the best solution for a long term implementation of hardware and software co-simulation as there are sets of commercial tools that can be used to simulate a platform without spending too much time on researching about communication issues. Additionally, their user community is bigger and a library of TLM models and processors are already supplied with some software suites, although all this benefits come at a high price. Concepts C and D are the immediate next option because SystemC is an open source industry standard even though right now the model synchronization is still an open issue; new translation techniques between TLM and RTL are currently being researched by the community and these options could represent a sustainable and cost-effective solution in the future. Concepts A and B have the advantage of being easy to develop but their main issue is that it is necessary to master a program-specific API to develop the models and thus no formal equivalence verification is possible (but manual simulation comparison is still an option). Concept E is not considered because of its low speed performance despite being an easy option to bind the components together.
Due to resource constraints and the limited boundaries of the assignment, a proof of concept simulation is to be attempted with the **PSE Concept 1 and SystemC concept 3**.

A brief description is provided on each of the conditions of the table:

### Platform concept

- **Synchronization between the actual VHDL model and the implemented solution**: refers to the need of translating, if needed, the VHDL to another language implementation and how easy is to keep them updated.
- **Platform approximation to the physical board**: similarity of the platform implementation respect to the hardware board, including connections and peripheral structure.
- **Available documentation and support**: references and previous examples on platform integration.
- **Ease of design**: represents the level of difficulty for integrating the platform, it is the result of evaluating aspects as the time cost for design, security provided by the solution to hide implementation details, mechanisms provided for organizing simulation tasks in order to archive a better performance, communication mechanisms between different implementation levels, availability of information on how to structure a complete platform and level of attention required on the whole setup when changing component.
- **Need of understanding new APIs for building platform**: amount of time needed to invest learning proprietary language or other concepts.
- **Standards compliance**: how common is the solution respect to already existing tendencies for co-simulation in the industry.

### Architectural strategies

- **Ease of deployment**: level of difficulty for installing the whole simulation environment on different workstations.
- **Error detection and recovery**: error tracing features provided by the concept.
- **Memory management policies**: efficiency of the solution in terms of memory usage.
- **Compatibility of language mixing**: features of the solution for calling functions in other languages from the test bench.

### Performance and resources

- **Simulation speed and scalability**: refers to the efficiency of the simulated time respect to wall-clock time, and the effect of increasing on the number of components in a platform respect to simulator time.
- **User friendly final solution**: easiness for the end user to run a simulation.
- **Licensing cost**: cost implied by buying single seat licenses.
10 UART proof of concept peripheral designed with PSE

In order to test the concept 1 which refers to a peripheral designed with PSE and a platform integrated with OVP’s ICM API, a UART peripheral was implemented based on the specifications of the internal document MDD-UART_Controller which describes the behaviour of a typical UART. This chapter explains the structure of this peripheral and the implementation process.

10.1 File structure and naming conventions

OVP uses a Vendor, Library, Name, Version (VLNV) format library structure for storing all the models. Therefore, it is necessary to respect some convention at the moment of deploying a peripheral. The default directory for storing source code is:

<installDir>/ImperasLib/source/<vendor>.ovpworld.org/<library>/<name>/<version>/

While the compiled binaries should be stored at:

<installDir>/lib/Linux32/ImperasLib/<vendor>.ovpworld.org/<library>/<name>/<version>/

These directories are subdivided into: model, PSE, tlm2.0. The first one is used to store the files which describe the semihosting model for the peripheral, if necessary. The last two contain the actual implementation. For the UART, these values were set as:

<vendor> = arm
<library> = peripheral
<name> = OceUart
<version> = 1.0

For defining the behaviour, all the peripherals on OVP follow the structure below which is indicated by the included generic Makefile for compiling the source under the /PSE/ directory.

- user.c: Implements the peripheral’s behaviour and non-generic callbacks.
- pse.igen.h: contains register structure and port declarations, non-generic callback function prototypes.
- pse.igen.c: peripheral’s entry point. Implements generic callbacks. Associates the registers to memory addresses and their corresponding callbacks. The peripheral’s constructor is also defined in this file.
- pse.macro.h: optional, defines macros for ease of register access on user.c.

The psesockets.h file provides serial functions semihosting, and it is included in this particular case to aid in the UART’s I/O from/to the host. It is located under:

<...>/ovpworld.org/modelSupport/psesockets/1.0/model
10.2 Implementation

The source code for the UART can be found on Appendix II including the necessary comments for each function. The receiving part of the peripheral was implemented, which reads random data from a text file or a terminal connection in the host computer. With this, it is possible to check how the peripheral interacts with the Linux driver. In a general way, the process from instantiation to execution is as follows:

![Diagram of UART execution process]

Figure 10.2: PSE UART execution process

10.2.1 Callbacks

When the thread starts waiting for a period of time \( t \) defined in the UART’s implementation, it will yield the control to the rest of the simulation. During this time, if a bus access occurs in the addresses mapped to any of the registers, a callback will be triggered to write/read data in the corresponding register. This process is represented on the figure above by connector A, and can execute as many accesses as the time quantum allows. An example of the actions that can occur in this lapse is:

![Diagram of UART callback triggering]

Figure 10.3: PSE UART Callback triggering
From the UART RX thread time perspective, the execution can be modelled as:

![Figure 10.4: Time flow in the PSE UART](image)

### 10.2.2 Creating a test case: overflow

One of the benefits of using PSE is the ease of behaviour modification: through the use of attributes, it is possible to enable or disable certain parts of the peripheral’s implementation for test case purposes directly from the platform definition. With this, it is possible to simulate certain error conditions on demand, such as overflows.

For demonstrating this, an overflow injection function was written as part of the peripheral’s internal implementation in order to see how the full system would react. It can be triggered at connector B on Figure 10.2. The data parsing function from the UART was defined in such a way that the input file where the data is read from should have the following format:

```markdown
/*input.in
Contains random data that will be feed into the UART’s RXFIFO. The first byte corresponds to the parity indicator character and the second part is the data byte. When the first byte is set to the '*' character, an overflow condition will be triggered.*/

1H 0I 0J 1K 0L *M 1N 1H
```

In this test case, the UART will read all the data normally until it parses the ‘*’ character. At that point, it will fill the RXFIFO with random data and trigger the overflow bit on the status register. After that, it will continue reading the remaining data until the end of file.

When using PSE, taking advantage of attributes and semihosting enables to create test cases tailored to the designer’s needs. Because the software driver is deployed as-is without modifications it is possible to see how it will react in the real system. All of this is executed in a faster way than traditional testing methods, given the abstraction level at which the system is modelled (see Figure 1.1).
Virtual platform implemented with OVP’s ICM

After creating the UART peripheral, an ARM Integrator Compact Platform was assembled to test the functionality of the device. The source code for it can be found on Appendix III. When constructing a platform, the general flow is:

11.1 Attributes
To make clear how the peripheral’s starting conditions and other behaviours can be modified directly from the platform, it is convenient to examine the UART instantiation. When creating the attribute list it is possible to define the infile from where the UART will start reading data (from the host computer, through semihosting); otherwise, this attribute can also be changed by portnum to instruct the UART to open a channel connection for a telnet session. The responses to the attributes set in the platform are defined in the UART’s implementation.

The peripheral was configured to accept initial values for the registers as attributes passed from the platform. Other parameters can also be set from here. Defining behaviours in this way allows for easy test case generation as the peripheral’s source code does not have to be changed for each different test situation.

Finally, the peripheral’s string path is constructed and the PSE instance is constructed. As an additional visibility detail, it is a recommended practice to specify different levels of output verbose for message printing in the peripheral’s implementation. With this, it is possible to directly see the peripheral’s status at the specified level of detail from the simulator’s output.

```c
// UART
* To modify register starting conditions, use icmAddU16Attr(attributeList, "NAME_OF_REGISTER", value in hex);

* Other possible settings are:
  * -icmAddDoubleAttr(attributeList,"FIFO_INT_LEVEL",number); where number is an integer from 0 to 32

//Create attribute list
icmAttrListP attributeList = icmNewAttrList();
icmAddStringAttr(attributeList,"outfile", "uart.out");
icmAddStringAttr(attributeList,"infile", "input.in");
icmAddStringAttr(attributeList,"outfile", "uart.out");
icmAddStringAttr(attributeList,"infile", "input.in");
```
// Set attributes for UART. If desired, set initial values for registers. Otherwise, they will assume the default value.
icmAddUns64Attr(attributeList, "UART_CONTROL_REG", 0x0001);
icmAddUns64Attr(attributeList, "UART_INT_EN_REG", 0x0000);
icmAddUns64Attr(attributeList, "FIFO_INT_LEVEL", 23);

// Construct model path VLNV strings
const char *vlnvRoot = 0; // when null use default library
const char *OceUart = icmGetVlnvString(vlnvRoot, "arm.ovpworld.org", "peripheral", "OceUart", "1.0", "pse");
const char *pseSocketsModel = icmGetVlnvString(0, 0, 0, "OceUart", 0, "model");

// Create PSE instance and connect it to the main bus
icmPseP uart2_p = icmNewPSE("uart2", OceUart, attributeList, pseSocketsModel, "modelAttrs");
icmConnectPSEBus(uart2_p, bus1_b, "bport1", 0, 0x17000000, 0x17000fff);

// Set level of verbose detail from lowest to highest (0 to 4).
icmSetPSEdiagnosticLevel(uart2_p, 2);

11.2 Simulation launching

Océ’s UART driver was added (unmodified) to the Linux kernel release 3.2.13 acquired from kernel.org. The compilation process log can be found on Appendix IV. After the zImage is generated and the root filesystem is packed into an image file (see section 3.3.1), the platform is ready to be launched.

./oceuart_platform.Linux32.exe zImage fs.img

As it is shown in the picture below, the semihosting files are loaded first and then the LCD screen appears. The UART will start receiving data from the input.in file, together with the parity information.

Figure 11.2: Simulation launch
The line framed by the red box on Figure 11.2 indicates that the driver was able to successfully recognize the UART device at address 0x17000000, just as it was mapped in the platform and specified in the driver. This proves that the original driver is running unmodified in the platform and is able to interact with the implemented peripheral according to the company’s internal specification.

The launched simulation contains a basic set of utilities implemented with Busybox, but they are enough to monitor the behaviour of the driver. To view the data flowing into the UART, the `cat` command can be executed over `/dev/ttyHDL0`, which corresponds to the UART’s driver. The figure below shows the data being read by the peripheral from the RX source, in this case, a text file.

![Figure 11.3: UART driver](image)

As this was implemented only with proof-of-concept purposes, the way that the data is parsed was configured to enable easy test case modification (via the text file). More specific set-ups can be defined depending on the testing needs. For example, a script that generates certain patterns could be written to modify the text file accordingly for later examining how the system reacts, in an automated manner.
Observing the response of the system to unusual conditions was one of the main aspects that motivated this research. For this reason, an overflow test case was designed, as mentioned on Chapter 10: the UART will read the data normally from the text file until the overflow is triggered by a "**" character. At this point, the simulation will fill the RXFIFO with random data and will set the overflow bit on the status register. The following figure shows the test case in action.

![Overflow test case](image)

Figure 11.4: Overflow test case
12 UART proof of concept peripheral designed with SystemC TLM 2.0

Concept 3 is about creating peripherals and the platform in SystemC TLM 2.0. For testing this approach the same UART peripheral presented on Chapter 10 was modelled, so that a conclusion by comparing the feasibility and simplicity of these two concepts can be drawn in the end.

12.1 General structure

TLM does not require a specific directory structure for storing models as OVP does. It is sufficient to include the header files that contain the prototypes of peripheral functions. This concept proposes the usage of a TLM wrapper for the ARM processor and all the OVP peripherals, while the UART will be coded as a pure TLM module. The intention of this is to focus on the UART implementation and general system visibility: redesigning all the peripherals for TLM is not necessary for proving the feasibility of this approach. As usual, the processor and peripherals can communicate through the bus, which in this case is modelled by a decoder peripheral on the platform.

As SystemC uses its own scheduling model, creating the semihosting part for the UART is not a trivial task. An ideal implementation would consist on creating an additional module "Term" that handles the scheduling constraints between SystemC and Linux. This is necessary because the semihosted terminal session (or data input from a file) performs blocking read and write calls that will hang the simulation if not handled in a special way. This configuration is represented by the figure below.

![Figure 12.1: TLM platform](image)

However, implementing the semihosting part is not indispensable for observing how a TLM platform works. Because of this, the design was focused on general communication, substituting this part by a random data generator. For completeness, this chapter will briefly describe some implications of constructing the terminal module.

On Chapter 6, the importance of interfaces and channels was highlighted. The TLM standard defines these for communicating the modules in a "clear" way, although understanding all the constructs and data transport structures requires a considerable time investment at the beginning. The next section will describe the interfaces present on the TLM platform.
12.2 Interfaces
Communication is focused mainly on transaction level by defining a small set of generic and reusable TLM interfaces. In this specific case, there are three public interfaces: a bus interface represents the transactions between peripherals and the bus, UART and other peripherals; an interrupt interface handles communication between the UART and the Interrupt Controller. Finally, a terminal interface reads data from a terminal session in the host computer through semihosting.

12.2.1 Bus interface
As mentioned before, the transaction between two modules is achieved through sockets. A socket is the natural model for the bus interface to the processor and peripherals. The UART has a target socket which is registered with a blocking transport method through which a generic payload is transmitted to the bus (values are passed by reference). The bus is an interconnect component, with an initiator socket connecting to the UART’s target socket and a target socket connecting to the processor’s initiator socket.

![Figure 12.2: Bus interface](image)

12.2.2 Terminal interface
For testing the whole platform, a test bench is required. The optimal method for exercising this is by connecting an additional SystemC module that handles a semihosted terminal connection to the UART. The TX part of the terminal is modelled as a port (term.tx) which is connected to the UART’s buffer (uart.rx). The method inside the UART that is in charge of receiving data is sensitive to changes in the uart.rx buffer, this means that when data is available in the buffer, it will be stored in the UART’s FIFO and immediately after, register values and interrupt state will be updated.

An additional thread is used for reading data from the terminal session. A set of utility functions that initialize a child terminal process and register the data read thread should be implemented in the terminal module. A Linux SIGIO signal will rise while the data read thread is waiting to indicate where data is available. Because of this it is necessary to create an event handler for the SIGIO signal to notify a SystemC event upon which the thread can wait, and only after the SIGIO signal handler agreed that input is available, the read function can be called.

For ease of design and platform functionality verification purposes, the terminal interface in the current implementation is substituted by a random data generator.
12.2.3 Interrupt interface
The interrupt net between the UART and the Interrupt Controller is established through TLM analysis ports, which are specified by the TLM standard as optimal for interrupt modelling, instead of normal ports. Whenever the interrupt conditions are satisfied, an interrupt signal will be sent to the Interrupt Controller the UART’s analysis port.

12.3 Implementation
The general flow for the TLM implementation is very similar to the PSE flow, as the modelling fundamentals stay the same. The source code for the peripheral can be found on Appendix V.

After receiving data (or checking that is not possible to receive it), the peripheral’s thread will make a call to wait. During this time the control is returned to the SystemC simulation kernel. This is represented by connector A on the figure above. At this point, the blocking transport method can be triggered for the bus to perform UART read operations, or write operations to the allowed registers.
It is important to remember that operations are made via transactions, where a transaction contains information such as address, data, etc. When the blocking transport is executed, the generic payload attributes will be stripped to perform the corresponding action (read or write from/to certain register).

12.4 Test case: overflow
The overflow test case was modelled in the same way as the PSE implementation: the UART’s RXFIFO will fill and the status register’s overflow bit will be set to true when a special character is received. This can occur at connector B on Figure 12.4.

![Figure 12.6: Overflow](image)

Notes
1The data read thread could be implemented with a low priority although the data has to be processed as soon as received. The terminal module could be written to poll regularly to check if there is data available. Implementing asynchronous notifications is a better solution, because the terminal module would receive a signal whenever there is data available. The SIGIO signal is sent when a file descriptor can perform input or output.
13 Virtual platform implemented with SystemC TLM 2.0

The same test platform presented in Chapter 11 was set up in TLM to test the functionality of the UART peripheral. The source code for the TLM platform can be found on Appendix VI. The flow chart below describes the simulation execution process:

The platform is structured in a very similar way as the OVP ICM platform, except that for TLM everything is coded in C++. The overall arrangement is clearer as the peripherals are object instances (therefore, the header file for each peripheral must be included in the platform). If desired, attributes can also be defined for peripherals, just as in PSE.

The main bus has an array of initiator sockets, where each position has to be connected with the corresponding peripheral’s target socket. Interrupt connections between peripherals and the interrupt controller also have to be defined in the platform.

```cpp
//Bus connection
bus1.initiator_socket[10](uartinstance.socket);
b1.setDecode(10, 0x17000000, 0x17000fff);

//Interrupt net connection
uartinstance.irq(pic1.ir2);
```

It is also important to mention that the whole platform and peripherals were modelled using a loosely-timed approach.

### 13.1 Simulation launching

Because the kernel compilation step was already effectuated for the OVP ICM platform implementation of Chapter 11, the same zImage file can be used for launching the simulation, as it is not necessary to make any changes for the driver to recognize the TLM peripheral.

As a reminder, it is important to have the SystemC and TLM libraries installed for correct compilation and simulation launching, together with their corresponding environment variables. Information on this process can be obtained through SystemC’s official website.
The following figure shows the simulation launching process. On the simulator’s output verbose, random data is being stored on the UART’s RXFIFO. Instead of using an LCD peripheral such as in the ICM platform implementation, in this case the OVP UART model was TLM-wrapped, and a terminal connection was established to the host computer through semihosting for viewing the actual simulation output.

![Simulation launch](image)

**Figure 13.2: Simulation launch**
14 Conclusions and recommendations

This research presented an insight of Hardware and Software co-simulation techniques, focused on virtual platform implementation. The objective of researching co-simulation was to determine how feasible is to use this technique for easy test case generation, for example, triggering unusual conditions that otherwise would be very hard to replicate on a real platform, such as overflows, parity errors, etc. Additionally, another objective was to prove that co-simulation provides a better platform visibility compared to traditional testing methods.

The general structure of a virtual platform was described, for later proposing a series of concepts that intended to provide a methodology for co-simulating in-company peripherals with existing software drivers. Five of the six concepts proposed were based on the usage of the open source Open Virtual Platforms simulator.

After analysing all the proposed co-simulation solutions, the benefits and disadvantages of each approach have become clear. The following pointers summarize the most important aspects of the solution concepts:

- **Model peripherals using the Peripheral Simulation Engine (PSE) and implement the platform using OVP’s ICM (Concepts 1 and 2):** these concepts provide a fast way to implement a peripheral model and platform. The advantages of these approaches are speed and ease of design. Unfortunately, the user has to master program-specific APIs to model the platform and peripheral behaviour, although the learning curve is not very steep. There is no automated way to provide formal equivalence between VHDL and PSE models either, but functional verification is possible by monitoring the expected results. The modelling is effectuated at a high level of abstraction using traditional C programming techniques. Peripheral attributes are an excellent way to easily generate different test cases. In conclusion, PSE is a good short term solution for co-simulation because peripheral and platform writing is very straightforward, but these options are not sustainable enough as TLM for a long term project because OVP’s APIs are not an industry standard yet. It is a good option for quickly implementing a platform that will be used in a single project. If the OVP PSE/ICM approach is chosen, it is recommended to learn how to structure a platform from the Demos provided by OVP.

- **Model peripherals and platform using TLM (Concepts 3 and 4):** these concepts are a very methodological way to implement the whole system because the TLM standard is quite abstract, although it is officially accepted by the EDA community and the IEEE. The learning curve is very pronounced and a good amount of time is required to understand the standard’s protocols. As a trade-off, the standard provides a safer way to implement the behaviour compared to PSE, as it is more structured. The main benefits of this approach are that it is also open source, it has a growing community support and peripherals are modelled at a high level of abstraction, skipping unnecessary implementation details. Currently, translators are being researched for providing automatic conversion between VHDL and SystemC TLM models. In conclusion, TLM is the best open source solution as it is backed up by the EDA ecosystem and the standard is continuously being updated, as a proof of this, big companies include TLM models as part of their commercial solutions.
At this moment the VHDL must be recoded manually, but when abstraction translation tools become available, setting up a platform in TLM should be a very quick task and model synchronization wouldn’t be a problem anymore. Therefore, in the future, TLM can be a reliable option for supporting a long term co-simulation project. If this approach is chosen, it is better to spend time learning its constructs instead of spending the efforts on understanding advanced SystemC programming techniques. Knowing the basic constructs, datatypes and interfaces that make up SystemC is enough. Keeping updated with news regarding abstraction level translators can be important to decide when to choose TLM for a permanent solution.

- **Use VHDL models directly, define platform in SystemC through ModelSim (Concept 5):** an easy way to implement the platform. The main advantage is that the VHDL is used as-is, but the benefit is countered by the fact that mixing high level of abstraction models (such as the TLM-wrapped processor) together with RTL models will yield a very slow simulation, rendering this concept unusable for test case generation purposes. In conclusion, this concept provides unacceptable performance for co-simulation purposes, although it is easy to implement and no model recoding is needed.

- **Commercial EDA tools (Concept 6):** The easiest way for setting up a platform and analyzing software output. Various commercial tools have an integrated library of TLM models, which can be useful to quickly setup a platform. They provide breakpoint/testing features and with the information presented in a graphical way, it is easier to understand what is happening in the simulation. They are backed up by large companies, thus they have good community support. The downside of this approach is the cost of acquiring a software license. In conclusion, commercial tools are very well equipped for error tracing and test case generation purposes. Depending on how big is the need for accelerating the HW/SW interaction process, a commercial tool could be considered. It is recommended to always follow closely the EDA companies to check if low-cost licenses for co-simulation tools are available and make use of trial versions provided by these companies if available.

**General conclusion:**

Hardware and Software co-simulation is a relatively new technique in the electronics industry, as such, it is constantly changing. For this reason, there is no single best way to do co-simulation: all options present different advantages and disadvantages, the methodology to choose will greatly depend on the engineers’ needs and available resources.

The solutions involving the usage of OVPsim’s APIs appear to be the most cost/benefit effective solution at the moment; however, the downside is that there is a need to rewrite the VHDL into PSE models and keep them in sync: this is a task that requires user effort and an initial time investment.

By observing how the TLM standard is currently gaining more support from the EDA community, it is safe to assume that in a near future, designing a platform with SystemC TLM 2.0 will be a more sustainable and easy option as accurate automated translators (between RTL and TLM) start to become available: this means that the tester can get the speed benefits of co-simulation without having to invest a considerable amount of time in setting up the peripherals and platform, which is the current case with open source alternatives.
Appendix

I. OVPsim installation log
Kubuntu 10.04 x86 was set up as the working environment for simulation and platform design purposes on an unique partition of a Core 2 vPro system. A samba share was configured to access the user’s data from network locations. Additionally, a Core i5 vPro system with Windows 7 was used as a research and multipurpose workstation.

Windows 7
OVPsim is only officially supported on x86 Windows XP architectures, but previous attempts of installations in Windows 7 system have been documented on OVP user forums. With these considerations, an implementation on Linux system was observed as the best choice for researching the OVP simulator.

Linux
OVP simulator and model files are available on the official website in a pre-compiled way and tested on an x86 Linux Fedora Core 4 system, but Makefiles are provided to compile all the files again, if needed. A compressed file is available for download including the necessary files to build a platform and run some examples. A configuration script,

<installDir>/Imperas/bin/setup.sh

which needs to be evaluated using the source command is provided to set up the environment variables for the current user instead of having to export them manually. After that, a setup function “setupImperas” is created within the current environment to establish the rest of the simulator’s configuration, where an “-m32” flag can be enabled for compiler compatibility reasons when working on 64 bit systems.

These environment variables allow easier platform and Makefile writing, because they point to the root of the tools installation, set the host OS type and architecture, point to the root of the compiled models libraries in the OVPsim installation, and specify which simulator will be loaded at runtime, among other functions.

An important remark is that they are automatically deleted each time the terminal session ends, so for convenience, the .bashrc script was modified with the purpose of executing this script every time a terminal session is started. OVP provides a set of Makefiles on

<installDir>/Imperas/ImperasLib/source/buildutils

that allow to compile platforms, peripherals, among other devices. At the beginning, the whole installation was attempted on an x86_64 system, but later on there were problems regarding compiler and architecture compatibility.

To compile successfully, it became necessary to start editing each one of the Makefiles provided in order to add an “-m32” flag for the gcc compiler to be able to make the linking correctly. Makefile editing began to turn into a tedious task to compile successfully, for this reason the whole working environment was reinstalled with a Kubuntu x86 version.
There are two types of simulators available. Imperas, which is OVP’s parent company, provides its own simulator, CpuManager\(^1\), which has a broader set of functions in comparison with OVPsim.

**Toolchains**

For compiling an application for a specific processor model and its variants, various toolchains are provided on the OVPWorld website. The ARM and OR1K toolchains were downloaded with the purpose of rebuilding examples for these processor models. Also, the PSE (Peripheral Simulation Engine) toolchain was downloaded in order to do some peripheral testing.

**Setting up licensing**

OVPsim requires a FLEXnet\(^2\) license file to be able to run. Licenses are freely provided for private use, academic research, academic teaching and other educational purposes. All the licenses are on a 90 day basis, and free renewal is required after this period. For commercial organizations, a free trial license of 90 days is provided and to continue usage after that period, an official license must be acquired. A single license of OVPsim costs $3,600 per year.

Trial licenses were requested on the OVPworld.org website for four workstations, and they were received quickly. However, a problem arose when trying to run OVPsim with the new licenses: OVP needs internet connection to validate a license but as the host computers had tunneled internet access through a proxy, OVPsim was not able to make the verification. Thus this issue was reported to OVPworld support and they responded to the request by providing another license without the online verification requirement.

**ARM Bare Metal platform from a general perspective**

An ARM processor bare metal (no OS) platform demo was examined with the purpose of acquiring a better understanding of the way OVPsim works. Two files are important for compiling the necessary binaries to run a bare metal platform:

- **Platform file:** defines how the models are connected inside the platform, defines address spaces for each model and sets attributes for the processor and paths for file locations.
- **Application file:** the actual program to be run by the simulated processor.

![Figure A1: ARM Bare Metal platform simulation structure](image)

A set of four test applications are provided in the ARM Bare Metal demo. The platform and the examples were recompiled before running them. Scripts are provided for auto-compiling and running the examples, but the compilation was done manually to identify the relationship between files.
Command to start simulation:

```
./OVPsim_arm_linux32.exe fibonacci.ARM7.elf
```

It is important to understand that the actual simulator is not embedded on OVPsim_arm_Linux32.exe, but this file relies on dynamically linked libraries/shared objects (where the simulator kernel is) to be able to start the actual simulation. In order to know how to internally assemble the platform, time needs to be spent on reading OVPsim and CpuManager user’s guide.

**ARM Bare Metal platform from the C perspective**

As OVPsim lacks VHDL support, it is necessary to think of alternatives on how to bind the OVP processor model with the company’s peripheral models, which is going to be the focus of the next chapter. For this reason, having an idea of what parts make up a platform file is vital to be able to identify what aspects influence the feasibility of possible co-simulation approaches. The platform described in the last section will be briefly analyzed here:

```c
#include<stdio.h>
#include<string.h>
#include "icm/icmCpuManager.h"
#define SIM_ATTRS (ICM_ATTR_RELAXED_SCHED)

int main(int argc, char **argv)
{
    //Check if program was run with the correct arguments
    if(argc!=2)
    {
        icmPrint("Incorrect arguments"); return -1;
    }

    //Store name of elf file
    const char *appName = argv[1];

    /*Initialize OVPsim, define platform attributes. OVPsim user’s manual provides a list of available attributes.*/
    icmInit(ICM_VERBOSE|ICM_STOP_ON_CTRL_C,0,0);

    //Build path of processor model location
    const char *arm7model = icmGetVlnvString(NULL,"arm.ovpworld.org","processor","arm","1.0","model");
    const char *arm7Semihost = icmGetVlnvString(NULL,"arm.ovpworld.org","semihosting","armNewlib","1.0","model");

    //Define list of attributes for the processor instance
    icmAttrListP icmAttr = icmNewAttrList();
    icmAddStringAttr(icmAttr, "compatibility", "gdb");
    icmAddStringAttr(icmAttr, "variant", "ARM7TDMI");
    icmAddStringAttr(icmAttr, "UAL", "1");
```

Figure A2: Simulation statistics
// Instance a processor
ICMProcessorP myProc = icmNewProcessor(
    "CPU1",       // CPU Name
    "arm",        // CPU type
    0,            // CPU cpuID
    0,            // CPU model flags
    32,           // Address bits
    arm7Model,    // Model path
    "modelAttrs", // Other attributes
    "SIM_ATTRS",  // Simulator attributes
    icmAttr,      // Processor attributes
    arm7Semihost, // Semihosting path
    "modelAttrs"  // Other attributes
);

/* OVP peripherals and buses can be instanced here, make appropriate connections.
Important to note that only PSE peripherals can be called from here. It is not possible to
make instances of SystemC, VHDL or other non-OVP components directly from the OVP platform
file. Instead, OVP provides functions to map external memory and define watchpoints. If an
OVP processor model wants to be used together with other simulators/interfaces, a TLM2.0
wrapper must be written. */

// Load application (which was fed as a parameter)
if(!icmLoadProcessorMemory(myProc, appName, False, False, True)
    return -1;

// Start simulation
ICMProcessorP start = icmSimulatePlatform();

// Call simulation end when execution finishes
icmTerminate();
return 0;
}

The actual application to be run is coded in normal C as long as it is able to be cross-compiled
successfully (with the toolchain of the processor model used) and does not depend from
auxiliary files.

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    printf("\n");
    printf("**************************\n");
    printf("Hw/SW CoSimulation project\n");
    printf("Venlo, February 2012\n");
    printf("**************************\n");
    printf("\n");
}
Notes

1 CPUManager is the commercial simulator product from Imperas. OVPsim is the free version intended for non-commercial usage.

2 FLEXnet is as software license manager that supports node-locked and floating licenses.
II. UART source code

/******* user.c - HDL_UART implementation *******
* Written by Manuel Munoz, Yu Cheng
* for Oce Technologies B.V. as part of
* graduation project: HW/SW Cosimulation
* March, 2012
*
* ****************************************************
* DIAGNOSTIC LEVELS: Level of detail for UART output (simulator verbose).
* Levels are inclusive in the order stated below.
* +(DIAG_HIGH)
* -Current interrupt state when updating interrupts.
* -Register initialization
* -Entered character receive loop.
* -Constructing UART
* +(DIAG_MEDIUM)
* -Interrupt state has changed
* -Detail of data and parity inserted into RXFIFO when an overflow is triggered.
* -Trigger of overflow bit
* -Notification when special overflow trigger (*) has been reached.
* -Detail of data and parity inserted into RXFIFO
* +(DIAG_LOW)
* -Attempt to open socket connection on a port.
* -UART crash messages.
* ****************************************************

// Defines, includes and variables for internal state
#include <string.h>
#include <stdlib.h>
#include "pse.igen.h"
#define BPORT1 0
#include "pse.macro.igen.h"
#include "psesockets.h"
#define DEFAULT_RX_DELAY 20000
#define DIAG_LOW (diagnosticLevel>=1)  // Set types of diagnostic levels
#define DIAG_MEDIUM (diagnosticLevel>=2)
#define DIAG_HIGH (diagnosticLevel>=3)
#define TOP (256*1024)  // Stack size for main loop thread
#define THRESHOLD thres  // FIFO_INT_LEVEL
#define status bport1_ab_data.status
#define control bport1_ab_data.control
#define type bport1_ab_data.type
#define version bport1_ab_data.version
#define int_id bport1_ab_data.int_id
#define int_en bport1_ab_data.int_en
#define int_level bport1_ab_data.int_level

Uns8 stack1[TOP];
static Uns32 rx_fifo[32];
static Uns32 pf[32];
static Int32 chars_in_fifo=0;
static Int32 fifo_read_pos=0;
static Int32 channel;
static Uns32 portnum;
static bhmEventHandle charReceived;
static Int32 thres;
/**FUNCTION: void updateInterrupt()**

*DESCRIPTION: Sets RXFIFO interrupt identification bit according to threshold level, also checks if the current interrupt state is different to the past state, if yes, change it and write the value to the interrupt net.*

*VARIABLES:*

- Bool getInterruptStateRX: Store the value of the calculated interrupt at that instant.
- static Bool current: Used to store the state of the interrupt.

```c
void updateInterrupts()
{
    Bool getInterruptStateRX;
    static Bool current = 0;

    if(chars_in_fifo<THRESHOLD)
        int_id.bits.bit0=0;
    else
        int_id.bits.bit0=1;

    getInterruptStateRX = (int_en.bits.bit1 & int_id.bits.bit0) && 1;

    if(DIAG_HIGH)
        bhmMessage("I", "OceUart_Update", "Int %d -> %d (int_id 0x%04x and int_en 0x%04x)",
                    current, getInterruptStateRX, int_id.value, int_en.value);

    if(getInterruptStateRX != current) //Change current interrupt state if its different
    {
        current = getInterruptStateRX;
        if(DIAG_MEDIUM)
            bhmMessage("I", "OceUart_Interrupt", "%d", current);

        ppmWriteNet(handles.irq, current);
    }
}
```

/**FUNCTION: Bool calculateParity(Uns32 c, Uns32 p)**

*DESCRIPTION: Calculates if there is a parity error of the received data C against the parity bit, depending on the parity mode set on the control register. Returns 0 if OK, 1 if ERROR.*

*VARIABLES:*

- Int32 i: index variable
- Int32 sum: store count of 1's in databyte
- Bool asciiToBool: store parity bit, it is 0 if received parity is (0x30 [0 in ASCII]).

```c
Bool calculateParity(Uns32 c, Uns32 p)
{
    Int32 i, sum=0;
    Bool asciiToBool;

    for(i=0;i<8;i++)
        sum+=c>>i&1;

    if(p == 0x30)
        asciiToBool=0;
    else
        asciiToBool=1;

    if(control.bits.bit1 == 0) //Odd mode
    {
        if(sum%2 == asciiToBool)
            return 1;
    ```
else return 0;
}
else
{
  if(sum%2 == asciiToBool)
    return 0;

  else return 1;
}

 CALLBACK: PPM_READ_REG_CB(readintID)
* DESCRIPTION: Callback when a read access of UART_INT_REG address (OFFSET+0x0E) is
  generated from software side. Sets UART_RX_INT_ID interrupt to 0.
*******************************************************************************/
PPM_REG_READ_CB(readintID)
{
  int_id.bits.bit0=0;  //Disable interrupt when reading from INT_ID register
  updateInterrupts();
  return *(Uns16*)user;
}

 CALLBACK: PPM_REG_WRITE_CB(writeinten)
* DESCRIPTION: Callback when a write access is attempted by software on the UART_INT_EN
  address.
*******************************************************************************/
PPM_REG_WRITE_CB(writeinten)
{
  *(Uns16*)user = data;
  updateInterrupts();
}

 CALLBACK: PPM_REG_READ_CB(readRX)
* DESCRIPTION: Callback when a read access of UART_TXDATA_REG address (OFFSET+0x0A) is
  generated from software side. Checks if the RXFIFO/PFIFO read position reached the end of the
  FIFO to start again from the beginning. If parity is enabled, will calculate data parity
  of the data retrieved and set the ERROR/NOERROR in the status register; else it will be
  NOERROR by default. If there was an overflow status on status register, it will be
  cleared.
* VARIABLES:
*  Uns32 c: Store character from current read position in RXFIFO
*  Uns32 p: Store parity bit of the data read in RXFIFO.
*  Bool parityCheck: 1 if data parity against parity bit is error.
*******************************************************************************/
PPM_REG_READ_CB(readRX)
{
  Uns32 c = rx_fifo[fifo_read_pos];
  Uns32 p = pf[fifo_read_pos];

  if(chars_in_fifo > 0)
  {
    chars_in_fifo--;
    if(++fifo_read_pos == 32)
      fifo_read_pos = 0;
  }
if(chars_in_fifo == 0)
    status.bits.bit0 = 1;

if(control.bits.bit0 == 1) //Calculate data parity and set status reg if parity is enabled.
{
    Bool parityCheck;
    parityCheck = calculateParity(c,p);
    if(parityCheck == 0)
        status.bits.bit4 = 0;
    else
    status.bits.bit4 = 1;
}
else
    status.bits.bit4 = 0;   //Set parity error bit to NOERROR if parity is not enabled.

if(status.bits.bit1 == 1)   //If RXFIFO was overflow, now it is not.
    status.bits.bit1 = 0;
updateInterrupts();
return c;

void serialInput()
{
    channel = 0;
    portnum = 0;
    char outPath[1024];
    char srcPath[1024];
    Uns32 log = 0;
    Bool portSet = bhmIntegerAttribute("portnum", &portnum);
    Bool outSet = bhmStringAttribute("outfile", outPath, sizeof(outPath));
    Bool srcSet = bhmStringAttribute("infile", srcPath, sizeof(srcPath));
    bhmIntegerAttribute("log", &log);

    if(portSet & srcSet)
        bhmMessage("F", "OceUart", "Cannot have input both in port and file! Quitting simulator");

    if(portSet & DIAG_LOW)
        bhmMessage("I", "OceUart", "Attempting to open socket on port:Xd\n", portnum);
    Uns32 *portp = portSet ? &portnum : NULL;  //Create pointers for infile and outfile
    char *out = outSet ? outPath : NULL;
    char *src = srcSet ? srcPath : NULL;
    channel = serOpenBlocking(portp, out, src, log);  //Open channel connection through semihosting library
    if(channel < 0)
        bhmMessage("F", "OceUart", "Serial channel could not be opened");
/*******************************************************************************
* FUNCTION: void initialState()
* DESCRIPTION: Set all registers to default. If attributes are specified in the platform,
*               retrieve them and set the register to the specified values. Other starting
*               configurations can also be defined here. If register is not here, it is
*               0x0000 by default.
* VARIABLES:
*        Bool set: check if the attribute was defined in the platform.
*        Uns32 hold: stores the retrieved value from the platform.
*******************************************************************************

void initialState()
{
    if(DIAG_HIGH)
        bhmMessage("I","OceUart","Initializing registers");

    Bool set;
    Uns32 hold;

    type.value |= UART_TYPE_DEFAULT;       //UART_TYPE
    set = bhmIntegerAttribute("UART_TYPE", &hold);
    if(set) type.value = hold;

    version.value |= 0x01;                   //UART_VERSION
    set = bhmIntegerAttribute("UART_VERSION", &hold);
    if(set) version.value = hold;

    status.value |= UART_STATUS_REG_DEFAULT; //UART_STATUS_REG
    set = bhmIntegerAttribute("UART_STATUS_REG", &hold);
    if(set) status.value = hold;

    int_level.value |= UART_INT_LEVEL_DEFAULT; //UART_INT_LEVEL
    set = bhmIntegerAttribute("UART_INT_LEVEL", &hold);
    if(set) int_level.value = hold;

    set = bhmIntegerAttribute("UART_CONTROL_REG", &hold); //UART_CONTROL_REG
    if(set) control.value = hold;

    set = bhmIntegerAttribute("UART_INT_EN_REG", &hold); //UART_INT_EN_REG
    if(set) int_en.value = hold;

    thres = 24;                             //Project setting FIFO_INT_LEVEL
    set = bhmIntegerAttribute("FIFO_INT_LEVEL", &hold);
    if(set) thres = hold;
}

/*******************************************************************************
* FUNCTION: Bool can_receive()
* DESCRIPTION: Check if RXFIFO is not overflow and is not full.
*******************************************************************************

Bool can_receive()
{
    if(!(status.value & RXFO) && (chars_in_fifo<32))
        return 1;
    else
        return 0;
}

/*******************************************************************************
* FUNCTION: void overflowRXFIFO()
* DESCRIPTION: Fill RXFIFO with data until it is full and trigger overflow flag. Parity bits
*               for this datas are assumed to be correct.
* VARIABLES:
*        Int32 c: Buffer for data used to fill RXFIFO. Currently set at 0x5a (Z).
*        Int32 parity: Buffer used to store correct parity.
*        Int32 slot: See function receiveBytes();
*******************************************************************************
void overflowRXFIFO()
{
    while(can_receive())
    {
        Int32 c, parity;
        Int32 slot = fifo_read_pos + chars_in_fifo;
        if(slot>=32)
            slot-=32;
        c = 0x5a;
        if(control.bits.bit1==0)
            parity = 0x31;
        else
            parity = 0x30;
        if(DIAG_MEDIUM)
            bhmMessage("I","OceUart","RANDOM in slot %d is %x%c and parity %x%c
: ", slot, (char)c,(char)c,(char)parity,(char)parity);
        rx_fifo[slot] = c;
        pf[slot] = parity;
        chars_in_fifo++;
        if(chars_in_fifo>0)
            status.bits.bit0 = 0;
        else
            status.bits.bit0 = 1;
        if(chars_in_fifo>THRESHOLD)
            int_id.bits.bit0 = 1;
        else
            int_id.bits.bit0 = 0;
    }
    status.bits.bit1 = 1;
    if(DIAG_MEDIUM)
        bhmMessage("I","OceUart","Overflow bit has been triggered!");
    updateInterrupts();
}
overflowRXFIFO();

} else {
    if(DIAG_MEDIUM)
    {
        bhmMessage("I", "OceUart", "RXFIFO[\%d] = 0x\%x(\%c)", slot, (char)value,
(char)value);
        if(control.bits.bit0 == 1)
            bhmMessage("I", "OceUart", "PFIFO[\%d] = 0x\%x(\%c)", slot,
(char)parity, (char)parity);
    }
    rx_fifo[slot] = value;
    pf[slot] = parity;
    chars_in_fifo++;
}

if(chars_in_fifo>0)
    status.bits.bit0 = 0;
else
    status.bits.bit0 = 1;

if(chars_in_fifo>THRESHOLD)
    int_id.bits.bit0 = 1;
else
    int_id.bits.bit0 = 0;

updateInterrupts();
}

/*******************************************************************************
* FUNCTION: void getChars(void *user)
* DESCRIPTION: Peripheral’s main loop, delays the thread a ”d” amount of time to yield control
* to the processor and do other tasks. Checks if RXFIFO can receive data, if yes, then reads parity (if enabled) and data from the input source defined on serialInput() and stores on PFIFO (parity if enabled, dummy value ’X’ if disabled) and RXFIFO accordingly; else it will keep yielding control to processor until RXFIFO can receive data.
* VARIABLES:
* double d: Amount of time in microseconds to yield control to processor.
* Uns8 c: Buffer for data byte.
* Uns8 parity: Buffer for parity bit (actually receives a whole byte containing 0x30 or 0x31 for 0 or 1.)
* CONSTANT VALUES:
* 0x58: Dummy character ’X’ to keep PFIFO aligned with RFIFO when parity is disabled.
*******************************************************************************

void getChars(void *user)
{
    while(1)
    {
        double d = DEFAULT_RX_DELAY;
        bhmWaitDelay(d);

        if(can_receive())
        {
            Uns8 c, parity;

            if(DIAG_HIGH)
                bhmMessage("I", "OceUart", "ENTERED RECEIVE LOOP");

            if(control.bits.bit0 == 1)
{  
  if (serRead(channel, &parity, 1))  
  {  
    bhmTriggerEvent(charReceived);  
    serRead(channel, &c, 1);  
    bhmTriggerEvent(charReceived);  
    receiveBytes(c, parity);  
  }  
  serRead(channel, &c, 1);  
}
else  
{
  if (serRead(channel, &c, 1))  
  {  
    bhmMessage("I", "OceUart", "PBIT LOOP");  
    bhmTriggerEvent(charReceived);  
    receiveBytes(c, 0x58);  
  }  
}

/***************************************************************
* CALLBACK: PPM_DESTRUCTOR_CB(closeDown)
* FUNCTION: This callback will be called by the simulator on finishing, if a port connection
* was established it will close the channel.
***************************************************************/
PPM_DESTRUCTOR_CB(closeDown)
{
  if (channel >= 0)
  serClose(channel);
}

/***************************************************************
* CALLBACK: PPM_CONSTRUCTOR_CB(constructor)
* FUNCTION: Entry point for peripheral execution, this callback must be called by the
* peripheral's main(). After creating all registers, setting data communication
* channel and writing default values to registers, a thread that runs the
* peripheral's execution loop is created.
***************************************************************/
PPM_CONSTRUCTOR_CB(constructor)
{
  if (DIAG_HIGH)
    bhmMessage("I", "UART_CONSTRUCTOR", "Starting UART constructor");

  periphConstructor();
  serialInput();
  initialState();

  bhmCreateThread(getChars, NULL, "getChars", &stack1[TOP]);
  charReceived = bhmCreateNamedEvent("RX", "Character received!");
}
#include "pse.igen.h"

bport1_ab_dataT bport1_ab_data;
handlesT handles;

/******************************************************************************
* Diagnostic level callback */
******************************************************************************

Uns32 diagnosticLevel;

static void setDiagLevel(Uns32 new)
{
    diagnosticLevel = new;
}

/******************************************************************************
* Generic callbacks */
******************************************************************************

static PPM_VIEW_CB(view16) { *(Uns16*)data = *(Uns16*)user; }
static PPM_READ_CB(read_16) { return *(Uns16*)user; }
static PPM_WRITE_CB(write_16) { *(Uns16*)user = data; }

/******************************************************************************
* Bus Slave Ports */
******************************************************************************

static void installSlavePorts(void)
{
    handles.bport1 = ppmCreateSlaveBusPort("bport1", 4096);
}

/******************************************************************************
* Memory mapped registers */
******************************************************************************

static void installRegisters(void)
{
    ppmCreateRegister("type", "UART_TYPE", handles.bport1, 0, 2, read_16, write_16, view16, &bport1_ab_data.type.value, True); //R/W
    ppmCreateRegister("version", "UART_VERSION", handles.bport1, 2, 2, read_16, write_16, &bport1_ab_data.version.value, True); //R/W
    ppmCreateRegister("status", "UART_STATUS_REG", handles.bport1, 4, 2, read_16, 0, view16, &bport1_ab_data.status.value, True); //R
    ppmCreateRegister("control", "UART_CONTROL_REG", handles.bport1, 6, 2, read_16, write_16, &bport1_ab_data.control.value, True); //R/W
    ppmCreateRegister("txdata", "UART_TXDATA_REG", handles.bport1, 8, 2, 0, write_16, &bport1_ab_data.txdata.value, True); //W
    ppmCreateRegister("rxdata", "UART_RXDATA_REG", handles.bport1, 10, 2, readRX, 0, view16, &bport1_ab_data.rxdata.value, True); //R
    ppmCreateRegister("int_en", "UART_INT_EN_REG", handles.bport1, 12, 2, read_16, writeinten, view16, &bport1_ab_data.int_en.value, True); //R/W
    ppmCreateRegister("int_id", "UART_INT_ID_REG", handles.bport1, 14, 2, readintID, 0, view16, &bport1_ab_data.int_id.value, True); //R/W
    ppmCreateRegister("bd_l", "UART_BAUD_DIVISOR_L_REG", handles.bport1, 16, 2, write_16, &bport1_ab_data.bd_l.value, True); //W
    ppmCreateRegister("bd_h", "UART_BAUD_DIVISOR_H_REG", handles.bport1, 18, 2, read_16, write_16, &bport1_ab_data.bd_h.value, True); //R/W
    ppmCreateRegister("int_level", "UART_INT_LEVEL", handles.bport1, 20, 2, read_16, write_16, &bport1_ab_data.int_level.value, True); //R/W
}
static void installNetPorts(void)
{
    handles.irq = ppmOpenNetPort("irq");
}

PPM_CONSTRUCTOR_CB(periphConstructor)
{
    installSlavePorts();
    installRegisters();
    installNetPorts();
}

int main(int argc, char *argv[])
{
    diagnosticLevel = 1;
    bhmInstallDiagCB(setDiagLevel);
    constructor();
    bhmWaitEvent(bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION));
    closeDown();
    return 0;
}
/*******************************************************************************/
*pse igen.h - HDL_UART register structure and other prototypes
* Written by Manuel Munoz, Yu Cheng
* for Oce Technologies B.V. as part of
* graduation project: HW/SW Cosimulation
* March, 2012
*******************************************************************************/
#ifndef PSE_IGEN_H
#define PSE_IGEN_H
#include "peripheral/impTypes.h"
#include "peripheral/bhm.h"
#include "peripheral/ppm.h"

//////////////////////////////// Externs ///////////////////////////////////
extern Uns32 diagnosticLevel;

//////////////////////////////// Register data declaration //////////////////////
typedef struct bport1_ab_dataS
{
    union
    {
        Uns16 value;
    } type;

    union
    {
        Uns16 value;
    } version;

    union
    {
        Uns16 value;
    } status;

    union
    {
        struct
        {
            unsigned bit0 : 1;
            unsigned bit1 : 1;
            unsigned bit2 : 1;
            unsigned bit3 : 1;
            unsigned bit4 : 1;
        } bits;
    } control;

    union
    {
        Uns16 value;
    } txdata;

}
Uns16 value;
)
union
{
    
| unsigned bit0 : 1;
| unsigned bit1 : 1;
| unsigned bit2 : 1;
|
}
}
union
{
    
| unsigned bit0 : 1;
| unsigned bit1 : 1;
|
}
}
union
{
    
| unsigned bit0 : 1;
|
}
}
union
{
    
| unsigned bit0 : 1;
|
}
}
union
{
    
| unsigned bit0 : 1;
|
}
}
union
{
    
| unsigned bit0 : 1;
|
}

bport1_ab_dataT, *bport1_ab_dataTP;

extern bport1_ab_dataT bport1_ab_data;

typedef struct handlesS {
    void *bport1;
    ppmNetHandle irq;
} handlesT, *handlesTP;

extern handlesT handles;

/*DEFINE SPECIAL R/W CALLBACKS HERE*/

PPM_REG_WRITE_CB(writeinten);
PPM_REG_READ_CB(readRX);
PPM_REG_READ_CB(readintID);
PPM_CONSTRUCTOR_CB(periphConstructor);
PPM_DESTRUCTOR_CB(periphDestructor);
PPM_CONSTRUCTOR_CB(constructor);
PPM_DESTRUCTOR_CB(closeDown);

#endif
#ifndef PSE_MACRO_IGEN_H
#define PSE_MACRO_IGEN_H

#define BPORT1_ERROR BPORT1 is undefined. It needs to be set to the port base address
#endif

#ifdef PSE_MACRO_IGEN_H
#define BPORT1_AB_UART_TYPE (BPORT1 + 0x0) //UART_TYPE
#define UART_TYPE_DEFAULT (0x12) //Fixed type
#define BPORT1_AB_UART_VERSION (BPORT1 + 0x02) //UART_VERSION
#define BPORT1_AB_UART_STATUS_REG (BPORT1 + 0x04) //UART_STATUS_REG
#define RXFE (0x1 << 0) //RX FIFO is empty
#define RXFO (0x1 << 1) //RX FIFO is overflow
#define TXFE (0x1 << 2) //TX FIFO is empty
#define TXFF (0x1 << 3) //TX FIFO is full
#define PARITY_ERROR (0x1 << 4) //Parity error
#define UART_STATUS_REG_DEFAULT (0x000D) //Initialization values for status reg
#define BPORT1_AB_UART_CONTROL_REG (BPORT1 + 0x06) //UART_CONTROL_REG
#define PARITY_ENABLE (0x1 << 0) //Parity mode enabled
#define PARITY_EVEN (0x1 << 1) //Parity is even
#define BPORT1_AB_UART_TXDATA_REG (BPORT1 + 0x08) //UART_TXDATA_REG
#define BPORT1_AB_UART_RXDATA_REG (BPORT1 + 0x0A) //UART_RXDATA_REG
#define BPORT1_AB_UART_INT_EN_REG (BPORT1 + 0x0C) //UART_INT_EN_REG
#define BPORT1_AB_UART_INT_ID_REG (BPORT1 + 0x0E) //UART_INT_ID_REG
#define BPORT1_AB_UART_BAUD_DIVISOR_L_REG (BPORT1 + 0x10) //UART_BAUD_DIVISOR_L
#define BPORT1_AB_UART_BAUD_DIVISOR_H_REG (BPORT1 + 0x12) //UART_BAUD_DIVISOR_H
#define BPORT1_AB_UART_INT_LEVEL (BPORT1 + 0x14) //UART_INT_LEVEL
#define UART_INT_LEVEL_DEFAULT (0x0018) //Initialization value for interrupt level reg
#endif
III. OVP ICM Platform source code

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "icm/icmCpuManager.h"

//Variables for storing simulation launching arguments
static Bool noGraphics = False;
static Bool wallClock = False;
static Bool programLoad = False;
static Uns32 uartPort = 0;
static Bool connectUartPort = False;
static Float stopafter = 0.0;
static char variant[32] = {"ARM926EJ-S"};

//String to print indicating the order of arguments
static const char* arguments = "%[variant <ARM cpu model variant>] [wallclock]n"
"[nographics] [nolinux] [attach <port number>]\n"
"[stopafter <seconds to stop after>]";

void parseArgs(int firstarg, int argc, char **argv) {
    for(int i=firstarg; i < argc; i++) {
        if(strcmp(argv[i], "wallclock") == 0) {
            wallClock = True;
        }
        else if(strcmp(argv[i], "nographics") == 0) {
            noGraphics = True;
        }
        else if(strcmp(argv[i], "nolinux") == 0) {
            programLoad = True;
        }
        else if(strcmp(argv[i], "attach") == 0) {
            sscanf(argv[++i], "%d", &uartPort);
            connectUartPort = True;
        }
        else if(strcmp(argv[i], "stopafter") == 0) {
            sscanf(argv[++i], "%f", &stopafter);
        }
        else if(strcmp(argv[i], "variant") == 0) {
            sscanf(argv[++i], "%s", variant);
        }
        else {
            fprintf(stderr, "invalid argument %s\\n", argv[i]);
            exit(1);
        }
    }
}

void createPlatform(char *kernelFile, char *ramDisk, Uns32 icmInitAttrs, Uns32 icmAttrs) {
    // Sets initial platform attributes and initialises all the elements in the platform: main bus, memory bus, processor, core module, interrupt controllers, etc.
    // Initialises RAM memory and defines IRQ/FIQ net connections for the processor, and IRQ connections between peripherals and the interrupt controller.
}
void createPlatform(char *kernelFile, char *ramDisk, Uns32 icmInitAttrs, Uns32 icmAttrs) {
    icmPrintf("Init: icmInitAttrs 0x%x icmAttrs 0x%x\n", icmInitAttrs, icmAttrs);
    icmPrintf("Files: kernelFile %s ramDisk %s\n", kernelFile, ramDisk);
    icmPrintf("Options: wallclock %u nographics %u nolinux %u attach %d variant %s\n",
               wallClock, noGraphics, programLoad, uartPort, variant);

    icmInit(icmInitAttrs, 0, 0);
    icmSetSimulationTimeSlice(0.001000);
    if (wallClock) icmSetWallClockFactor(3.0);
    icmSetPlatformName("ArmIntegratorCP");

    //Platform attributes
    icmBusP bus1_b = icmNewBus("bus1_b", 32);
    icmBusP membus_b = icmNewBus("membus_b", 32);

    //**************************ARM processor**************************
    //Create VLNV string
    const char *arm1_path = icmGetVlnvString(0,"arm.ovpworld.org",0,"arm",0,"model");

    //Processor's attribute list
    icmAttrListP arm1_attr = icmNewAttrList();
    icmAddStringAttr(arm1_attr, "variant", variant);
    icmAddStringAttr(arm1_attr, "compatibility", "ISA");
    icmAddStringAttr(arm1_attr, "showHiddenRegs", "0");
    icmAddDoubleAttr(arm1_attr, "mips", 200.000000);
    icmAddStringAttr(arm1_attr, "endian", "little");

    //Create processor instance
    icmProcessorP arm1_c = icmNewProcessor("arm1", "arm", 0, 0x0, 32, arm1_path, "modelAttrs", 0x20, arm1_attr, 0, 0);

    //Connect processor to bus1_b and use it as data and instruction bus.
    icmConnectProcessorBusses(arm1_c, bus1_b, bus1_b);

    //**************************Core module**************************
    //Create VLNV string
    const char *cm_path = icmGetVlnvString(0,"CoreModule9x6",0,"pse");

    //Peripheral's attribute list
    icmAttrListP cm_attr = icmNewAttrList();
    icmPseP cm_p = icmNewPSE("cm", cm_path, cm_attr, 0, 0);

    //Memory-map the port 'bport1' from the core module to the main bus.
    icmMapPSEBus(cm_p, bus1_b, "bport1", 0, 0x10000000, 0x10000fff);
Hardware and Software Co-simulation

```c
//**************************Interrupt controller 1**************************
const char *pic1_path = icmGetVlnvString(0,0,0,"IntICP",0,"pse");
icmAttrListP pic1_attr = icmNewAttrList();
icmPseP pic1_p = icmNewPSE("pic1",pic1_path,pic1_attr,0,0);

icmConnectPSEBus( pic1_p, bus1_b, "bport1", 0, 0x14000000, 0x14000fff);

//**************************Interrupt controller 2**************************
const char *pic2_path = icmGetVlnvString(0,0,0,"IntICP",0,"pse");
icmAttrListP pic2_attr = icmNewAttrList();
icmPseP pic2_p = icmNewPSE("pic2",pic2_path,pic2_attr,0,0);

icmConnectPSEBus( pic2_p, bus1_b, "bport1", 0, 0xca000000, 0xca000fff);

//**************************Interrupt Controller Timer Module**************************
const char *pit_path = icmGetVlnvString(0,0,0,"IcpCounterTimer",0,"pse");
icmAttrListP pit_attr = icmNewAttrList();
icmPseP pit_p = icmNewPSE("pit",pit_path,pit_attr,0,0);

icmConnectPSEBus( pit_p, bus1_b, "bport1", 0, 0x13000000, 0x13000fff);

//**************************Integrator Board Controller**************************
const char *icp_path = icmGetVlnvString(0,0,0,"IcpControl",0,"pse");
icmAttrListP icp_attr = icmNewAttrList();
icmPseP icp_p = icmNewPSE("icp",icp_path,icp_attr,0,0);

icmConnectPSEBus( icp_p, bus1_b, "bport1", 0, 0xcb000000, 0xcb00000f);

//**************************Debug Leds and Dip Switch**************************
const char *ld1_path = icmGetVlnvString(0,0,0,"DebugLedAndDipSwitch",0,"pse");
icmAttrListP ld1_attr = icmNewAttrList();
icmPseP ld1_p = icmNewPSE("ld1",ld1_path,ld1_attr,0,0);

icmConnectPSEBus( ld1_p, bus1_b, "bport1", 0, 0x1a000000, 0x1a000fff);

//**************************Keyboard**************************
const char *kb1_path = icmGetVlnvString(0,0,0,"KbPL050",0,"pse");

//Semihosting file
const char *kb1_pe = icmGetVlnvString(0,0,0,"KbPL050",0,"model");
icmAttrListP kb1_attr = icmNewAttrList();
icmAddUns64Attr(kb1_attr, "isMouse", 0);
icmAddUns64Attr(kb1_attr, "grabDisable", 0);
icmPseP kb1_p = icmNewPSE("kb1",kb1_path,kb1_attr,kb1_pe,"modelAttrs");

icmConnectPSEBus( kb1_p, bus1_b, "bport1", 0, 0x18000000, 0x18000fff);

//**************************Mouse**************************
const char *ms1_path = icmGetVlnvString(0,0,0,"KbPL050",0,"pse");

//Semihosting file
const char *ms1_pe = icmGetVlnvString(0,0,0,"KbPL050",0,"model");
icmAttrListP ms1_attr = icmNewAttrList();
icmAddUns64Attr(ms1_attr, "isMouse", 1);
```
Hardware and Software Co-simulation

HZ University of Applied Sciences

icmAddUns64Attr(ms1_attr, "grabDisable", 1);

icmPseP ms1_p = icmNewPSE("ms1", ms1_path, ms1_attr, ms1_pe, "modelAttrs");

icmConnectPSEBus( ms1_p, bus1_b, "bport1", 0, 0x19000000, 0x19000fff);

//**************************Real Time Clock**************************
const char *rtc_path = icmGetVlnvString(0, 0, 0, "RtcPL031", 0, "pse");

icmAttrListP rtc_attr = icmNewAttrList();

icmPseP rtc_p = icmNewPSE("rtc", rtc_path, rtc_attr, 0, 0);

icmConnectPSEBus( rtc_p, bus1_b, "bport1", 0, 0x15000000, 0x15000fff);

//**************************UART 1**************************
const char *uart1_path = icmGetVlnvString(0, 0, 0, "UartPL011", 0, "pse");

//Semihosting file
const char *uart1_pe = icmGetVlnvString(0, 0, 0, "UartPL011", 0, "model");

icmAttrListP uart1_attr = icmNewAttrList();

//Define the UART's input source
if (connectUartPort) {
  icmAddDoubleAttr(uart1_attr, "portnum", uartPort);
  icmAddStringAttr(uart1_attr, "finishOnDisconnect", "on");
} else {
  icmAddStringAttr(uart1_attr, "outfile", "uart1.log");
}

icmAddStringAttr(uart1_attr, "variant", "ARM");

icmPseP uart1_p = icmNewPSE("uart1", uart1_path, uart1_attr, uart1_pe, "modelAttrs");

icmConnectPSEBus( uart1_p, bus1_b, "bport1", 0, 0x16000000, 0x16000fff);

//Set verbose level
if (connectUartPort) {
  icmSetPSEdiagnosticLevel(uart1_p, 1);
}

//**************************OceUart**************************
const char *vlnvRoot = 0;
const char *OceUart = icmGetVlnvString(0, 0, 0, "OceUart", 0, "pse");

// Semihosting file
const char *pseSocketsModel = icmGetVlnvString(0, 0, 0, "OceUart", 0, "model");

icmAttrListP attributeList = icmNewAttrList();
icmAddStringAttr(attributeList, "outfile", "uart.out");
icmAddStringAttr(attributeList, "infile", "input.in");
icmAddUns64Attr(attributeList, "UART_CONTROL_REG", 0x0001);
icmAddUns64Attr(attributeList, "FIFO_INT_LEVEL", 23);
icmAddUns64Attr(attributeList, "UART_INT_EN_REG", 0x0000);
icmPseP uart2_p = icmNewPSE("uart2", OceUart, attributeList, pseSocketsModel, "modelAttrs");
icmConnectPSEBus(uart2_p, bus1_b, "bport1", 0, 0x17000000, 0x17000fff);

// Set verbose level
icmSetPSEdiagnosticLevel(uart2_p, 2);
Hardware and Software Co-simulation

//**************************Multimedia card interface**************************
const char *mmci_path = icmGetVlnvString(0,0,0,"MmcPl181",0,"pse");

icmAttrListP mmci_attr = icmNewAttrList();

icmPseP mmci_p = icmNewPSE("mmci", mmci_path, mmci_attr, 0, 0);

icmConnectPSEBus(mmci_p, bus1_b, "bport1", 0, 0x1c000000, 0x1c000fff);

//**************************LCD**************************
const char *lcd_path = icmGetVlnvString(0,0,0,"LcdPl110",0,"pse");

//Semihosting file
const char *lcd_pe = icmGetVlnvString(0,0,0,"LcdPl110",0,"model");

icmAttrListP lcd_attr = icmNewAttrList();
icmAddUns64Attr(lcd_attr, "scanDelay", 50000);
icmAddDoubleAttr(lcd_attr, "noGraphics", noGraphics);
icmAddDoubleAttr(lcd_attr, "busOffset", 0x80000000);

icmPseP lcd_p = icmNewPSE("lcd", lcd_path, lcd_attr, lcd_pe, "modelAttrs");
icmConnectPSEBus(lcd_p, bus1_b, "bport1", 0, 0xc0000000, 0xc0000fff);
icmConnectPSEBusDynamic(lcd_p, membus_b, "memory", 0);

//**************************Boot peripheral**************************
//Special peripheral that initialises memory, generates boot code at
//the reset vector of the processor. Loads the zImage and the ramdisk
//into memory, patches boot code for jumping to kernel start.
//Initialises processors registers and Linux's command line.

const char *smartLoader_path =
icmGetVlnvString(0,"arm.ovpworld.org",0,"SmartLoaderArmLinux",0,"pse");

//If a program is set as an argument when launching simulation, do not load
//the kernel file.
icmAttrListP smartLoader_attr = icmNewAttrList();
if(programLoad) {
icmAddStringAttr(smartLoader_attr, "disable", "1");} else {
icmAddStringAttr(smartLoader_attr, "initrd", ramDisk);
icmAddStringAttr(smartLoader_attr, "kernel", kernelFile);
}

icmPseP smartLoader_p = icmNewPSE("smartLoader", smartLoader_path, smartLoader_attr, 0, 0);
icmConnectPSEBus(smartLoader_p, bus1_b, "mport", 1, 0x0, 0xffffffff);

//**************************RAM**************************
icmMemoryP ram1_m = icmNewMemory("ram1_m", 0x7, 0x7fffffff);
icmConnectMemoryToBus(membus_b, "sp1", ram1_m, 0x0);

//**************************AMBA dummy**************************
icmMemoryP ambaDummy_m = icmNewMemory("ambaDummy_m", 0x7, 0xffff);
icmConnectMemoryToBus(busi1_b, "sp1", ambaDummy_m, 0x1d000000);

//**************************RAM Bridge 1**************************
icmNewBusBridge(busi1_b, membus_b, "ram1Bridge", "sp", "mp", 0x0, 0x7fffffff, 0x0);

//**************************RAM Bridge 2**************************
icmNewBusBridge(busi1_b, membus_b, "ram2Bridge", "sp1", "mp", 0x0, 0x7fffffff, 0x00000000);
Hardware and Software Co-simulation

//**************************************************Processor's interrupt nets**************************************************
icNetP irq_n = icmNewNet("irq_n");
icConnectProcessorNet( arm1_c, irq_n, "irq", ICM_INPUT);
icConnectPSENet( pic1_p, irq_n, "irq", ICM_OUTPUT);

icNetP fiq_n = icmNewNet("fiq_n");
icConnectProcessorNet( arm1_c, fiq_n, "fiq", ICM_INPUT);
icConnectPSENet( pic1_p, fiq_n, "fiq", ICM_OUTPUT);

//**************************************************Other peripherals interrupt nets**************************************************
icNetP ir1_n = icmNewNet("ir1_n");
icConnectPSENet( pic1_p, ir1_n, "ir1", ICM_INPUT);
icConnectPSENet( uart1_p, ir1_n, "irq", ICM_OUTPUT);

icNetP ir27_n = icmNewNet("ir27_n");
icConnectPSENet( pic1_p, ir27_n, "ir27", ICM_INPUT);
icConnectPSENet( uart2_p, ir27_n, "irq", ICM_OUTPUT);

icNetP ir3_n = icmNewNet("ir3_n");
icConnectPSENet( pic1_p, ir3_n, "ir3", ICM_INPUT);
icConnectPSENet( kb1_p, ir3_n, "irq", ICM_OUTPUT);

icNetP ir4_n = icmNewNet("ir4_n");
icConnectPSENet( pic1_p, ir4_n, "ir4", ICM_INPUT);
icConnectPSENet( ms1_p, ir4_n, "irq", ICM_OUTPUT);

icNetP ir5_n = icmNewNet("ir5_n");
icConnectPSENet( pic1_p, ir5_n, "ir5", ICM_INPUT);
icConnectPSENet( pit_p, ir5_n, "irq0", ICM_OUTPUT);

icNetP ir6_n = icmNewNet("ir6_n");
icConnectPSENet( pic1_p, ir6_n, "ir6", ICM_INPUT);
icConnectPSENet( pit_p, ir6_n, "irq1", ICM_OUTPUT);

icNetP ir7_n = icmNewNet("ir7_n");
icConnectPSENet( pic1_p, ir7_n, "ir7", ICM_INPUT);
icConnectPSENet( pit_p, ir7_n, "irq2", ICM_OUTPUT);

icNetP ir8_n = icmNewNet("ir8_n");
icConnectPSENet( pic1_p, ir8_n, "ir8", ICM_INPUT);
icConnectPSENet( rtc_p, ir8_n, "irq", ICM_OUTPUT);

icNetP ir23_n = icmNewNet("ir23_n");
icConnectPSENet( pic1_p, ir23_n, "ir23", ICM_INPUT);
icConnectPSENet( mmci_p, ir23_n, "irq0", ICM_OUTPUT);

icNetP ir24_n = icmNewNet("ir24_n");
icConnectPSENet( pic1_p, ir24_n, "ir24", ICM_INPUT);
icConnectPSENet( mmci_p, ir24_n, "irq1", ICM_OUTPUT);

//If a program was defined as an argument, populate the memory. If cannot load correctly,
//then quit simulation
if(programLoad) {
    if(icmLoadProcessorMemory(arm1_c, kernelFile, False, True, True)) {
        // All files loaded correctly
    } else {
        icmPrintf("Failed to load %s\n", kernelFile);
        exit(1);
    }
}


int main(int argc, char *argv[]) {
  if((argc<3)) {
    icmPrintf("usage: %s <kernelFile> <ramdisk>\n", argv[0], arguments,
    return -1;
  }
  char *kernelFile = argv[1];
  char *ramDisk = argv[2];
  parseArgs(3, argc, argv);
  if(!icmSetSimulationStopTime(stopafter)){
    icmPrintf("Couldn't set stop time \fn", stopafter);
    return -1;
  }
  //Start initializing platform members
  createPlatform(kernelFile, ramDisk, icmInitAttrs, icmAttrs);
  //Run simulation for a limited number of instructions
  if (stopafter) {
    icmSimulatePlatform();
    icmTerminate();
    return 0;
  }
  //Start simulation
  icmSimulatePlatform();
  //Terminate all threads and do clean up
  return 0;
}
IV. Linux kernel compilation log
Sourcery Codebench Lite Edition for ARM processors was used to cross compile the Linux kernel release 3.2.13. The following environment variables need to be set:

```bash
PATH=${PATH}+<sourcery_install_dir>/bin/
CROSS_COMPILE=arm-none-linux-gnueabi
ARCH=arm
```

Océ’s UART driver was copied to /linux-3.2.13/drivers/tty/serial/hdl_uart.c. The Makefile and Kconfig files under that directory have to be modified to include this driver for compilation and add it to the menu of the configuration utilities, respectively.

```bash
//Makefile
obj-$(CONFIG_HDL_UART)+=hdl_uart.o

//Kconfig
config HDL_UART
    bool "HDL_UART"
    default y
```

An additional line was appended to the serial_core.h file located on /linux-3.2.13/include/linux/serial_core.h for setting a port for the UART.

```
#define PORT_HDL_UART 99
```

The UART resources need to be added to the Integrator platform configuration, which is under /linux-3.2.13/arch/arm/mach_integrator/integrator_cp.c

```c
static struct resource hdl_uart_resources[] = {
    [0] = {
        .start = 0x17000000,
        .end = 0x17000fff,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = IRQ_CP_ETHINT,
        .end = IRQ_CP_ETHINT,
        .flags = IORESOURCE_IRQ,
    },
};
```

```c
static struct platform_device intcp_hdl_uart_device = {
    .name = "fpga_uart",
    .id = -1,
    .dev = {
        .platform_data = NULL,
    },
    .num_resources = 2,
    .resource = hdl_uart_resources,
};
```

```c
static struct platform_device *intcp_devs[] __initdata = {
    &intcp_flash_device,
    &smc91x_device,
    &intcp_hdl_uart_device,
};
```

After that, it is necessary to configure the kernel. The default configuration for the Integrator platform can be written to the .config file with the following issued from the top level:

```bash
make integrator_defconfig
```
The GUI kernel configuration utility is launched, also from the top level, with:

```
make menuconfig
```

The UART’s driver can be enabled at Device drivers → Character devices → Serial drivers → [X] HDL_UART.

After configuring all the desired settings for the platform and exiting the configuration tool, the zImage file can be compiled:

```
make zImage
```

After compilation, the file can be found at /linux-3.2.13/arch/arm/boot/zImage, which can be used when launching the simulation.
V. TLM UART Source code

```cpp
#include <iostream>
#include "OceUart.hpp"

using namespace sc_core;
using namespace sc_dt;
using namespace std;

SC_HAS_PROCESS(OceUart);

int chars_in_fifo = 0; // Gets the number of data in RXFIFO
int fifo_read_pos = 0; // Position of the last data read in RXFIFO
unsigned char rx_fifo[FIFO_SIZE]; // RXFIFO data array
int slot = chars_in_fifo + fifo_read_pos; // Calculate which position of the array to read
double delay = 20; // Time delay for the main peripheral thread
bool current = 0; // Stores the interrupt state
int x = 1;

SC_THREAD(getChars);
initialValues();
socket.register_b_transport(this, &OceUart::b_transport);

OceUart::OceUart(sc_core::sc_module_name name) : sc_module(name) {
    SC_THREAD(getChars);
    initialValues();
    tlm::tlm_analysis_port<int> irq;
}

FUNCTION: unsigned char OceUart::gen_data()

DESCRIPTION: Implement behaviour of the UART's RX port in this part. This example will
simulate the reception of the message "HELLO WORLD" through the RX port. Data generation
or specific patterns can be implemented here; or a semihosted connection with a terminal
module. It is important to remember that a LF character is necessary to end the data
stream.

VARIABLES:
* tlm::tlm_analysis_port<int> irq: interrupt net to be connected to the IC peripheral.

unsigned char OceUart::gen_data()
{
    if(x%2==0)
    {
        x++;
        return 0x0A;
    }
    else
    switch(x)
    {
        case 1: x++; return 0x48;
        case 3: x++; return 0x45;
        case 5: x++; return 0x4c;
        case 7: x++; return 0x4c;
        case 9: x++; return 0x4f;
        case 11: x++; return 0x20;
    }
}
```
```cpp
    case 13: x++; return 0x57;
    case 15: x++; return 0x4f;
    case 17: x++; return 0x4c;
    case 19: x++; return 0x44;
    case 21: x++; return 0x44;
  }
}

COPYRIGHT

FUNCTION: void OceUart::updateInterrupts()
*
* DESCRIPTION: Handles the interrupt logic and interrupt state updates. Will activate or
* clear the bit0 of the interrupt identification register when the number of datas is
* bigger than the threshold.
*
* VARIABLES:
* - bool getInterruptStateRX: stores the value of the next-to-be interrupt level respect to
*   the actual state of the registers.
******************************************************************************************/

void OceUart::updateInterrupts()
{
  bool getInterruptStateRX;
  if(chars_in_fifo<24)
  {
    clr(regs.int_id, INT_IDENTIFICATION);
  }
  else
  {
    cout << "Chars in fifo more than 24, set int id"<<endl;
    set(regs.int_id, INT_IDENTIFICATION);
  }
  getInterruptStateRX = (regs.int_en & 0x02) && (regs.int_id & 0x1);
  //cout << "UPDATE INTERRUPT"<<endl;
  cout << "INT_EN & 0x02 is "<<((regs.int_en & 0x02)<<endl;
  cout << "INT_ID & 0x1 is "<<(regs.int_id & 0x1)<<endl;*/
  if(getInterruptStateRX != current)
  {
    current = getInterruptStateRX;
    if(current)
    {
      cout<<"Interrupt change from 0 to 1"<<endl;
      irq.write(1);
    }
    else
    {
      cout<<"Interrupt change from 1 to 0"<<endl;
      irq.write(0);
    }
  }
}

COPYRIGHT

FUNCTION: void OceUart::initialValues()
*
* DESCRIPTION: Sets the default value for each register. This function is called during
* the peripheral's construction.
******************************************************************************************/

void OceUart::initialValues()
{
    regs.type = UART_TYPE;
    regs.version = UART_VERSION;
    regs.status = UART_STATUS;
    regs.control = UART_CONTROL;
    regs.tx_data = UART_TX_DATA;
    regs.rx_data = UART_RX_DATA;
    regs.int_en = UART_INT_EN;
    regs.int_id = UART_INT_ID;
    regs.bd1 = UART_BAUD_DIVISIOR_L;
    regs.bd2 = UART_BAUD_DIVISIOR_H;
    regs.int_level = UART_INT_LEVEL;
}
/* FUNCTION: void OceUart::getChars() */
/* DESCRIPTION: Peripheral's main thread. If the restrictions are met, it will read data */
/* from the RX source represented by gen_data(). Updates the status register accordingly */
/* to the number of characters available in the RXFIFO. Activates bit0 on the interrupt */
/* identification register if the number of datas in RXFIFO is bigger than the threshold */
/* value. */

void OceUart::getChars()
{
    while(1)
    {
        wait(delay, sc_core::SC_US);
        if((chars_in_fifo < FIFO_SIZE) && !(regs.status & RXFO))
        {
            slot = fifo_read_pos + chars_in_fifo;
            if(slot >>=32)
                slot = 0;

            //Read data from RX source and store in the RXFIFO
            rx_fifo[slot] = gen_data();
            cout << "FIFO[" << slot << "] received a data " << rx_fifo[slot] << " at " << sc_time_stamp() << endl;
            chars_in_fifo++;

            //Update the status register
            if(chars_in_fifo>0)
                clr(regs.status, RXFE);
            else
                set(regs.status, 0x1);

            //Update the interrupt identification register
            if(chars_in_fifo >= 24)
            {
                cout << "More than 24, set int id" <<endl;
                set(regs.int_id, INT_IDENTIFICATION);
            }
            else
                clr(regs.int_id, INT_IDENTIFICATION);
        }
        /*An unusual situation (i.e. overflow) can be implemented at this */
        /*point*/
        updateInterrupts();
    }
}

/*/ FUNCTION: void OceUart::trans_info(tlm::tlm_command cmd, sc_dt::uint64 adr, unsigned char* ptr, unsigned int len, unsigned char* byt) */
/*/ DESCRIPTION: Print the information of the generic payload that was received from the bus. */

void OceUart::trans_info(tlm::tlm_command cmd, sc_dt::uint64 adr, unsigned char* ptr, unsigned int len, unsigned char* byt)
{
    if(cmd==tlm::TLM_READ_COMMAND)
        cout<<endl<<"==READ TRANSACTION=="<<endl;
    else
        cout<<endl<<"==WRITE TRANSACTION=="<<endl;

        cout<<"Address is: "<<adr<<endl;
        cout<<"Address hex: "<<hex<<adr<<endl;
        cout <<"Data ptr is: " << hex << (unsigned)ptr <<endl;
        cout <<"Data value is: " << hex << (unsigned)ptr <<endl;
        cout<<"Length is: "<<len<<endl;
}
FUNCTION: void OceUart::b_transport(tlm::tlm_generic_payload &trans, sc_core::sc_time &delay)

DESCRIPTION: Implements the blocking transport method. Depending if the command is a READ/WRITE transaction, it will read/write values from/to the corresponding registers and update interrupts when needed.

VARIABLES:
* -tlm::tlm_command cmd: Stores the type of transaction (TLM_READ/ WRITE_COMMAND)
* -sc_dt::uint64 adr: Stores the memory mapped address related to the transaction.
* -unsigned char* ptr: A pointer to the data in the address above.
* -unsigned int len: Width of the transaction in bytes.
* -unsigned char* byt: Byte enable, not used. Not compatible with OVP.

void OceUart::b_transport(tlm::tlm_generic_payload &trans, sc_core::sc_time &delay)
{
    irq.write(0);

    //Retrieve data from the generic payload
    tlm::tlm_command cmd = trans.get_command();
    sc_dt::uint64    adr = trans.get_address();
    unsigned char*   ptr = trans.get_data_ptr();
    unsigned int len = trans.get_data_length();
    unsigned char*   byt = trans.get_byte_enable_ptr();
    trans_info(cmd, adr, ptr, len, byt);

    if(cmd == tlm::TLM_READ_COMMAND)
    {
        switch(adr)
        {
        case UART_TYPE_ADDRESS:
            *ptr = regs.type;
            cout << "Access UART_TYPE" << endl;
            break;
        case UART_VERSION_ADDRESS:
            *ptr = regs.version;
            cout << "Access UART_VERSION" << endl;
            break;
        case UART_STATUS_REG_ADDRESS:
            *ptr = regs.status;
            cout << "Access UART_STATUS_REG" << endl;
            break;
        case UART_CONTROL_REG_ADDRESS:
            *ptr = regs.control;
            cout << "Access UART_CONTROL_REG" << endl;
            cout << "Read hex value from reg is:
            "<<hex<<(unsigned)*ptr<<endl;;
            break;
        case UART_RXDATA_REG_ADDRESS:
            cout << "Access RXDATA_REG" << endl;
            *ptr = rx_fifo[fifo_read_pos];
            if(chars_in_fifo>0)
            {
                cout << "The data in FIFO[" << fifo_read_pos << "] is transmitted" << endl;
                cout << "Pointer data is:"<< *ptr << endl;
                chars_in_fifo--;
                if(++fifo_read_pos==32)
                    fifo_read_pos=0;
            }
            if(chars_in_fifo == 0)
                set(regs.status, RXFE);
            updateInterrupts();
            break;
        case UART_INT_EN_REG_ADDRESS:
            *ptr = regs.int_en;
            cout << "Access UART_INT_EN_REG" << endl;
            break;
        
    }}
```cpp
    case UART_INT_ID_REG_ADDRESS:
        cout << "Read hex value from reg is: 
" <<hex<<(unsigned)*ptr<<endl;
        clr(regs.int_id, INT_IDENTIFICATION);
        updateInterrupts();
        *ptr = regs.int_id;
        cout << "Access UART_INT_REG" << endl;
        break;
    case UART_BAUD_DIVISIOR_L_REG_ADDRESS:
        *ptr = regs.bdl;
        cout << "Access UART_BAUD_DIVISIOR_L_REG" << endl;
        break;
    case UART_BAUD_DIVISIOR_H_REG_ADDRESS:
        *ptr = regs.bdh;
        cout << "Access UART_BAUD_DIVISIOR_H_REG" << endl;
        break;
    case UART_INT_LEVEL_ADDRESS:
        *ptr = regs.int_level;
        cout << "Access UART_INT_LEVEL" << endl;
        break;
    }
    //This command is part of the TLM standard and must be set
    //after a transaction indicating there was no error in the processing
    trans.set_response_status(tlm::TLM_OK_RESPONSE);
```
```c++
void OceUart::set(unsigned char &reg, unsigned char flag)
{
    cout << "Writing: " << (reg | flag) << endl;
    reg |= flag;
}

void OceUart::clr(unsigned char &reg, unsigned char flag)
{
    reg &= ~flag;
}

bool OceUart::isSet(unsigned char reg, unsigned char flag)
{
    return flag == (reg & flag);
}

bool OceUart::isClr(unsigned char reg, unsigned char flag)
{
    return flag != (reg & flag);
}
```
ifndef OCEUART_HPP
#define OCEUART_HPP
#define FIFO_SIZE 32

#include "systemc"
#include "tlm.h"
#include "tlm_utils/simple_target_socket.h"

class OceUart : public sc_core::sc_module
{
    public:
        OceUart(sc_core::sc_module_name name);

        //TLM-2 simple target convenience socket for the receive FIFO sending data to bus
        tlm_utils::simple_target_socket<OceUart> socket;

        //A byte wide SystemC buffer for input to the Uart
        sc_core::sc_buffer<unsigned char> rx;

        //TLM-2 external output port provided for the interrupt
        tlm::tlm_analysis_port<int> irq;

    protected:
        //SystemC thread, handle data reading
        void getChars();

        unsigned char gen_data();
        void updateInterrupts();

        //Transaction information
        void trans_info(tlm::tlm_command cmd, sc_dt::uint64 adr, unsigned char* ptr,
                        unsigned int len, unsigned char* byt);

        //Register TLM-2 socket to a blocking transport callback function
        virtual void b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& delay);

        //Struct regs to hold the value of each register
        struct
        {
            unsigned char type;
            unsigned char version;
            unsigned char status;
            unsigned char control;
            unsigned char tx_data;
            unsigned char rx_data;
            unsigned char int_en;
            unsigned char int_id;
            unsigned char bdl;
            unsigned char bdh;
            unsigned char int_level;
        } regs;

        //Flag handling utilities
        void set( unsigned char &reg, unsigned char flag);
        void clr( unsigned char &reg, unsigned char flag);
        bool isSet( unsigned char reg, unsigned char flag);
        bool isClr( unsigned char reg, unsigned char flag);
    }
#endif
#ifndef _REG_OCEUART_H
#define _REG_OCEUART_H

#define BPORT1 0x00000000

/**
 * Defines for the relative addresses
 */

#define UART_TYPE_ADDRESS BPORT1 + 0x00
#define UART_TYPE 0x12
#define UART_VERSION_ADDRESS BPORT1 + 0x02
#define UART_VERSION 0x02
#define UART_STATUS_REG_ADDRESS BPORT1 + 0x04
#define UART_STATUS 0x0D
#define RXFE 0x01
#define RXFO 0x02
#define TXFE 0x04
#define TXFF 0x08
#define PARITY_ERROR 0x10
#define UART_CONTROL_REG_ADDRESS BPORT1 + 0x06
#define UART_CONTROL 0x00
#define PARITY_ENABLE 0x01
#define P_EVEN 0x02
#define UART_TXDATA_REG_ADDRESS BPORT1 + 0x08
#define UART_RXDATA_REG_ADDRESS BPORT1 + 0x0A
#define UART_INT_EN_REG_ADDRESS BPORT1 + 0x0C
#define UART_INT_EN 0x00
#define INT_ENABLE 0x01
#define UART_INT_ID_REG_ADDRESS BPORT1 + 0x0E
#define UART_INT_ID 0x00
#define INT_IDENTIFICATION 0x01
#define UART_BAUD_DIVISIOR_L_REG_ADDRESS BPORT1 + 0x10
#define UART_BAUD_DIVISIOR_L 0x00
#define UART_BAUD_DIVISIOR_H_REG_ADDRESS BPORT1 + 0x12
#define UART_BAUD_DIVISIOR_H 0x00
#define UART_INT_LEVEL_ADDRESS BPORT1 + 0x14
#define UART_INT_LEVEL 0x18
#define UART_TX_DATA 0x00
#define UART_RX_DATA 0x00

#endif
VI. TLM Platform source code

---------------------------------------------------------------------
* reg_occeuart.h – Macro file
* Written by Manuel Munoz, Yu Cheng
* Based on the OVP Demo ARM IntegratorCP TLM example
* For Oce Technologies B.V. as part of
* graduation porject: HW/SW Cosimulation
* May, 2012

stantiate <iostream>
#include "tlm.h"
#include "ovpworld.org/modelSupport/tlmPlatform/1.0/tlmPlatform.hpp"
#include "ovpworld.org/modelSupport/tlmDecoder/1.0/tlmPlatform.hpp"
#include "ovpworld.org/memory/ram/1.0/tlmMemory.hpp"
#include "arm.ovpworld.org/peripheral/IntICP/1.0/tlmPlatform.hpp"
#include "arm.ovpworld.org/peripheral/IcpCounterTimer/1.0/tlmPlatform.hpp"
#include "arm.ovpworld.org/peripheral/IcpControl/1.0/tlmPlatform.hpp"
#include "arm.ovpworld.org/peripheral/DebugLedAndDipSwitch/1.0/tlmPlatform.hpp"
#include "arm.ovpworld.org/peripheral/OceUart/1.0/tlmPlatform.hpp"
#include "arm.ovpworld.org/peripheral/MmciPL181/1.0/tlmPlatform.hpp"
#include "arm.ovpworld.org/peripheral/SmartLoaderArmLinux/1.0/tlmPlatform.hpp"

typedef enum bootConfigE { BCONF_UBOOT, BCONF_LINUX, BCONF_BAREMETAL } bootConfig;

class ArmIntegratorCP : public sc_core::sc_module {

public:
    ArmIntegratorCP (sc_core::sc_module_name name, bootConfig bconf, bool connect, int port, const char * variant);

    icmTLMPlatform decoder
    ram<3,15> bus1;
    ram ram1;
    ram ambaDummy;
    arm arm1;
    CoreModule9x6 cm;
    IntICP pic1;
    IntICP pic2;
    IcpCounterTimer pit;
    IcpControl icp;
    DebugLedAndDipSwitch ldi;
    KbpL050 kbl;
    KbpL050 msl;
    RtcPL031 rtc;
    UartPL011 uart1;
    OceUart uartinstance;
    MmciPL181 mmci;
    SmartLoaderArmLinux smartLoader;
    ram lCdDummy;

    icmAttrListObject *attrsForarm1(const char * variant) {
        icmAttrListObject *userAttrs = new icmAttrListObject;
        userAttrs->addAttr("showHiddenRegs", "0");
        userAttrs->addAttr("compatibility", "ISA");
        userAttrs->addAttr("variant", variant);
    }
Hardware and Software Co-simulation

userAttrs->addAttr("endian", "little");
userAttrs->addAttr("mips", 200);
userAttrs->addAttr("override_debugMask", 0);
return userAttrs;
}

icmAttrListObject *attrsForkb1() {
icmAttrListObject *userAttrs = new icmAttrListObject;
userAttrs->addAttr("isMouse", 0);
userAttrs->addAttr("grabDisable", 0);
return userAttrs;
}

icmAttrListObject *attrsForms1() {
icmAttrListObject *userAttrs = new icmAttrListObject;
userAttrs->addAttr("isMouse", 1);
userAttrs->addAttr("grabDisable", 1);
return userAttrs;
}

icmAttrListObject *attrsForuart1(bool connect, int port) {
icmAttrListObject *userAttrs = new icmAttrListObject;
if (connect) {
    userAttrs->addAttr("portnum", port);
    userAttrs->addAttr("finishOnDisconnect", "on");
}
userAttrs->addAttr("outfile", "uart1.log");
userAttrs->addAttr("variant", "ARM");
return userAttrs;
}

icmAttrListObject *attrsForsmartLoader(bootConfig bc) {
icmAttrListObject *userAttrs = new icmAttrListObject;
// Local Image
userAttrs->addAttr("initrd", "fs.img");
userAttrs->addAttr("kernel", "zImage");
userAttrs->addAttr("command", "console=ttymA0,38400n8");
if (bc != BCONF_LINUX) {
    userAttrs->addAttr("disable", "True");
}
return userAttrs;
}

ArmIntegratorCP::ArmIntegratorCP (sc_core::sc_module_name name, bootConfig bconf, bool connect, int port, const char *variant)
: sc_core::sc_module (name), Platform ("icm", ICM_VERBOSE | ICM_STOP_ON_CTRLC | ICM_ENABLE_IMPERAS_INTERCEPTS | ICM_WALLCLOCK),
  bus1("bus1"),
  ram1 ("ram1", "sp1", 0x80000000),
 ambaDummy ("ambaDummy", "sp1", 0x1000),
  arm1 ("arm1", 0, ICM_ATTR_SIMEX | ICM_ATTR_TRACE_ICOUNT, attrsForarm1(variant)),
  cm ("cm"),
  pic1 ("pic1"),
  pic2 ("pic2"),
  pit ("pit"),
  icp ("icp"),
  ld1 ("ld1")
Hardware and Software Co-simulation

// bus1 masters
arm1.INSTRUCTION.socket(bus1.target_socket[0]);
arm1.DATA.socket(bus1.target_socket[1]);
smartLoader.mport.socket(bus1.target_socket[2]);

// bus1 slaves
bus1.initiator_socket[0](cm.bport1.socket); // Peripheral
bus1.setDecode(0, 0x10000000, 0x10000000);
bus1.initiator_socket[1](pic1.bport1.socket); // Peripheral
bus1.setDecode(1, 0x14000000, 0x14000000);
bus1.initiator_socket[2](pic2.bport1.socket); // Peripheral
bus1.setDecode(2, 0x1c000000, 0x1c000000);
bus1.initiator_socket[3](pit.bport1.socket); // Peripheral
bus1.setDecode(3, 0x13000000, 0x13000000);
bus1.initiator_socket[4](icp.bport1.socket); // Peripheral
bus1.setDecode(4, 0x1d000000, 0x1d000000);
bus1.initiator_socket[5](ld1.bport1.socket); // Peripheral
bus1.setDecode(5, 0x1a000000, 0x1a000000);
bus1.initiator_socket[6](kb1.bport1.socket); // Peripheral
bus1.setDecode(6, 0x19000000, 0x19000000);
bus1.initiator_socket[7](ms1.bport1.socket); // Peripheral
bus1.setDecode(7, 0x18000000, 0x18000000);
bus1.initiator_socket[8](rtc.bport1.socket); // Peripheral
bus1.setDecode(8, 0x17000000, 0x17000000);
bus1.initiator_socket[9](uart1.bport1.socket); // Peripheral
bus1.setDecode(9, 0x16000000, 0x16000000);
bus1.initiator_socket[10](uartinstance.socket); // Peripheral
bus1.setDecode(10, 0x15000000, 0x15000000);
bus1.initiator_socket[11](mmci.bport1.socket); // Peripheral
bus1.setDecode(11, 0x14000000, 0x14000000);
bus1.initiator_socket[12](ram1.sp1); // Memory
bus1.setDecode(12, 0x13000000, 0x13000000);
bus1.initiator_socket[13](ambaDummy.sp1); // Memory
bus1.setDecode(13, 0x12000000, 0x12000000);
bus1.initiator_socket[14](lcdDummy.sp1); // Memory
bus1.setDecode(14, 0x11000000, 0x11000000);

// Net connections
pic1.irq(arm1.irq);
pic1.fiq(arm1.fiq);
uart1.irq(pic1.irq);
uartinstance.irq(pic1.irq);
kb1.irq(pic1.irq);
ms1.irq(pic1.irq);
pit.irq0(pic1.ir5);
pit.irq1(pic1.ir6);
pit.irq2(pic1.ir7);
rtc.irq(pic1.ir8);
mmci.irq0(pic1.ir23);
mmci.irq1(pic1.ir24);
}

int sc_main (int argc, char *argv[]) {

    sc_report_handler::set_actions("/IEEE_Std_1666/deprecated", SC_DO NOTHING);

    sc_set_time_resolution(1,SC_NS);

    bootConfig bc = BCONF_LINUX;

    bool timeoutSet = false;
    bool connect = true;
    int port = 9999;
    const char* variant = "ARM926EJ-S";
    sc_time stop(0,SC_MS);

    int i;
    for (i=1; i < argc; i++) {
        if (strcmp(argv[i], "t") == 0) {
            stop = sc_time(atoi(argv[++i]), SC_MS);
            timeoutSet = true;
        } else if (strcmp(argv[i], "n") == 0) {
            connect = false;
        } else if (strcmp(argv[i], "p") == 0) {
            port = atoi(argv[++i]);
        } else if (strcmp(argv[i], "v") == 0) {
            variant = argv[++i];
        } else {
            cout << "Usage: " << argv[0] << " [n] [t <time>] [v <variant>]" << endl;
            cout << " t = timeout for simulation in milliseconds" << endl;
            cout << " n = no interactive connection to uart (logging only)" << endl;
            cout << " p = set port number for uart connect (default 9999)" << endl;
            cout << " v = set variant for the arm processor (default ARM926EJ-S)" << endl;
            exit(0);
        }
    }

    // Ignore some of the Warning messages
    icmIgnoreMessage("ICM_NPF");

    cout << "Constructing." << endl;
    ArmIntegratorCP top("top", bc, connect, port, variant); // instantiate top module

    top.arm1.setIPS(200000000);
    top.cm.setDiagnosticLevel(3);
    top.uart1.setDiagnosticLevel(1);
    top.smartloader.setDiagnosticLevel(7);

    cout << "Starting sc_main." << endl;
    cout << "default time resolution = " << sc_get_time_resolution() << endl;

    if (timeoutSet) {
        sc_core::sc_start(stop);                   // start the simulation
    } else {
        sc_core::sc_start();                      // start the simulation
    }

    cout << "Finished sc_main." << endl;
    return 0;                                  // return okay status
}
References


Aldec, Inc. (n.a.) Virtual Modeling with Aldec and Imperas. Henderson, USA.


