Virtual Cycle-accurate Hardware and Software Co-simulation Platform for Cellular IoT

Marcel Tovar
`elt11mto@student.lth.se`
Patrik Elfborg
`dic11pel@student.lth.se`

Department of Electrical and Information Technology
Lund University

Supervisor: Liang Liu

Examiner: Erik Larsson

October 4, 2017

Abstract

Modern embedded development flows often depend on FPGA board usage for pre-ASIC system verification. The purpose of this project is to instead explore the usage of Electronic System Level (ESL) hardware-software co-simulation through the usage of ARM SoC Designer tool to create a virtual prototype of a cellular IoT modem and thereafter compare the benefits of including such a methodology into the early development cycle. The virtual system is completely developed and executed on a host computer, without the requirement of additional hardware. The virtual prototype hardware is based on C++ ARM verified cycle-accurate models generated from RTL hardware descriptions, High-level synthesis (HLS) pre-synthesis SystemC HW accelerator models and behavioural models which implement the ARM Cycle-accurate Simulation Interface (CASI). The micro-controller of the virtual system which is based on an ARM Cortex-M processor, is capable of executing instructions from a memory module.

This report documents the virtual prototype implementation and compares both the software performance and cycle-accuracy of various virtual micro-controller configurations to a commercial reference development board. By altering factors such as memory latencies and bus interconnect subsystem arbitration in co-simulations, the software cycle-count performance of the development board was shown possible to reproduce within a 5% error margin, at the cost of approximately 266 times slower execution speed. Furthermore, the validity of two HLS pre-synthesis hardware models is investigated and proven to be functionally accurate within three clock cycles of individual block latency compared to post-synthesis FPGA synthesized implementations.

The final virtual prototype system consisted of the micro-controller and two cellular IoT hardware accelerators. The system runs a FreeRTOS 9.0.0 port, executing a multi-threaded program at an average clock cycle simulation frequency of 10.6 kHz.

Popular Science Summary

Designing and simulating embedded computer systems virtually

Cellular internet of things (IoT) is a new technology that will enable the interconnection of everything: from street lights and parking meters to your gas or water meter at home, wireless cellular networks will allow information to be shared between devices. However, in order for these systems to provide any useful data, they need to include a computer chip with a system to manage the communication itself, enabling the connection to a cellular network and the actual transmission and reception of data. Such a chip is called an embedded chip or system.

Traditionally, the design and verification of digital embedded systems, that is to say a system which has both hardware and software components, had to be done in two steps. The first step consists of designing all the hardware, testing it, integrating it and producing it physically on silicon in order to verify the intended functionality of all the components. The second step thus consists of taking the hardware that has been developed and designing the software: a program which will have to execute in complete compliance to the hardware that has been previously developed. This poses two main issues: the software engineers cannot begin their work properly until the hardware is finished, which makes the process very long, and the fact that the hardware has been printed on silicon greatly restricts the possibility of doing changes to accommodate late system requirement alterations; which is quite likely for a tailor-made application specific system such as a cellular IoT chip.

A currently widespread technology used to mitigate the previously mentioned negative aspects of embedded design, is the employment of field-programmable gate array (FPGA) development boards which often contain a micro-controller (with a processor and some memories), and a gate array connected to it. The FPGA part consists of a lattice of digital logic gates which can be programmed to interconnect and represent the functionality of the hardware being designed. The processor can thus execute software instructions placed on the memories and the hardware being developed can be programmed into the gate array in order to integrate and verify a full hardware and software system. Nevertheless, this boards are expensive and limit the design to the hardware components available commercially in the different off-the-shelf models, e.g. a specific processor which might not be

the desired one.

Now imagine there is a way to design hardware components such as processors in the traditional way, however once the hardware has been implemented it can be integrated together with software without the need of printing a physical silicon chip specifically for this purpose. That would be extremely convenient and would save lots of time, would it not? Fortunately, this is already possible due to Electronic System Level (ESL) design, which is compilation of techniques that allow to design, simulate and partially verify a digital chip, all within any normal laptop or desktop computer. Moreover, some ESL tools such as the one investigated in this project, allow you to even simulate a program code written specifically for this hardware; this is known as virtual hardware software co-simulation.

The reliability of simulation must however be considered when compared to a traditional two-step methodology or FPGA board usage to verify a full system. This is because a virtual hardware simulation can have several degrees of accuracy, depending on the specificity of component models that make up the virtual prototype of the digital system. Therefore, in order to use co-simulation techniques with a high degree of confidence for verification, the highest accuracy degree should be employed if possible to guarantee that what is being simulated will match the reality of a silicon implementation. The clock cycle-accurate level is one of the highest accuracy system simulation methods available, and it consists of representing the digital states of all hardware components such as signals and registers, in a cycle-by-cycle manner.

By using the ARM SoC Designer ESL tool, we have co-designed and co-simulated several microcontrollers on a detailed, cycle-accurate level and confirmed its behaviour by comparing it to a physical reference target development board. Finally, a more complex virtual prototype of a cellular IoT system was also simulated, including a microcontroller running a a real-time operating system (RTOS), hardware accelerators and serial data interfacing. Parts of this virtual prototype where compared to an FPGA board to evaluate the pros and cons of incorporating virtual system simulation into the development cycle and to what extent can ESL methods substitute traditional verification techniques. The ease of interchanging hardware, simplicity of development, simulation speed and the level of debug capabilities available when developing in a virtual environment are some of the aspects of ARM SoC Designer discussed in this thesis. A more in depth description of the methodology and results can be found in the report titled *Virtual Cycle-accurate Hardware and Software Co-simulation Platform for Cellular IoT*.

Acknowledgements

We would like to thank first and foremost Magnus Midholt and Michal Stala at ARM Sweden for giving us the opportunity to carry out this project in a new and exciting field for us, and for their continuous support and encouragement. We would also like to thank everyone else at the ARM LPWAN team for aiding us at several occasions as we build up an understanding of the system.

Furthermore, we would also like to thank our supervisor Liang Liu for his guidance in determining a clear goal for the thesis as well as his help in condensing all of the broad work we have done into a coherent project.

Table of Contents

List of Figures

List of Tables

Glossary

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
API	Application Programming Interface
ARM	Advanced Risk Machine (ARM) Ltd
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
CADI	Cycle-accurate Debug Interface
CASI	Cycle-accurate Simulation Interface
DMAC	Direct Memory Access Controller
DSP	Digital Signal Processing
ELF	Executable Linkable Format
ESL	Electronic System Level
FPGA	Field-programmable Gate Array
HLS	High-level Synthesis
HDL	Hardware Description language
ILA	Integrated Logic Analyzer
IP	Intellectual Property
RTL	Register Transfer Level
RTOS	Real-time Operating System
SoC	System on Chip
SystemC	C++ HDL library and simulation kernel
UART	Universal Asynchronous Receiver/Transmitter.
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Chapter 1

Introduction

1.1 Background

Embedded design for system-on-chip (SoC) has been traditionally carried out with varying degrees of separation between hardware and software. Since there is an inherent need to synchronize the two, physical target hardware must be available to verify system functionality is as intended. This entails that a decision must be made regarding the target hardware environment, creating a bottleneck in the development process as the physical hardware must be developed in order to begin designing fully verifiable corresponding software. The fixed hardware design is then generally synthesized and taped-out for physical silicon production as an application specific integrated circuit (ASIC). The isolation of the two designs thus severely extends development time as they are not implemented synchronously. Furthermore, such a rigid methodology causes any modifications to the system design requiring chip alterations to become costly in both time and resources, as new hardware must be re-produced before adapting the software.

Consequently, SoC development has been trying to move away from the two step hardware-software design process in order to keep up with the rapidly increasing processing power of modern chips, while reserving ASICs for the later development stages and the final product. A few different techniques have emerged to address this issue, the most widely used in industry being the integration of re-configurable SoCs or Field-Programmable Gate Array (FPGA) into the development process, see figure 1.1. An FPGA is typically encountered as a physical hardware development board, which has specific logic sections that can be configured according to the register transfer level (RTL) design of the hardware module that is being implemented. This enables a certain degree of flexibility in the development of the system through testing, but at the same time restricts the project to only use the hard IP available on the board; as it is often bundled with a fixed processing core, memories and other peripherals. Techniques such as this one are referred to as hardware software co-design, a term which has been defined by J. Teich as:

> *"[T]he process of concurrent and coordinated design of an electronic system comprising hardware as well as software components based on a system description that is implementation independent by the aid of design automation."* [1]

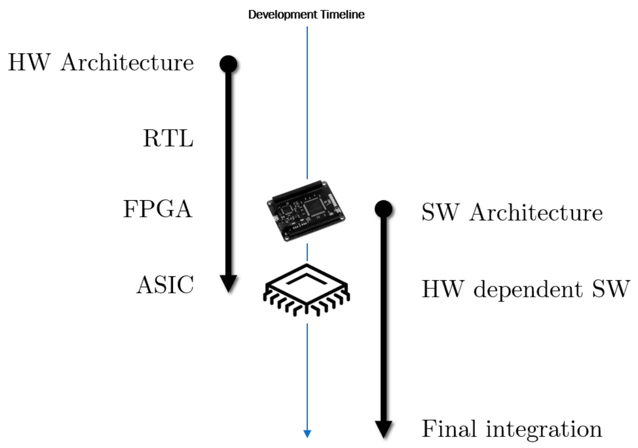

Figure 1.1: Embedded SoC development flow with FPGA.

Another technology that attempts to bridge the gap between hardware and software design and which allows for complete independence from *physical* target hardware during early development is hardware software co-simulation, see figure 1.2. Software with co-simulating capabilities, which is the main focus of this thesis, simulates hardware models according to RTL or higher abstraction level languages and enables a virtual system where different modules can be interconnected. With the addition of a processing core model to such a system, software can thereafter be executed to simulate a complete SoC; all hosted in a standard computer. The usage of hardware-software co-simulation techniques can be used to achieve faster development and reduced costs, compared to FPGA exclusive work-flows by supplying a non-rigid platform which is vastly reconfigurable.

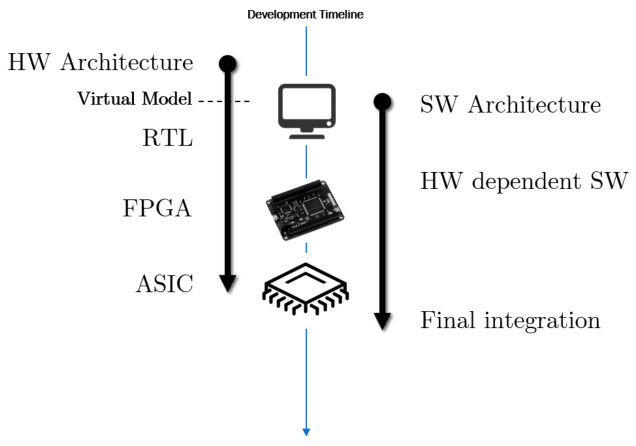

Figure 1.2: Embedded SoC development flow with virtual HW/SW system simulation.

1.2 Cellular IoT and ARM Wireless Business Unit

Cellular Internet of Things (IoT) is the concept of interconnecting physical devices for data transfer through mobile networks. It is a growing technology which is on the verge of a major transition from Global System for Mobile Communications (GSM) to Long-Term Evolution (LTE) communication standards due to infrastructural support changes and performance advantages. LTE potentially provides reduced device cost and power consumption as well as extended coverage at the cost of increased hardware complexity. In the near future there will be billions of devices connected through cellular protocols and there will therefore be a demand for systems with communication capabilities for these.

ARM Wireless Business Unit, is developing embedded solutions based on LTE Narrowband (NB) IoT which offers signal processing software, digital hardware and hardware control software. The systems developed consist of the interfacing between real-time operating system (RTOS) running on processor cores, hardware accelerators and the peripherals required to implement a wireless cellular modem from the physical layer to the transport layer.

1.3 Aims and challenges

The objective of this thesis is the utilization of an existing System-on-chip (SoC) hardware-software co-design and co-simulation tool (SoC Designer) from the microprocessor and semi-conductor company ARM, to realize a completely virtual clock cycle-accurate co-simulation platform for parts of a cellular wireless modem module. In this context the term virtual entails that the whole system (hardware and software) are simulated synchronously on a host machine. Furthermore, a virtual co-simulated environment also entails that the developed solution is completely independent from a physical instance of the target hardware for execution, but is ideally bound to the physical constraints of the actual silicon hardware.

The purpose in the creation of the hardware-software co-simulation platform is to enable the eventual completion of a virtual prototype of an LTE IoT embedded system and accelerate development with early hardware-software integration. The key points of focus are thus putting together a minimal working hardware base, perform module integration and debugging and optimizing system parameters with a cycle-accurate model based on an existing system that is currently being designed with an FPGA work-flow. The challenges of this project are described in detail below:

- To establish a limited but functional cycle-accurate hardware layout with an ARM core and carry out porting of existing software to evaluate functionality and perform benchmarking with focus on the tool itself and the cycle-accuracy of the models. This is to be done with a comparison of a known functioning hardware solution in the form of a development board containing the same core.
- To utilize the reference hardware to check optimization possibilities utilizing the co-simulation platform with regard to performance and resource cost by swapping and altering the base hardware components.

- Expanding the base platform by integrating hardware blocks which make up the back-end of the wireless module, so that they function in the cycle-accurate environment. This is achieved through the wrapping existing high-level synthesis (HLS) C++ hardware blocks and creating/adjusting the interfaces between the modules and the created platform base configuration. A related challenge is to carry out the necessary debugging in order to achieve intended system functionality.

- To perform basic hardware-software time synchronization as a proof of concept for building a complete virtual system prototype, porting an RTOS into the platform is required. The RTOS is used to schedule the access of the hardware components by the software and manages the usage of resources.

- To quantify the functionality of the project, several simulations shall be performed for system validation and verification according to the system requirements at a block level. This entails an analysis of system block response to given inputs with known expected outputs.

- Ultimately, the outcome of this thesis is a virtual cycle-accurate co-simulation prototype for the development of a system that will enable the evaluation of system aspects such as performance and module compatibility at an early development stage. This in turn should enable a co-simulation platform for debugging of both existing and new IP modules in an alternative work-flow to the one currently employed.

1.4 Previous work and alternative tools

The field of hardware-software co-simulation has been developing for the last couple of decades. Early published works include J. A. Rowson's analysis of available techniques with respect to simulation speed, model complexity, debug capabilities and co-verification turn around time [2]. In this paper from the year 1994, cycle-accurate simulation is mentioned among the modelling techniques. It is however also pointed out that system emulation in physical hardware is more widespread than co-simulation, due to the lack of models and complexity of generating them and cost of execution on a host system.

Although there is not a vast amount of publications documenting HW/SW co-simulation implementations, most of them are based on the SystemC standard. One of the most relevant examples of such a methodology, which sets similar goals to this thesis, is outlined by A. Sayinta et al. through a case study of a wireless local area network (WLAN) SoC development [3]. It describes a SystemC based model, which runs concurrently with HDL RTL through a wrapper interface in conjunction with an instruction set simulator. This allows algorithm coherence with HDL implementation, while executing the intended firmware providing system validation. This methodology allows for verification to be done with a re-usable SystemC test-bench for all implementation abstraction levels. Nevertheless, the publication focuses on functionality validation and does not emphasize on simulation cycle-accuracy.

Modern HW/SW co-simulation tools are compatible or based on SystemC to some degree, including SoC Designer. The most prominent industry equivalent tools for SoC interconnect and architecture exploration are Synopsis Platform Architect MCO [4] and Vista from Mentor [5]. Both of these claim to allow for hardware and software validation and performance measurement through TLM models and SystemC, where Platform Architect specifies cycle-accuracy as possible.

1.5 Thesis outline

The thesis project, as documented in this report is comprised of two an introductory section consisting of two chapters on the implications of the technology, three main chapters which comprise the methodology, one result chapter and a final discussion chapter. The process of evaluating the virtual hardware-software co-design/co-simulation platform development process through a virtual prototype implementation for Cellular IoT is divided as follows:

Chapter (2) serves as an ingress into Electronic System Level (ESL) design techniques, FPGA development, simulation abstraction accuracy, SystemC and High Level Synthesis (HLS).

Chapter (3) places ARM SoC Designer within ESL methodologies and gives a functional description of the tool and its relation to SystemC.

Chapter (4) deals with the establishment of a basic functioning co-simulation system by putting together a virtual hardware micro-controller configuration based on a Cortex-M core and other ARM verified IP models. Additionally, it describes the development of a corresponding software bring-up for booting the processor.

Chapter (5) compares the virtual micro-controller system's performance to a closely equivalent system by executing benchmark software functions on a physical silicon hardware reference development and measures the clock cycle discrepancy in the two platforms.

Chapter (6) consists of evaluating HW/SW system co-simulation capabilities with respect to scalability and HW accelerator block development. This is done by measuring latency of two specific blocks instantiated in an expanded version of the initial virtual micro-controller and comparing it to values measured in a corresponding FPGA implementation.

Chapter (7) presents the condensed results obtained in the methodology chapters (4-6).

Chapter (8) culminates the report with an in-depth analysis of the results, an evaluation of the ARM SoC Designer tool with respect to FPGA driven development as well as suggests further work and improvements upon the methodology.

Chapter 2

Electronic System Level Design

The definitions of ESL are diverse, however G. Martin et al. have provided the following condensed description:

> *[T]he utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner, while meeting necessary constraints.* [6]

For the purposes of this thesis, the term is focused to the subsection of this definition which is met by hardware-software co-design and early hardware-software integration methods.

The following chapter aims to describe FPGA driven development while comparing it to ASIC exclusive methodologies and thereafter explore the various ESL simulation techniques to provide context regarding the potential benefits of adopting these methodologies into the development work-flow. The effects of ESL on the development timeline according to J. Teich are illustrated in figure 2.1.

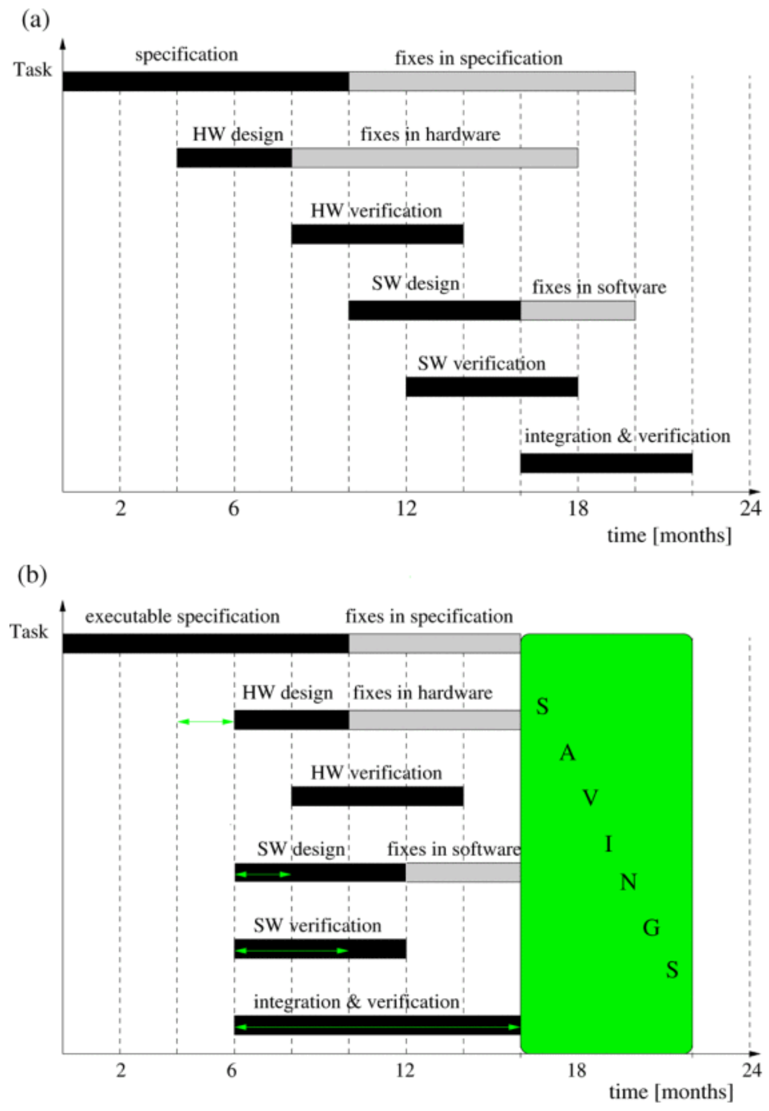

Figure 2.1: (a) Classical design flow and (b) ESL design flow starting from an executable specification and allowing for concurrent development of hardware and software after an initial delay for specification and design space exploration. Savings of up to six months may be expected. At the same time, the risk of late design errors and of overdesigning and underdesigning a system is reduced. [1]

2.1 FPGA

A field-programmable gate array is a semiconductor device which consists of logical gates and registers, which exists in a none-fixed configuration and provides no useful functionality off-the-shelf. Instead, the logic functionality of the gate arrays can be switched into a representation of a user defined hardware description. It is even possible to continuously re-define the hardware layout throughout the different development phases. FPGAs are often standardized boards that enable the verification of a system's hardware blocks by loading RTL models and recreating the intended functionality by realigning the coupling within the configurable logic sections. Nevertheless, FPGA boards usually also include none-modifiable hard or fixed IP, which provides basic and optimized functionality for standard components such as CPUs, memory blocks, serial communication peripherals. Thus, software can be executed on the FPGA given that its design is compliant to the on-board components.

To place the usage of FPGA within the ESL of design, there is a requirement to touch upon the term electronic design automation (EDA). EDA tools are involved with the translation of RTL hardware designs into a logic gate configuration layout, which is something not required in virtual hardware-software co-simulation techniques; where RTL is often the lowest level of abstraction employed.

2.1.1 FPGA vs ASIC

Benefits of using FPGA technology over ASICs during development include reduced non-recurring engineering and shorter time to market [7]. Although FPGAs enable flexibility since they can be reused in different development cases, they require larger silicon area and consume more power due to a higher transistor count, they introduce delays and thus have decreased performance compared to an ASIC chip [8, 9]. Furthermore, the design can never exceed the existing hardware constraints provided by an FPGA. Even though FPGA based co-design reduces the development time, there is still a need to acquire non-final hardware just for co-verification purposes. [10]

2.1.2 High-level synthesis

High-level synthesis (HLS), also known as ESL synthesis is "an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior" [15]. This translates in most scenarios to the production of RTL HDL code based on C++ abstraction level hardware descriptions [16]. As this thesis deals with several hardware blocks developed through HLS with the Catapult HLS tool from Mentor Graphics and its C++ library Catapult C [17], it is important to briefly highlight the design flow and different methods that can be utilized for design testing from an HLS perspective as shown in figure 2.2.

The initial step is generally to develop a block description and perform algorithmic testing at a C++ level to check behavioural and functional correctness of the hardware. Thereafter HLS is performed to obtain an HDL description which can be input into an RTL event simulator with a signal test-bench.

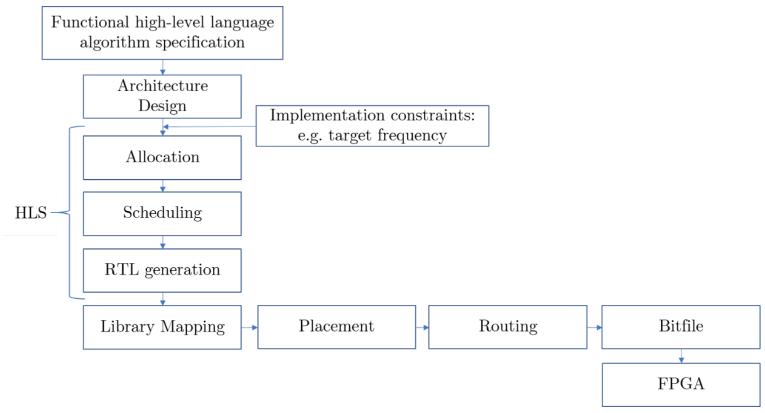
Functional high-level language algorithm specification
Architecture Design
Implementation constraints: e.g. target frequency
Allocation
HLS
Scheduling
RTL generation
Library Mapping
Placement
Routing
Bitfile
FPGA

Figure 2.2: HLS hardware development flow for FPGA.

It is during the HLS process that target constraints and technology library information results in several degrees of optimization and thus often introduce changes to the generated RTL, with respect to the high level design. These generally include the addition of hardware pipelines to ensure hardware timings are met, as well as other factors that may affect the synthesized block latency when compared to the C++ level description.

Finally, the RTL gate and register logic can be synthesized with a library and programmed unto an FPGA to perform integration testing and system verification before moving to the production of an ASIC. With this in mind it is noteworthy that any alterations that may be done intentionally or unintentionally in the automated RTL generation process must be considered when comparing any high level design source to a synthesized hardware block running on target, so as to be able to properly evaluate the fidelity of the synthesized block to the original description of the block, both functionally and structurally.

2.2 System simulation

In order to move towards a completely physical hardware independent hardware-software co-design, virtual co-simulation tools have recently increased in popularity. Such tools allow for software execution in a hardware target environment prior to its availability in a silicon ASIC. Several methods exist to conduct SoC co-designing and co-verification and most of these were originally based on VHDL and `C/C++`. One of the most widespread open license libraries to do system hardware-software co-simulation is the encompassing SystemC C++ expansion library, which is capable of both describing hardware and simulating the execution of it through a kernel [11]. As the ESL field is still in a settling phase, the terminology and taxonomy for the various techniques within it is a controversial subject. Nevertheless, from a SystemC point of view, Grötker et al. define several critical characteristics of a model based simulation in their book System Design with SystemC [12], both from an individual component model and full system perspective. This terminology is one adapted in this dissertation when addressing the subject.

2.2.1 Model characteristics

A simulated system represents a corresponding physical target or a proposed implementation of such and can thus be evaluated in terms of how accurate it is represented as a model according to various principles. Grötker et al. provide a terminology list of the accuracy criteria for a simulated hardware model, when comparing it to the real physical implementation, which is described below. [12]

- ***Structural accuracy*** refers to how well the model reflects the structure of the actual implementation. Regarding hardware, this concerns whether the signals and pins are accurate to the real system.
- ***Timing accuracy*** indicates the degree to which the model mirrors the timing of the target. A physical target introduces delays due to required processing and a model implementation could have delays enforced by design specification.
- ***Functional accuracy*** is a measure of how correct the model behaves in terms of functionality compared to the target. It is common to simplify complex functions and behaviours in order to achieve faster simulation speed with a high-level model.
- ***Data organization accuracy*** refers to the fidelity in which the model stores data structurally with regard to the physical target. For example, hardware register matching.
- ***Communication protocol accuracy*** concerns how fully modeled the communication between modules is, from a protocol perspective. Protocols can be more or less complex and can therefore be modeled on different levels of abstraction with a trade-off in accuracy. The communication protocol accuracy term reflects how well the model is true to the target implementation.

2.2.2 Simulation abstraction models

There are several abstraction levels when it comes to modelling a hardware-software system. These have different areas of focus that involve trade-offs between detail in simulation accuracy and execution time. The following section attempts to organize two different abstraction level descriptions into a coherent taxonomy that incorporates the various terminology that is often used.

Grötker et al. define the different levels, ranging from the highest to lowest level of abstraction, as Executable specification, Untimed functional, Timed functional, Transaction-level, Behavioural modelling, Cycle-accurate and Register-transfer level, where as R.Schaumont has divided the range of abstraction into Transaction-accurate, Instruction-accurate, Cycle-accurate, Discrete-event, Continues time. They both incorporate some of the same functionality, but from different perspectives. Grötker et al. take on the abstraction definition from an implementation perspective and R.Schaumont focuses on defining it based on granularity of time.

Executable specification

An executable specification abstraction level models the desired functionality of a design based purely on the specification, meaning it does not take into account any proposed implementation aspects. [12]

Untimed functional

A model which is implemented on the untimed functional level does not include any representation of delays, even if it is present in the specification. This is what sets it apart from the executable specification level. None of these two are structurally accurate to the modelled component. Communication is modeled point-to-point without any shared objects such as a bus in between. Instead, this is usually done through utilization of FIFOs with blocking write and read to ensure correctness. [12]

Timed functional

Timed functional models add delays to achieve a closer to reality simulation. These delays can for instance be timing constraints based on the specification, or delays introduced by a specific target implementation's processing. The communication between two modules is however, still modeled as point-to-point but it is possible to add delays in regard to this aspect as well. The none true to target structural representation of untimed functional and executable specification abstractions also applies to timed functional models. [12]

Transaction-level

In a transaction-level model (TLM) the focus lies on the interaction between modules on a pure functional, none protocol specific, perspective and allows for exploring of system design as to where and when data should be communicated. The communication is often implemented in a more realistic way than untimed and timed functional abstraction levels by utilizing function calls. This modeling technique allows for accuracy in regards to both functionality and timing, but structural accuracy is not obtained as the pin-level transfers are bundled together as one coherent function call to increase the simulation speed. [12, 13]

Behavioural modelling

This level of abstraction focuses, like the TLM, on the flow of the design with a concern on exploration of relative ordering of input and output events. However, a behavioural model adds pin-accuracy and describes the transactions in more detail making it more accurate in regard to communication protocol accuracy. It would though not be considered completely structural accurate as the internal structure of the module does not reflect the target implementation. [12]

Instruction-accurate

A more detailed overview than TLM is also provided by Instruction-accurate level, which utilizes instruction sets as the frame of reference instead of transactions when sequencing the simulation. This level of abstraction is only applicable when the simulated system contains a microprocessor. [13]

Cycle-accurate

The term cycle-accurate simulation is defined by A.Khan et al. as *a simulation that conforms to the cycle-by-cycle behaviour of the target design. The behaviour may be characterized in terms of the values of all the state elements of a machine (registers, memories, etc.) for every clock cycle.* [14]

This type of simulation takes real time events and places them on the next chronological clock-cycle thus creating functionally accurate results for each cycle. As single-clocked hardware circuits have behaviour defined by the frequency of the clock, simulating at this level enables close to target behaviour at a high granularity of time. The model is also pin-accurate, but its internal representation of components such as the registers and the combinational logic does not need to be true to target. Using this level of abstraction it is possible to analyze real-time performance of a system, which is not available at higher abstraction levels [12, 13, 14].

Since this thesis deals with a time-critical LTE system, the capability to simulate at a cycle-accurate level is essential for ensuring that the complete hardware-software system is capable of synchronizing with the carrier network.

Register-transfer level

RTL simulation adds structural accuracy to cycle-accurate models which makes it functional, timing and structural accurate. A model in a register-transfer level abstraction is defined down to the transactions between functional units and register files. These and interconnections between them are together called data paths. Hidden by the abstraction level are gates and latency of computation. In fact, computations and data transfer are treated as to take zero time, depending on the clock. Meaning that the propagation delay of a critical signal path is not taken into account and there is thus no guarantee that the target implementation will be able to meet the time constrains of the clock period. [12]

2.2.3 System models

Individual component versus system abstraction level

Based on the model characteristics and the simulation abstraction levels described in the previous sections, an explanation from a system point of view is required. In order to model a hardware system, there are several aspects which need to be simulated, these can be simplistically divided into individual component blocks and their communication with other blocks. As a system expands, several abstraction levels are often employed to achieve inter-connectivity, improve simulation speed or for ease of implementation.

From the perspective of a cycle-accurate system, what is lost is structural and data organizational accuracy to a real physical implementation, however this does not necessarily compromise functional and timing accuracy. For example, a transaction level abstraction can be employed for the communication between two cycle-accurately described component models, but as long as any delays are represented the system should retain communication protocol accuracy as well as a cycle-by-cycle and pin-by-pin accurate data value.

2.2.4 Type of models utilized

Since this project does not aim to structurally recreate any specific physical system, but instead seeks to create and evaluate a cycle-accurate virtual prototype, the developed system utilizes a few types of models to achieve cycle-accuracy while keeping the simulation load to a minimum. These models are mainly cycle-accurate models generated from RTL which are described in a cycle-by-cycle fashion, behavioural non-structurally accurate models, and transaction level protocol communication models used between blocks. The different occurrences of these in the system are described in the later sections of this report.

2.3 SystemC

SystemC is a hardware description language (HDL) and a system-level modeling library specified in the IEEE 1666-2011 standard [18]. It is driven by the Accellera Systems Initiative and is available through their website [19]. SystemC consists of various macros and classes implemented as an ANSI C++ class library, created with the purpose of enabling the modeling of hybrid hardware and software systems. It includes a simulation scheduler, or kernel, which is event driven, meaning it can advance the simulation time to the next scheduled event if there is no immediate event pending and thus resulting in a more time effective simulation. The SystemC language structure is displayed in figure 2.3. Its core language consists of the kernel, modules, ports, interfaces and channels.

The modules and ports represent a systems structural information and the interfaces and channels act as an abstraction for the communication. Modules are the system's building blocks, and most often contains ports for communication with other blocks and an implementation of the functionality. Modules can contain other modules, allowing for a hierarchical structure with interconnected sub-blocks that make up a complete functional module. This allows for code recycling, as well as makes it possible to inspect a model with different detail level abstractions as is suitable for a given system simulation. Also included in the library are predefined channels, a set of commonly used entities such as signals, clocks, FIFOs, mutexes and semaphores. Even more, utilities and data types enables debug functionality and modeling of digital logic and fixed-point arithmetic. It is possible to add other C++ libraries and user defined classes into the design of a system given that they comply to the SystemC standard.

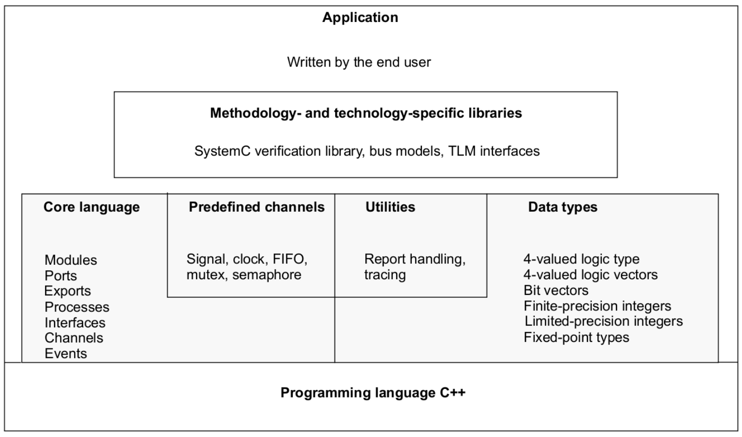

Figure 2.3: The SystemC language architecture. [18]

One such library which extends the usage of SystemC is the OSCI TLM-2.0 language [20], which comes bundled in SystemC since version 2.3.0. It describes and adds the functionality of Loosely timed and Approximately timed coding styles. These coding styles describes different ways of designing models of a simulated system while focusing on the timing accuracy of the communication, hence they work on the TLM abstraction level. Loosely timed are the less accurate but enables a faster simulation speed of the two and allows for temporal decoupling from the system simulation time, which means individual SystemC processes are allowed to run ahead of simulation time until they reach a point where they need to interact with other processes. The idea is to reduce the amount of context switches since every switch adds overhead to the simulation. The communication itself are divided into two timing points, one for the transaction request and one for the response.

In a simulation which utilizes the Approximately timed coding style, the timing is more accurate and each process has to comply with the system simulation time. The communication is allowed to be defined into more than two timing points to allow modeling of multiple phases within one transaction. The delay of a transaction is also expressed within the Approximately timed coding style.

The TLM-2.0 language reference manual also mentions that it would be possible to derive a cycle-accurate modelling coding style using techniques present in the specification, but states that it is currently out of the specification's scope.

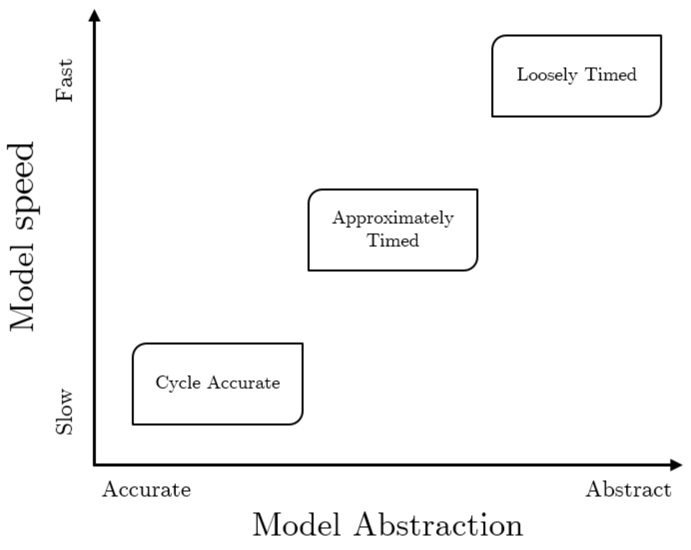

Figure 2.4: OSCI TLM 2.0 coding style characteristics.

Chapter 3

ARM SoC Designer

3.1 Overview

The ESL approach used for this thesis investigation into co-design and co-simulation is using ARM's SoC Designer [21], a SystemC based toolkit that enables the development and simulation of virtual prototypes. The software contains utilities for both set-up and running of pre-configured models, as well as tools that are used for crafting a custom platform based on ARM intellectual property (IP), through either the ARM IP Exchange, included behavioural models bundled with the tool, and/or custom third party models. The IP Exchange is a database that contains a wide array of licensed ARM hardware cycle-accurate models such as processing cores, peripheral components and other SoC components [22].

In the case of the ARM processor models, the SoC Designer environment allows them to execute instructions from an executable binary placed in a memory in the same way as a silicon implementation of the processing core would; as long as the software is compiled compliant to the hardware configuration of the virtual prototype. This is the aspect that allows for the co-simulation aspect of the tool.

The SoC designer software features the possibility to run hardware and software concurrently in different modes, namely loosely timed, and cycle-accurate, depending on the type and capability of the IP models which are included in the simulation [23]. The types of models provided by ARM are divided into cycle-models (cycle-accurate), and fast models (loosely timed), and this can be used simultaneously or interchangeably in a simulation. This means that cycle-accurate simulation can be entered at specific time breakpoints where execution is time-critical, while allowing the simulation to run at higher abstraction levels in sections that are not a debugging priority. A particular advantage to this functionality would be to reduce booting time during for example system start-up, increasing efficiency. Alternatively, both a hybrid cycle-model and fast-model system could be constructed that could run marginally faster than a purely cycle-accurate configuration. The variations of simulation involving Fast Models are however not investigated in this thesis due to the availability of such models for the processing core utilized as well as the real-time requirements of the intended Cellular IoT system prototype.

The cycle-accurate IP provided in the ARM IP Exchange for the tool is compiled utilizing cycle-based modelling for all the components that are available officially for the tool on the ARM IP exchange. These component models are largely generated through the ARM Model Studio software from HDL descriptions of RTL by translating the low-level RTL to a higher level C++ cycle-scheduled representation which retains full functional implementing the ARM ESL modelling interfaces [21, 24]. The ESL interfaces are explained more in detail in the next section of this report.

SoC Designer thus supports any C++ hardware modules that implement their ESL API libraries natively or through wrapping. The tool contains platform debugging tools that will allow for breakpoint setting, register, memory and even hardware signal inspection through the usage of the cycle-accurate simulation, debug and profiling interfaces which are described ahead in this chapter.

3.2 Modelling libraries

Firstly, in order to import any given C++ or SystemC model to SoC Designer, it must be compliant to the cycle-accurate simulation interface (CASI). Secondly, in order to provide visibility and interactivity into the simulated signals, registers and other internal workings of a hardware model, the cycle-accurate debug interface (CADI) must be implemented. These two C++ modelling libraries along with the cycle-accurate profiling interface (CAPI) make up the ESL APIs which are core to SoC Designer.

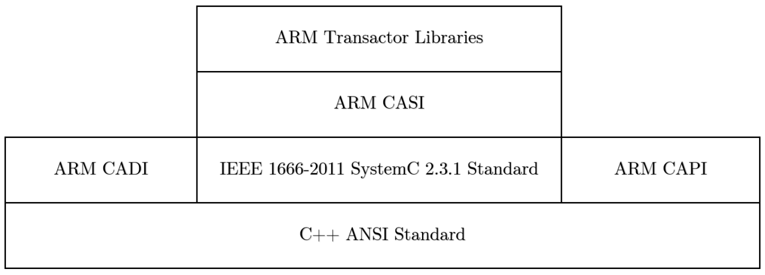

Figure 3.1: Hierarchy of ARM transactors and ESL APIs over SystemC and the C++ ANSI standard.

3.2.1 CASI

The cycle-accurate simulation interface is based on the SystemC library and manages the interconnection and communication between components as well as the advancing of time in the simulation environment. It provides infrastructure for both cycle-based scheduling and transaction level communication modeling and has support for the legacy SystemC event-driven simulation. It is thus possible to simulate both CASI cycle driven components along side event-driven SystemC components since the CASI clock signal generation is based on the SystemC OSCI scheduler.

The communication is modeled directly in a TLM fashion through port-based interconnection and has support for both signal based and transaction based com-

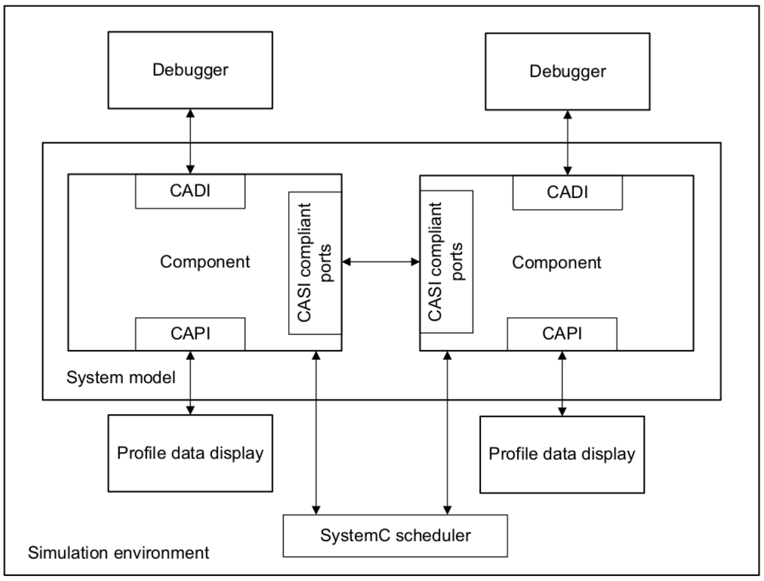

Figure 3.2: The cycle-accurate simulation, debug and profiling interfacing layers [25]

munication, which models different degrees of protocol accuracy. A signal based communication carries one signal while a transaction based communication models several pins as one. Other than the signal and transaction based signals, there also exists generic SystemC signals by which it is possible to interconnect SystemC components. The protocol accuracy in place when using generic SystemC signals would depend on the underlying SystemC port as it is possible to define the signal structure being communicated.

All of these communication alternatives are realized by CASI ports which are divided into master and slave rather than input and output port, where a master must be connected to a slave port of the same type. The different port types as represented in the SoC Designer GUI is illustrated in fig 3.3. It should be however noted to the reader that the CASI TLM implementation is different from the SystemC implementation of SystemC TLM 2.0 mentioned in section 2.3.

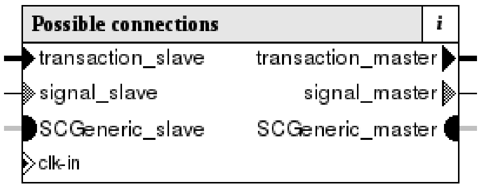

Figure 3.3: Communication port types as displayed in SoC Designer.

The CASI simulation is cycle-based synchronous and each simulation cycle is divided into two phases, a communicate phase and an update phase, which together represents one hardware clock cycle. During the first, communication phase, all inter-component communication should be performed, at which no modification to shared resources are allowed. Instead, the updating of shared resources should be performed in the next, update phase.

In the case of a simulation environment with both CASI components and legacy SystemC components, the clocked components are triggered on the edge of a clock pulse, while the event-driven components are allowed to take place during clock phases.

AMBA Transactors

As an addition to the CASI ports and signal types, SoC designer has support for a proprietary collection of individual libraries that bundle the standard AMBA communication protocols called transactors. These are an ARM proprietary C++ library implementation of CASI transaction based communication that aims to raise the abstraction of the protocols to transaction-level, with a cycle-accurate coding style. In addition to bundling pins together and thereby facilitating connection between components in SoC Designer, the transactor protocol bundle also implements the CADI and CAPI adding debug capabilities. These include functionality of breakpoints to control simulation, transaction views which interprets the pin-level protocol signals into readable AHB definitions and the possibility to capture waveforms. This feature is explained in more detail in section 3.2.2 and 3.2.3. It is also worth noting that the vast majority of the models available from the ARM IP Exchange solely implement the transactor library for all the standard protocol communication.

3.2.2 CADI

The cycle-accurate debug interface enables debug features such as viewing and altering internal registers and memory contents for any type of component. It also has the functionality to set breakpoints on these registers and memory addresses which will trigger when a value is updated. Another feature is the possibility to toggle arbitrary functionality, such as the viewing of the internal registers. This is done by implementing component parameters as variables in the code which can be interactively set through the CADI and thus changed from the GUI within SoC Designer prior to, or during halted simulation, depending on the parameter type. CADI also enables interaction with an external debugger which supports the interface. [25]

3.2.3 CAPI

Apart from the above interfaces, the cycle-accurate profiling interface (CAPI) provides the additional functionality of creating and assigning data streams to specific nodes, enabling the storage of for example signal, register or port values in a CASI module and thereafter providing a historical view of the accumulated data during a simulation. [25]

3.3 The canvas & simulator

The SoC Designer software is divided into two programs, the project builder tool called the canvas where the hardware system is assembled and configured, and the actual simulator which executes the output from the canvas.

All of the shared object models are loaded into the canvas individually into the tool library in order to interconnect a system. All the blocks are placed and optional parameters for the models are set, and basic signal type connectivity assertion is performed. It is worth noting that a shared object compiled from a hardware block model source which implements the CASI individually will be able to be included in isolation, however, to achieve interconnection with other blocks and successful simulation, every relevant port must be connected graphically through a wire to a port of the same type containing the same signal type.

Thereafter the interconnected system is inputted into the simulator where all the run-time interaction functionality occurs as previously described by the ESL APIs.

3.4 Processor core models and ELF files

Any ARM IP Exchange processor models included into an SoC Designer simulation are compatible with executable and linkable format (ELF) files (.elf or .axf extension), which contain the compiled software instructions to be executed. An ELF file consists of the information required to run the program on the target system; including an indication of memory map of code regions as well as initial stack and heap pointers. These files support the inclusion of additional information to enable debug capabilities during software execution.

Chapter 4

Developing a Virtual System Micro-controller

Reaching a base for the system included designing a base configuration of the hardware design, developing system-specific start-up code as well as creating dynamic build automation templates. It is worth mentioning that all the SoC Designer models utilized in this section are either ideal/behavioural configurable hardware components, or fully cycle-accurate models compiled by ARM from RTL code.

4.1 Virtual hardware base configuration

To put together a micro-controller, the minimal hardware requirements are a processing core for instruction execution and memories to store both instructions and data. These need to be connected at pin level for software execution to take place. For this initial section, a specific cycle-accurate processor model has been used as the core for the base design. This processing core, an ARM Cortex-M which will be from this point forward referred to as *Core M*, implements the AHBv2 standard bus connectivity protocol. The core, has three bus master interfaces which are to be connected into the system. The first two, the instruction and data buses are for fetching the processor instructions and data from code respectively. This are expected to point to the same memory, however they are separated into two bus in order to support various hardware designs such as the utilization of a flash memory with instruction data prefetching and bypass for data code, or a memory cache with hit-under-miss capabilities between code and instructions. The third and last bus master, the system interface, controls all of the remaining memory accesses during execution such as the heap, stack and any or all present slave peripherals in the system.

In order to provide support for additional peripheral components in the design, an ARM model of a bus matrix was added. A bus matrix or interconnect, is a block that which is placed between a master block, or controller, such as a processing core and all other peripherals, acting as a multiplier for bus access points and provides arbitration of shared resources between bus masters and slave (controlled) blocks. The presence of such an interconnect also allows for the expansion of the system through the integration of additional hardware blocks. In this case the bus matrix handles the arbitration of the access data, instruction and system buses as well as peripheral access. Bus arbitration consists of determining which master input port of the matrix has access to a specific slave output port. This model employed a fixed arbitration scheme on the buses, which means that one AHB port always has the highest priority and the remaining ones have lower, fixed priorities [26].

Consequently, two behavioural, configurable RAM memory models were connected to the bus matrix, forming a system capable of executing software. One RAM model was simulated as a read-only memory (ROM) used for code and instruction storing and the other RAM as a read/write memory which holds the system stack and heap. Both models were initially configured to have a standard default access times and latencies.

For the purpose of being able to interact with the simulation externally and dynamically under execution, a Universal Asynchronous Receiver/Transmitter (UART) was required in the base configuration. The available UART model in SoC designer implemented however the APB standard, and to the means of make it compatible with the AHBv2 bus matrix, an AHBv2 to APB adapter was placed in between them. The achieved base hardware configuration design, which represents the microcontroller of the embedded system, is illustrated in figure 4.1 and later simplified in figure 5.1.

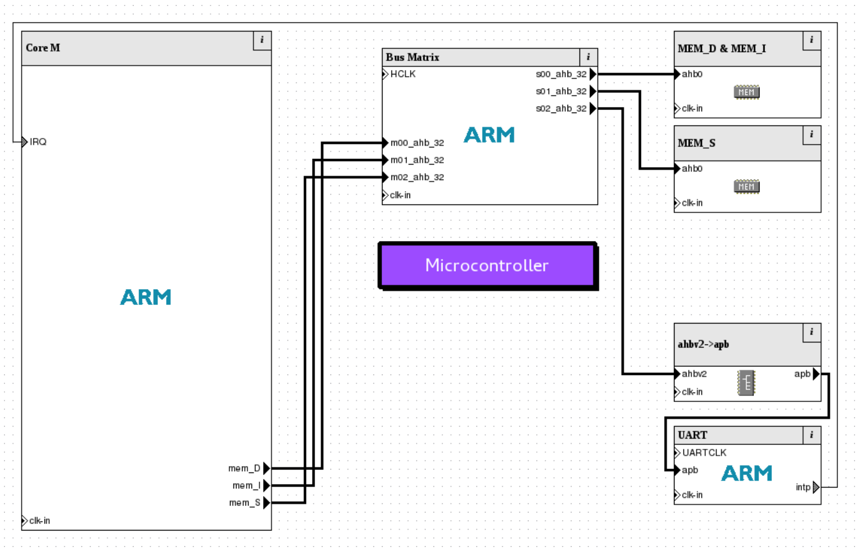

Figure 4.1: The virtual platform micro-controller hardware configuration in SoC Designer.

Regarding the UART model used in the platform, since it is placed in a virtual simulation which cannot be accessed externally at pin level, a TCP (Transport Control Protocol) server functionality included in the model was used as the method for input and output of serial data. When utilizing the this functionality, the RX (receive) and TX (transmit) pins are disabled and rerouted to the TCP server's data in/out. The server is hosted locally on the machine on a user determined port which is set as a component parameter. However, while using this added functionality, the model has a limitation which concerns the FIFO queue usage and the interrupt generation: the input queue is limited to one character entry, and it is therefore not possible to use an interrupt driven input handling since the hardware interrupt is triggered depending on FIFO queue load. The selectable interrupt trigger levels are 1/8, 1/4, 1/2, 3/4, 7/8 of queue capacity and the RX and TX FIFO queues have a fixed size of 16, making it impossible to trigger an interrupt for a load less than two characters.

4.2 Start-up code

The first step towards enabling the execution of arbitrary code on the system was the initialization of present hardware. The hardware components are configured through software drivers which allow for the control of the hardware component's registers. In the drivers, which have been written in the C programming language, a representation of the registers have been defined as structs and these have been associated with the physical address of the component in the AHB memory map. In addition to this, there are also several functions that control the operation of the hardware by accessing the register structs.

In the virtual prototype micro-controller, the components which needed to be configured were the processing core and the UART. Beginning with the core, the system control space (SCS) registers must be specified. These registers include, but is not restricted to, system control block (SCB), the nested vectored interrupt controller (NVIC), system timer (SysTick) and memory protection unit (MPU):

- The system control block provides processor information and the control of the processors features, such as internal interrupts (faults). It also includes a vector table offset register (VTOR) to specify the address of the vector table. The vector table in turn specifies the initialization value for the stack pointer as well as the entry points of all exception handlers [27]. The micro-controller system implements both the internal exception handlers and a handler for the interrupts coming from the UART.

- The NVIC controls the enabling, disabling and clearing of external interrupts.

- The SysTick is a continuous counter which decreases in value until it reaches zero, at which point it resets to a reload value and triggers an interrupt before it starts decreasing again [27]. The SysTick register value is the time granularity unit used for RTOS execution scheduling, this is relevant later in section 6.7.1.

- To protect a system from running code which might alter data at the wrong address, there exists a possibility to configure the MPU regions if it is present in the hardware. The MPU defines access rights according to execution privilege level to certain regions of the memory. If a process tries to read or write at an unauthorized region the execution results in a hardware fault. This functionality mainly prevents erroneous or unintentional access to important memory sections and helps maintain system stability. [27]

Configuration of the UART was done by utilizing an existing driver which implements functionality to initialize, reconfigure and handle data transactions such as receive and transmit. Initialization consists of setting the baud rate, enabling sending interrupts and specifying the threshold at which the UART should trigger an interrupt based on how full the FIFO (first-in-first-out) queues are. Regarding the interrupts, as mentioned previously when using the TCP server functionality in the UART, the receiving FIFO queue can only hold a value at a time. This, together with the fact that it is not possible to set the interrupt signaling threshold to less than two entries in the queue, results in no interrupt generation from the UART while receiving data. Instead, polling was used in the system to receive serial data from the UART to circumvent this issue.

However, as an addition to the driver, the C standard output was redirected to use the UART in order to receive a response from the system in the TCP server whenever `printf()` is invoked. The re-targeting involved re-declaring the `__FILE` and `__stdout` structs and overwriting `fputc()` and `ferror()` to each place the output character to the transmit register of the UART instead of the standard implementation which takes into account which filehandle the output is supposed to be directed to. [28, 29]

4.3 Compiling the start-up binary

Compiling the software for the initial minimal micro-controller system was done using the ARM Compiler 5.06u4 Toolchain [30], the toolchain includes a GCC style compiler, linker and librarian for compiling C and Assembly language files and linking object files and libraries in order to create an ELF image. In order to arrange the placement of the different .elf executable segments, a scatter file which instructs the linker on how to organize the memory layout of the system image is used. The scatter file is useful in many situations for embedded systems and especially when it comes to placement of data and code onto different types of memories. Moreover, it eases the use of memory-mapped peripherals by the same use of placing hardware representational structs at the corresponding address of the peripheral. This allows for the separation of hardware placement and hardware functionality, meaning it is more convenient to keep the hardware layout and software representation in sync. Another functionality of the scatter file is to define the stack and the heap in the memory, both in terms of placement and size. [31]

For instance, the vector table was placed as the first element of the data section in the resulting executable file since the initialization value of the VTOR was not changed from zero.

Chapter 5

Software Performance of a Minimal System

With the aim of testing the cycle-accurate capability of the SoC Designer and the virtual platform through existing software function benchmarking, a physical hardware development board was used as a reference hardware system. The board was chosen since it runs on an instance of Core M with hardware design schematic that correlates to what was available as verified ARM IP models. Although the development board has a different hardware configuration from the developed virtual minimal system, it was used to adapt and optimize the initial platform hardware and software compilation methods in order to achieve a comparable cycle-count between the physical system and the virtual platform.

5.1 Reference development board

As a reference hardware system layout to the one developed using SoC designer, a standard off-the-shelf development board was used as a comparison target. The board's hardware configuration is more complex and includes several additional blocks than the minimal microcontroller developed in the virtual platform. As shown in 5.2 the development board includes reciprocal IP consisting of one Core M processor running at 100 MHz, RAM memory, UART and a bus matrix. On the other hand, also included are a flash memory and flash cache accelerator block which are central to the execution of the system. These two blocks replace what would correspond to the second ROM RAM memory module in the virtual platform. Furthermore, the board also includes direct memory access controller (DMAC) blocks, timers and additional I/O connectivity ports which fall outside of the scope of the software executed during the testing of the platforms and is therefore not utilized or illustrated.

The key hardware aspects of the development board system which differs from the virtual platform is the memory subsystem. For starters, the software binary is placed on the flash memory as opposed to a RAM module, and the system thereafter executes pre-fetched instructions through the flash accelerator. The purpose of the flash cache is to counteract the miss-match in speed between the flash memory access times and the higher clocking rate of the processor by fetching multiple- instruction blocks large enough to keep the processor executing for a certain number of cycles. This accelerator also implements a branch cache that adapts to avoid wait states and minimize the effect of disrupting sequential code

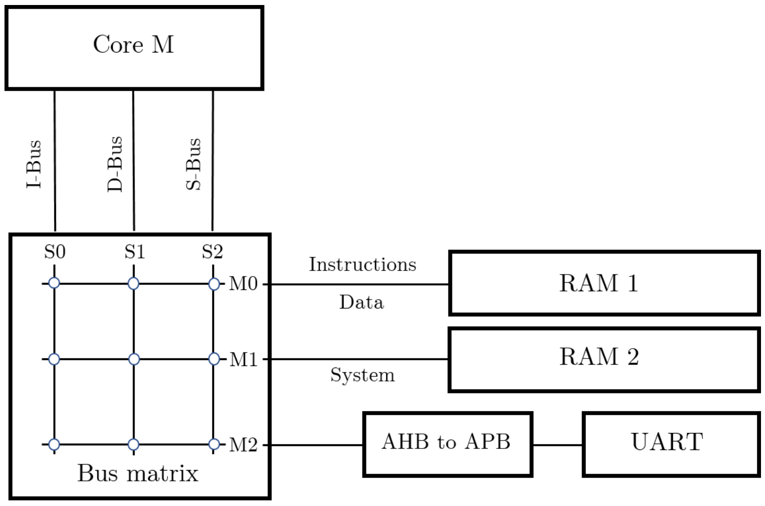

Figure 5.1: Simplified virtual platform micro-controller hardware configuration.

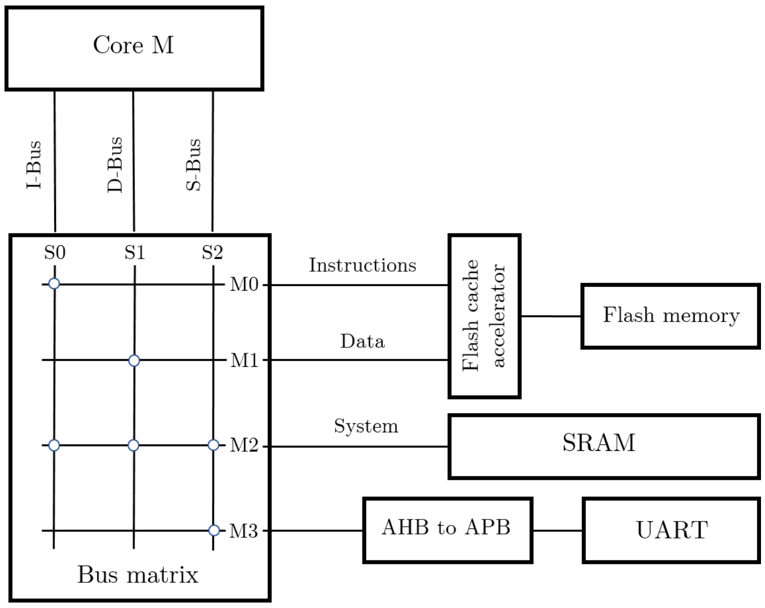

Figure 5.2: Simplified reference development board micro-controller hardware configuration.

execution when branching. The vendor of the board claims that this configuration averages zero-wait-state execution. Additionally, the system RAM (SRAM) that the board uses is marked as zero-wait state.

Another decisive factor in the performance of the board is its bus matrix, which is implemented so that all the AHB masters, in this case the instruction, data and system buses can achieve independent none-blocking access to the different AHB slaves. This entails that the bus matrix should not introduce wait states to the system and even several peripherals can be controlled simultaneously by different AHB masters. The nodes in illustrated inside the bus matrix in 5.2 show the possible bus collision points for bus access on different master ports (M0-M3) by different requestors (bus masters) on the slave ports (S0-S2).

5.2 Altering the bus and memory subsystem

The bus and memory layout in the virtual platform microcontroller were modified into various hardware configurations to investigate the performance impact on software benchmarking. The first optimization done to the initial design, was altering the memory subsystem to reciprocate the reference development board design by eliminate the access times on both the data/instruction and system RAM memory modules. This entailed reducing the column access strobe (CAS), row address strobe (RAS) and page latencies to zero on the behavioural models to simulate the zero-wait-state present on the board.

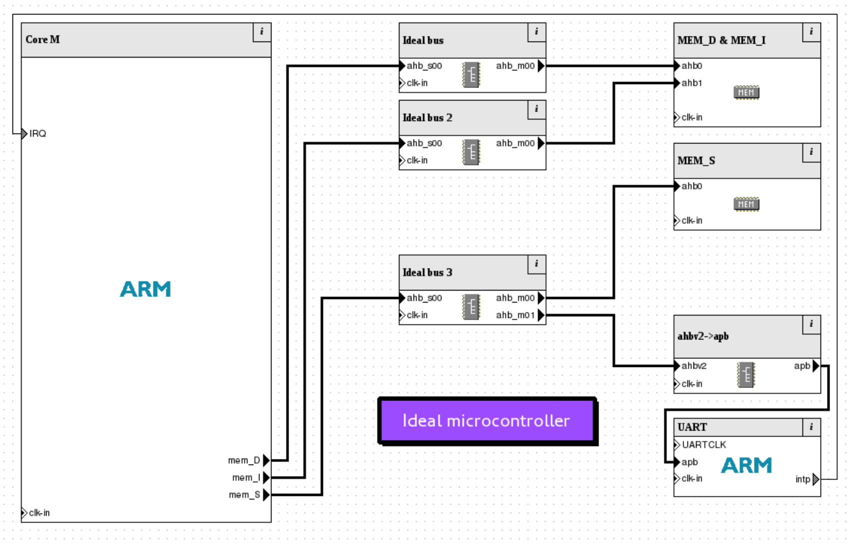

Figure 5.3: Ideal bus microcontroller configuration in SoC Designer.

With a zero-wait-state achieved from the memory blocks, two variations of the bus matrix model were compiled and tested in the system simulation. The difference between the two was the bus arbitration scheme, which varied from fixed, as in the initial configuration, to a round-robin scheme. Round-robin arbitration is performed on every active clock-cycle of the HREADYM. The port priority decreases from the first requestor on slave port 0 as the port index increases. When several requests are made, the active priority goes to the highest priority requestor, relative to the active one. Additionally, fixed length bursts are not interrupted but are allowed to finished before passing the priority to another requesting port. [26]

To further model the memory subsystem behavior on development board and suppress the wait-states introduced by the bus matrix acting as a blocking resource, an alternative ideal bus configuration was introduced. This consisted on providing an exclusive ideal AHB bus model for every one of the Instruction, Data and System buses as seen on figure 5.3. To enable this change, the Instruction/Data RAM memory was replaced with an equivalent model with two AHB slave inputs.

5.3 Benchmark software functions

A wide array of 62 use cases of digital signal processing (DSP) and cellular IoT specific functions were included into a testbench for benchmarking performance across the different systems. Several of the functions included in the test suite utilize core-specific CMSIS (Cortex Microcontroller Software Interface Standard) DSP package functions [32].

5.4 The testbench ELF image

The executable image loaded on to the two systems for testing consisted of three different parts linked together: the start-up code, code for the benchmark software functions and a command parser. Both the benchmark code and the command parser are pre-written C programs which were compiled into individual static libraries prior to the final linking into the start-up code. The command parser allows for a more dynamic test environment making it possible to execute specific benchmark functions depending on the system serial data input through the UART. The command parser contains entries that represent each supported test case. The purpose of the parser is to determine if the command is present as an entry and if the provided parameters are supported for that test function.

In order to measure the cycle count of each function in the same way, a special debug register included in the processing core was utilized. This register increments continuously with every clock-cycle during execution, and it can be reset at any moment to initialize a new measurement. Each entry thus begins by parsing the input and if successful, the counter is reset before calling the test function, probing the register for a cycle-count as it returns. The final count is then transmitted to the UART. In case of failure a standardized failure message is sent instead.

The flow structure of the testing program loaded on to the system consists of the start-up code which leads to a main function that polls the UART address through continuous AHB reads and forwards the received characters to the command parser, which in turn calls upon the relevant benchmark function with the provided parameters and measures the elapsed clock-cycles.

5.5 Compilation of the test binaries

Comparing the software performance on the platforms required the execution of the exact same instructions and hence the benchmark functions had to be compiled in the same manner. However, legacy code restrictions in both the start-up of the development board and the virtual core did not allow to use the same compiler for both the test ELF images. Nevertheless, the issue was solved in the following way:

Both the benchmarking functions and command parser were compiled equally into static libraries with the GNU ARM Embedded Toolchain 5.2.1 (20151202 release) [33]. In the case of the development board, both the start-up code and linking of the final image were also done with this compiler toolchain. On the other hand, the virtual platform start-up code was compiled and linked together with the pre-compiled static libraries using the ARM Compiler 5.06u4 Toolchain, to ensure that the function calls and cycle-measurements are the same at a instruction level.

5.5.1 Build automation

The software source compilation was done using GNU make, a utility which determines what files need to be compiled, and issues commands to compile them according to user defined directives called makefiles [34]. Through the use of makefiles the build process was made dynamic, allowing the compilation of the same software ELF image for different hardware configurations. This is specially valuable when changing core add-on presence such as co-processors, which require a different set of flags for compiling with the correct library code corresponding to the hardware layout. Furthermore, since the project required build support for several targets and compilers, makefiles were essential in order to support the building of static libraries utilizing a librarian and linker from mixed toolchains.

5.6 Testing the systems

A Python script was utilized to administer the testing functions and manage the results for both systems, the script performed an identical test suite on both systems, except for the UART serial input communication method which is target dependent. The development board was connected via USB and the communication was performed through a teletypewriter (TTY) R/W file. The Pyserial package was employed for this purpose. In the case of the virtual platform, as previously mentioned the communication is done via TCP/IP socket instantiation of the standard python socket class.

The script performs the serialized character inputs to the target and then performs a blocking read of the expected response byte length including the cycle-count of the tested function from the command parser. In the case of failure, such as the serial data arriving corrupted to the target, the command string is re-issued until success is achieved. The time elapsed for each test case is also measured and recorded in the script. An overview of the test system's flow is presented in figures 5.4 and 5.5.

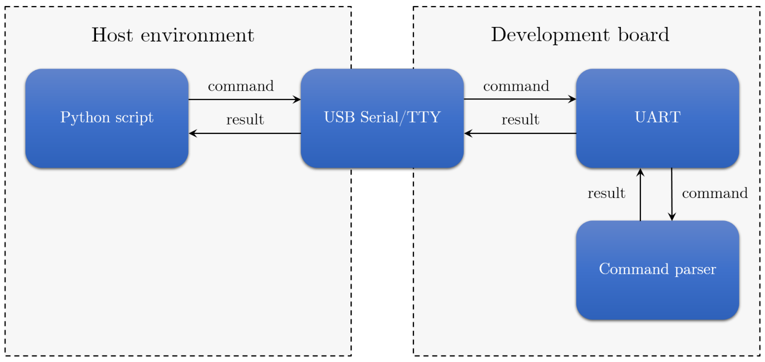

Figure 5.4: Benchmarking software test flow for development board.

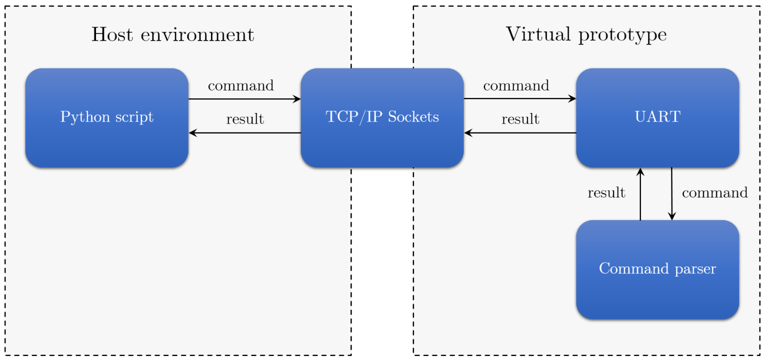

Figure 5.5: Benchmarking software test flow for virtual platform.

5.6.1 Host Environment Specifications

- Intel Core i7-6700 CPU (8 cores @ 3.40GHz)
  - single core usage for simulations
- 15.4 GiB RAM
- CentOS 7 (64bit)
- SoC Designer (9.1.0) for Linux64 with gcc 4.8.3

Chapter 6

Co-design and Integration of Hardware Blocks

Following the software performance benchmarking of the virtual micro-controller, the scalability of both the virtual prototype and the simulation platform was investigated from a hardware software co-design perspective for an embedded cellular IoT system. The system being prototyped exists as an FPGA implementation that has a completely different micro-controller configuration due to the hard IP constraints of the FPGA board. It is nevertheless comprised of several hardware accelerators and a real-time operating system that manages a multi-threaded software application. Thus, to expand the virtual prototype towards the FPGA system implementation, and achieving the functional and cycle-accurate compatibility of additional hardware IP blocks to the base microcontroller, the integration and testing of three specific modules was carried out. This was done in several steps:

- Wrapping the SystemC HLS pre-synthesis target hardware blocks with the ESL APIs in order to provide simulation compatibility with SoC designer.
- Creating the interface blocks between the developed minimal microcontroller made up of ARM verified IP models, which utilizes transactor signals for the protocol implementation, and the pin level implementation of the IP models under testing.
- Integrating the wrapped hardware blocks into the working virtual platform and achieving functional connectivity by developing a Transactor to SystemC translation block chain.
- Debugging and performing behavioural verification for functional accuracy on the individual blocks through software test-cases and utilization of CADI features.
- Performing cycle-count measurements of individual blocks on the platform and comparing them to their FPGA implementation equivalents to evaluate latency deltas between the two.
- Porting FreeRTOS to the virtual platform and carrying out real-time synchronous testing of the blocks under analysis.

Since the FPGA system hardware and software configuration are different from the virtual prototype, it is only used to check the validity of the individual HW accelerator implementations in the virtual platform and not the complete system.

6.1 Integrated hardware blocks

The two hardware blocks that were originally intended to be integrated into the virtual prototype were late design stage verified IP, developed in Catapult C with an HLS design flow. These blocks are hardware accelerators involved in the uplink and downlink data processing paths and are thus hereforth referred to as the *Uplink HW accelerator* and the *Downlink HW accelerator* for confidentiality reasons. Both of the accelerators utilize a subset of the AXI-stream protocol signals to interface with the system and therefore posed a challenge to integrate into the developed virtual micro-controller due to the communication protocol incompatibility as well as the Transactor to SystemC signal type miss-match. Both accelerators also have additional signals for control/configuration of the block which are set outside of the protocol interface. To provide these signals, a native CASI implementing signal generator was developed to configure the required static values for each test-case at a signal level.

The communication protocol issue was thereafter addressed by integrating and verifying a third block into the virtual prototype in SoC Designer: an early development stage *AHB to AXI-stream bridge* developed in SystemC at RTL level. This model includes among other things buffering FIFOs that introduce latency in the bridge. The integration of two instances of this module thus allows for a pin-level, signal type and protocol matching for both the Uplink and Downlink HW accelerators in the simulation. The simulation signal type difference was resolved by developing a series of zero-latency CASI blocks to form a Transactor to generic SystemC signal translation chain which is described later in section 6.4.

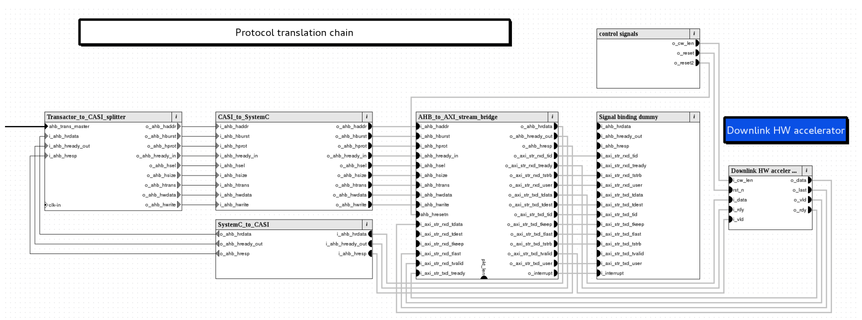

Figure 6.1: Downlink HW accelerator signal chain, containing a Transactor to generic SystemC signal translation chain, AHB to AXI-stream bridge, control signal generator, unused signal binding dummy and the HW accelerator.

6.2 Wrapping SystemC with ESL APIs

As described in chapter 3, in order for a C++ or SystemC level model to run in a SoC Designer simulation, the model must primarily comply to the CASI library. Thereafter, the communication methods, port types and pin-level signal representations will depend on if the model has been natively designed with cycle-based scheduling from CASI or relies on SystemC ports and signal types for an event-driven section block. As the blocks under test were created through HLS, the SystemC integration approach was default.

The wrapping consists of creating new source code which instantiates a CASI top-level module with a one to one mapping of the ports in the target SystemC module by either directly connect to the internal port, or to add intermediate signals which are updated with the wrapper port's value and cast the signal type into the required internal type class, which is read by the corresponding port in the SystemC module that is being wrapped. The signal conversion and forwarding was done with sensitivity lists sensitive only to the wrapped signals and not to the clock in order to avoid adding any register latency to the simulation from the wrapper.

The clock port is tied to the CASI master clock signal directly, so that there is no need to manually clock them with a component inside the simulation. The clock can however be handled as a standard signal to connect to a clock-divider if there is a need to introduce different clock domains in the simulation.

In order to provide visibility into the registers, internal and intermediate signals and to be able to set hardware debug points when simulating the models in SoC Designer, the CADI was also implemented for both accelerator blocks. The CADI module in the wrapper was done so that there was SoC Designer register views for the wrapper ports and intermediate signals as well as the ports of the internal SystemC module. Furthermore, every register of the internal SystemC sub-module was also traced in order to provide high-visibility into the internal component simulation at a cycle-accurate level. Internal signal traces were also implemented to generate waveforms of the module execution.

Finally, the CADI module also implemented parameters that allow the enabling and disabling of the aforementioned debug functionality in each block from SoC Designer as booleans. Having the option to toggle debug features during execution prevents the simulation to do unnecessary processing on every clock-cycle that the user does not want to review, thus accelerating execution.

6.3 AHB to AXI-stream bridge

This module was functionally verified with the help of the CADI together with the HW blocks that were to be integrated by the scope of the thesis. As seen in figure 6.1 the AHB to AXI-stream bridge requires a signal binding dummy in order to function correctly within the simulation as it is in early development stage. As this block is not present in the FPGA implementation, all delays caused by the buffering are not investigated when analyzing the signal translation chain as they cannot be compared. These delays are inherent to the design and would be required in a physical implementation of the model.

6.4 Transactor to SystemC signal translation chain

To interface the micro-controller to the AHB to AXI-stream bridge and the Uplink/Downlink HW accelerators, a three module translation chain from a AHB Transactor signal to generic SystemC type signals was developed by implementing the CASI natively. The chain, illustrated in figure 6.2, is comprised of a Transactor to CASI signal splitter, a CASI to generic SystemC adapter and a generic SystemC to CASI signal adapter. These components had to be designed to not introduce wait-states in the chain or have any latency, in order to ensure that the sequence of the signal handshaking in the AHB protocol between the Core M (bus master) and the AHB to AXI-stream bridge (bus slave) is satisfied, and to not compromise the cycle-accuracy of the whole simulation.

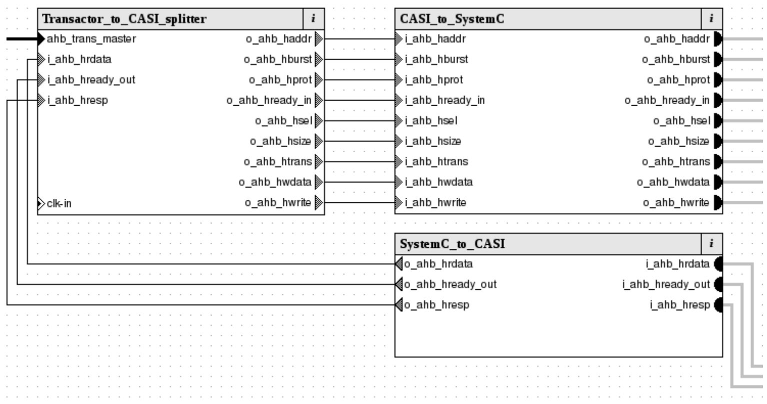

Figure 6.2: AHB Transactor signal conversion chain consisting of three blocks which convert a CASI transactor signal into generic SystemC signals and enhances communication protocol accuracy.

6.4.1 AHB transactor to CASI signal splitter

This component instantiates one transactor AHB port and the subset of AHB CASI signals shown in figure 6.2. On every clock-cycle it requests the value of the transactor for each of the signal through the AHB Transactor API, updates its internal CADI registers with the value, and drives out the same value on the CASI signal ports.

As seen in figure 6.3 and 6.4, the input signals to the ports of the transactor splitter block are forwarded through to the output signals in the same clock cycle. Thus, the transactor signal splitter does therefore not introduce any delay or affect the signals coming to and from between the microcontroller and the connected block.

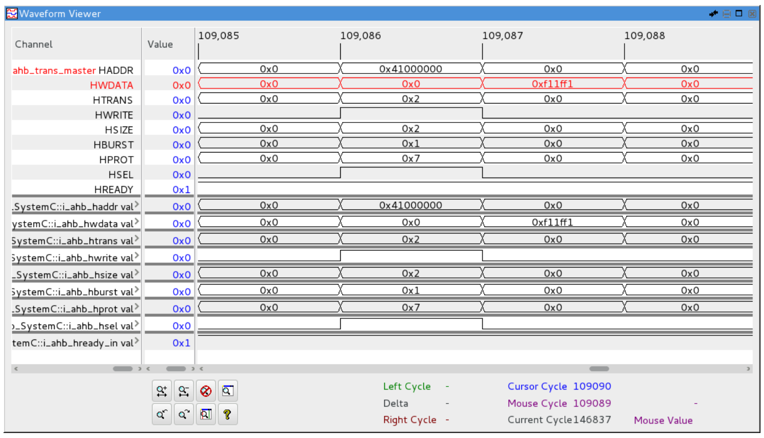

Figure 6.3: Waveforms of signal chain from transactor signal splitter to CASI to SystemC block with zero latency. Signals from the bus matrix is shown above the corresponding signals going to the CASI to generic SystemC conversion block.

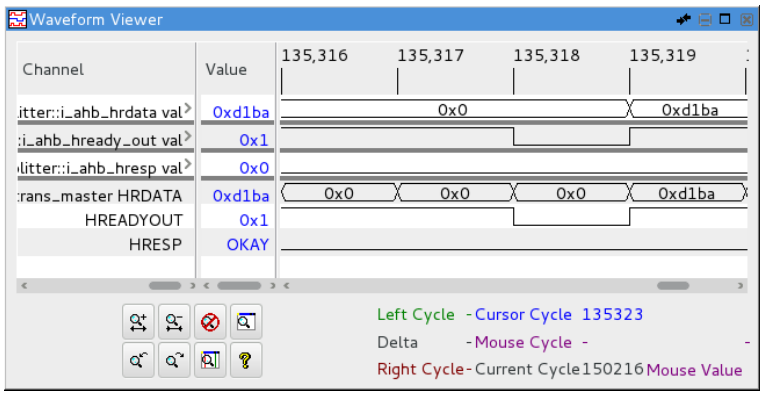

Figure 6.4: Waveforms of signal chain from transactor signal splitter to bus matrix with zero latency. The first three signals are input to the transactor signal splitter and the following three are output signals going back to the bus matrix.

6.4.2 CASI Signal to SystemC Signal adapters

This block similarly to the AHB Transactor splitter, only read the instantiated CASI ports, update the internal CADI register and write out to the corresponding value of each signal to the respective Generic SC master ports.

6.4.3 Verifying translation signal chain

A similar procedure as the verification of the AHB transactor to CASI signal splitter, which entailed comparison between the input and output signals, was applied to CASI to SystemC block and confirmed that no extra clock cycles were introduced by checking the signal registers of the components with CADI for the same clock-cycle.

6.5 Generating SoC Designer models

To generate a SoC Designer component, all the CASI compliant source code for the model needs to be compiled and linked into a shared object file (.so) that is compliant with the SoC Designer run-time environment. The tool utilized for the shared object generation was the GNU Compiler Collection (gcc version 4.8.3). The compilation was thus carried out with the C++ compiler (g++) using the position independent code directive (-fPIC) and including the expanded SystemC library from SoC Designer which includes the ESL APIs. The linking was thereafter done with the -shared option while including the SoC Designer common libraries included within the installation directory of the tool.

6.6 Hardware IP block cycle-count measurements

This section describes the methodology for measuring the latency of both the uplink and downlink HW accelerators on the two target platforms. The latency were measured the exact same way by executing a test case with the same data length and values.

6.6.1 FPGA

The existing target FPGA hardware configuration had to be altered to add an integrated logic analyzer (ILA) to the implementation which was set to trigger on the positive edge of the measured i_last and o_last signals for each of the HW accelerator blocks. This implementation was then synthesized and programmed into the FPGA. The measured signals are high (logic value one) exclusively during the clock cycle at which the last input data value is received and when the last output value is sent from the block. The difference between the edges of the signals thus represents the latency of the block in clock cycles as the ILA samples once per clock cycle.

6.6.2 SoC Designer

The performance of the blocks were measured by triggering CADI register brake points when the value of the i_last and o_last signals connected to the respective internal ports of the SystemC block inside the CASI component transitioned from low to high.

6.7 RTOS hardware-software synchronization

In order to verify hardware-software synchronization from a RTOS driven full system perspective, FreeRTOS was ported to the virtual prototype. The hardware configuration used together with the RTOS contained a microcontroller, a direct memory access controller (DMAC), a downlink HW accelerator and a uplink HW accelerator, which is illustrated in fig 6.6.

6.7.1 FreeRTOS port

FreeRTOS version 9.0.0 was used as the RTOS as there existed a port for the Core M. However, low level details such as the definition of interrupt handlers, scatter file details and hardware addresses were updated to convey with the virtual prototype's hardware configuration. Drivers for the downlink and uplink HW accelerator blocks as well as drivers for the DMAC were also developed.

6.7.2 DMAC

The DMAC was introduced in the hardware configuration to simulate a more complex system where the handling of data transfer from peripherals and memories are carried out by the DMAC instead of the processor, increasing the complexity of the bus accesses representing a more realistic full system, where the processor is not the only bus master. It also serves to verify the functionality of peripheral interrupts inside the simulation environment.

6.7.3 Full system simulation

Figure 6.6 shows the final hardware of the virtual prototype. To verify this system, a use-case with three threads was implemented consisting of: a simple printing thread, one thread to read and write to the downlink HW accelerator directly, and a thread interacting with the uplink HW accelerator through a DMAC. The threads were constructed to have both a critical section making the AHB transactions and printing thread safe. Verified data from existing test-benches was utilized for each of the threads and the passing conditions were set to asserting the expected data was correctly received after the correct number of AHB read transactions; comparing each value as it is read. Furthermore, the threads yield between sending data to a HW accelerator to not stall the system until it is either scheduled again by the RTOS due to an appropriate time passing, or after receiving an terminal count interrupt from the DMAC, in which its handler prompts a read request to the HW accelerator. One possible execution of the system use-case test is illustrated in figure 6.5. All the treads were run continuously in a loop as an embedded system would managing all the resources. The execution was both debugged and verified by utilizing the CAPI software trace function of the processor which shows a historical view of the program's execution in a function by function manner.

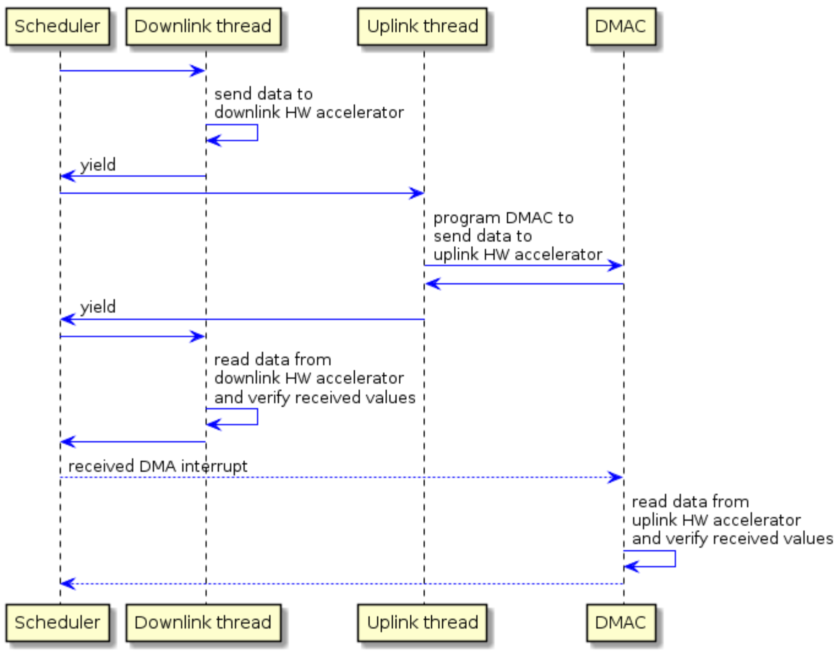

Figure 6.5: Sequence diagram for one possible execution of HW/SW co-simulation test program excluding printing thread.

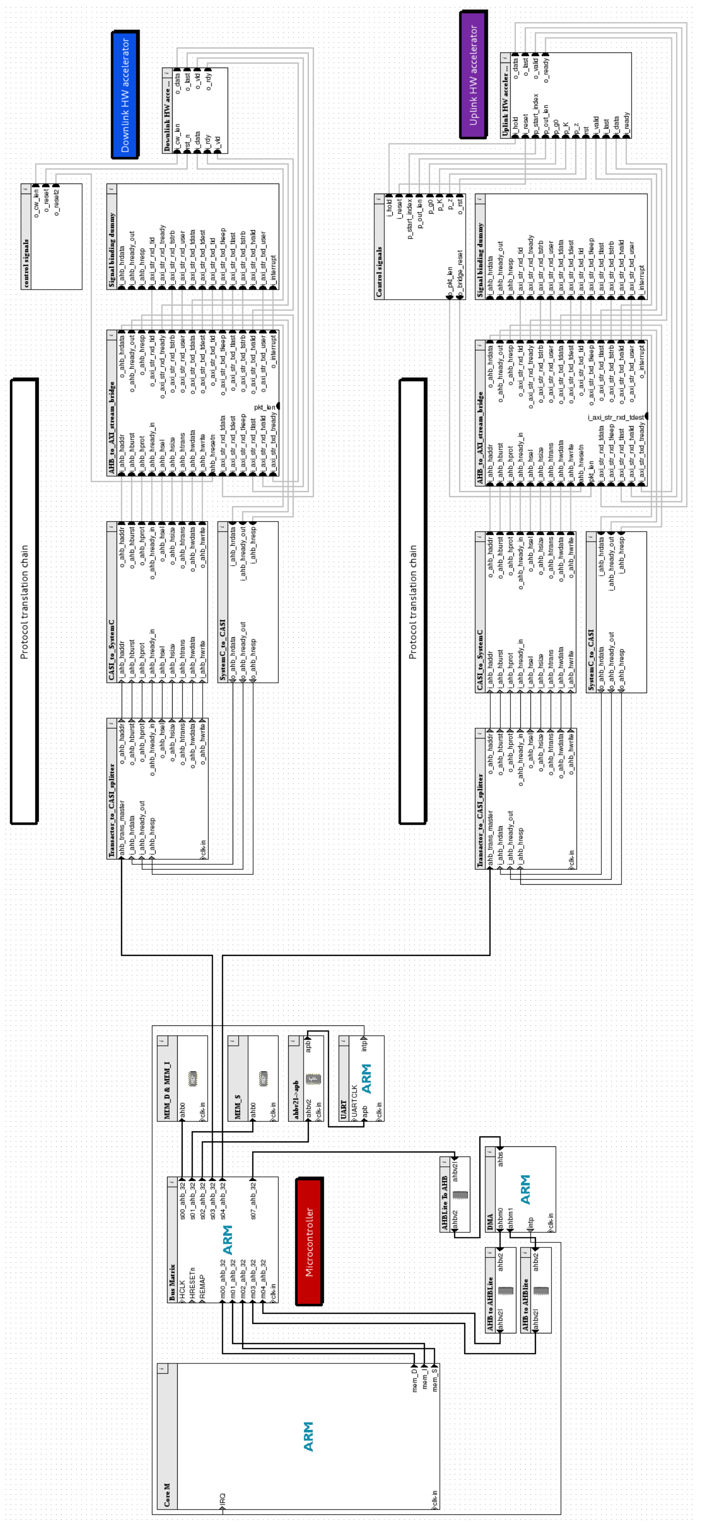

Figure 6.6: The Final virtual prototype. The system is divided into three sections, a micro-controller including a DMAC to the left, a Downlink HW accelerator to the upper right and an Uplink HW accelerator to the lower right. Both the Downlink and Uplink HW accelerators are connected to the micro-controller bus matrix through a signal protocol translation chain. The prototype executed all the test threads concurrently, passing all tests at an average clock cycle simulation frequency of 10.6 kHz.

Chapter 7

Results

The following chapter consists of a condensed presentation of the quantitative results achieved in this thesis. The results are comprised of the comparisons between the co-simulated virtual platform and the reference development board and FPGA in terms of software performance, cycle count, simulation speed and SystemC HLS model to FPGA synthesized IP component fidelity.

7.1 Sotfware performance of virtual microcontroller prototypes and development board

This section presents the results from chapter 5 and evaluates the software performance of the various targets on which the test-bench was executed.

Table 7.1 lists and labels the different virtual prototype configurations (VP A-F) used throughout the test cases as described in section 5.2. The different aspects investigated are modeled memory latency, modeled bus and its arbitration scheme. Matrix being a model of the existing multi-layer bus matrix component and ideal referring to an ideal behavioural model.

Virtual prototype configuration	**Memory latency**	**Bus**	**Arbitration**
VP A	standard	matrix	fixed
VP B	standard	matrix	round robin
VP C	standard	ideal	round robin
VP D	none	matrix	fixed
VP E	none	matrix	round robin
VP F	none	ideal	round robin

Table 7.1: Virtual platform configurations where memory latency, bus model and the arbitration scheme of the bus are inspected.

7.1.1 Cycle count

Table 7.2 compares virtual platform configurations to the development board with regard to clock cycle count when executing the test-bench. The relative average clock cycles column expresses the difference in elapsed clock cycles for each of the configurations as a percentage, relative to the clock-cycle count value obtained from the reference development board.

Virtual prototype configuration	**Relative average clock cycles (%)**
VP A	+ 68.77
VP B	+ 68.77
VP C	+ 49.68
VP D	+ 5.05
VP E	+ 5.05
VP F	- 5.19

Table 7.2: Software performance comparison between different virtual platform configurations relative to the reference development board.

It can be seen that VP configurations E and F are able to simulate the performance of the reference development board within an error margin of 5%, showing that with more detailed implementation description of the target it would be possible to replicate the performance cycle-accurately in a virtual model. See section 8.1.1 for more in-depth analysis.

Furthermore, subsets of the test-bench are presented ahead in figures 7.1-7.2 showing the clock cycle count for test functions on each of the different virtual platform configurations as well as the reference development board. It can be seen in these figures that there is a stable trend between the relative performance of the different targets.

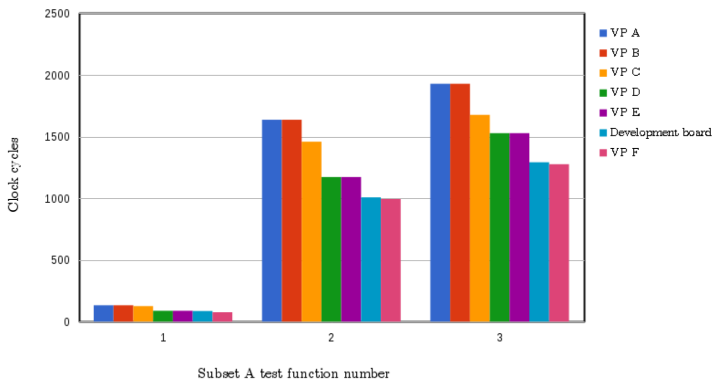

Figure 7.1: Cycle count for subset A of the software test-bench on the various targets.

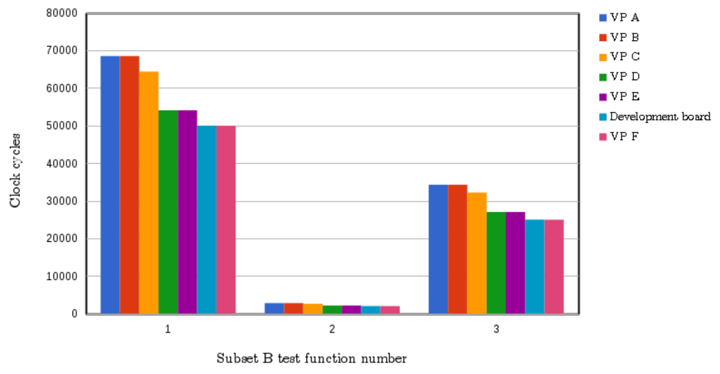

Figure 7.2: Cycle count for subset B of the software test-bench on the various targets.

7.1.2 Execution time and simulation speed

Table 7.3 presents execution time and speed for execution of a complete test-bench on both the reference development board and the different virtual prototype configurations. Clock rate for the development board is the actual frequency of physical clock, whereas the clock rate for the virtual platform configurations represents the execution speed in terms of average simulated clock cycles per second. The time factor is the multiplicity of the total elapsed time for the test-bench to complete for the various virtual platform configurations with respect to the development board test. The test time is measured from the python script.

Target	**Clock rate (kHz)**	**Test time (hh:mm:ss)**	**Time factor**
Development board	100000.000	00:01:19	1
VP A	51.398	07:07:39	326
VP B	50.169	07:16:29	333
VP C	69.646	05:24:59	248
VP D	49.984	05:25:46	249
VP E	49.034	05:26:10	249
VP F	66.781	04:12:03	192

Table 7.3: Total software test-bench execution time and simulation speed on virtual platform configurations with reference to development board target.

The time it takes for execution of the test-bench on the virtual configurations are on average 266 times longer than the development board which can be explained by both the complexity of the models comprising the simulated system as well as the overall host CPU load during testing. The reason behind these values will be elaborated on in section 8.1.2.

7.2 HW accelerator block latency measurements

This section presents the results from chapter 6. In table 7.4 comparison in clock cycles between running the pre-synthesized hardware accelerators in the virtual platform and post-synthesized on FPGA are presented. The measurements performed on the FPGA through the use of an ILA are shown in figures 7.3 - 7.4.

Accelerator block	**FPGA (clk cycles)**	**Virtual platform (clk cycles)**	**Difference (clk cycles)**
Downlink HW	50	49	1
Uplink HW	149	146	3

Table 7.4: HW block comparison in clock cycles between FPGA and virtual prototype pre-synthesis HLS implementation.

The above table shows that there is a difference between the FPGA and the virtual platform implementations; a delta of one and three clock cycles for the downlink and uplink hardware accelerator respectively. A clear source for this additional cycles would be the HLS process introducing library implementation specific RTL modification to adhere to physical constraints.

The following figures present the data received from the ILA output and shows the sample number on the horizontal axis and the logic value level on the vertical axis for the different signals. The sample rate is the clock-rate at which the ILA was connected to, which is the same as the tested block.

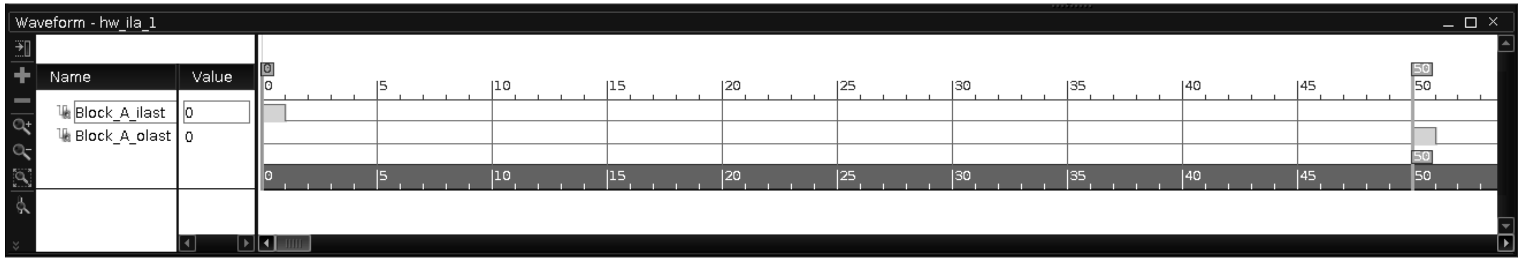

Figure 7.3: ILA view of waveforms showing the latency of 50 clock cycles in downlink HW block (block A) on FPGA target. The ILA is triggered from the positive edge of input last (i_last) signal to the block and the output last (o_last) from the block.

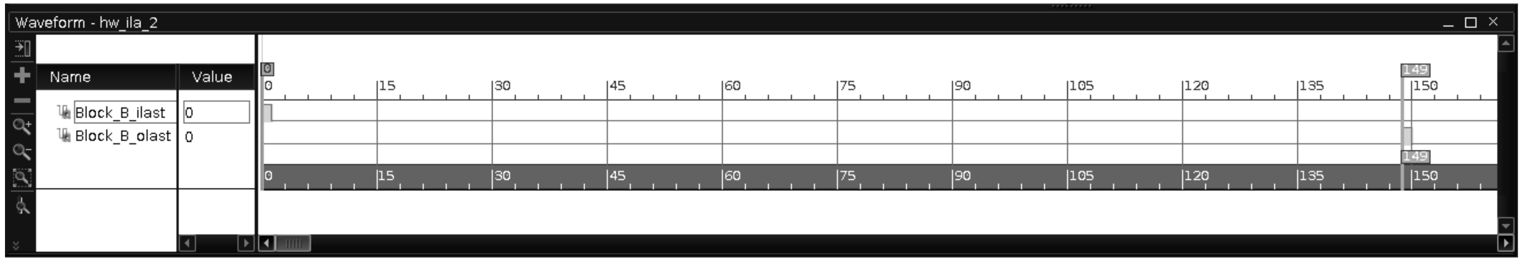

Figure 7.4: ILA view of waveforms showing the latency of 149 clock cycles in uplink HW block (block B) on FPGA target. The ILA is triggered from the positive edge of input last (i_last) signal to the block and the output last (o_last) from the block.

Chapter 8

Discussion and Conclusion

8.1 Virtual prototype versus development board and FPGA

8.1.1 Virtual micro-controller configurations

The presented results in table 7.2 by altering the virtual prototype micro-controller configuration an error margin of approximately 5% was achieved when trying to recreate the of the reference development boards clock-cycle performance for the test-software. This shows that given the known design and components of a specific silicon implementation, one can approximate the performance by behaviourally simulating the expected wait-states introduced by a specific sub-system. More specifically, by reducing the memory latency to zero to recreate the performance of an optimal zero-wait flash cache the source of the cycle-count difference can be narrowed down to the bus matrix model utilized, being less optimal than the development boards component but not completely ideal as modeled and will, in some cases, have at least one-clock cycle latency for a memory read access (e.g. when a cache miss occurs or during the execution of none-sequential code segments).

Furthermore, the lack of result variance between round-robin and fixed priority arbitration schemes in the bus matrix model used in the virtual microcontroller is expected due to the following. As previously shown in figure 5.1, even though there are three bus masters coming from the processor, all of the bus accesses are none-interfering with each other since the S-Bus exclusively accesses the RAM 2 module, whereas the I-Bus and D-Bus access the RAM 1 module sequentially as the processor does not generate instruction and data fetches on the same clock cycle. Therefore, as the micro-controller is the same and the instruction/data bus is maintained as M0 with the highest priority, the arbitration scheme does not result in any observed performance differential.

Taking the above into consideration it can be said that the virtual platform provides the potential to cycle-accurately simulate any system given that there is enough knowledge and information of the target system being modeled, as well as showing that confidence in the accuracy of the integrated models into a virtual prototype is essential.

8.1.2 Execution speed

In terms of execution speed, table 7.3 shows that the virtual prototype simulations are slower than the development board implementation by a time factor of 266 on average. This may depending on the use case be discouraging for the methodology however, for a simple system or subsystem of larger complex one the hardware-software co-simulation verification possibility may be justified as an operation that takes 1 second on an development board running at 100 MHz will take approximately 4 minutes. These values are directly coupled to the complexity of the actual models being included in the simulation; the same table shows a maximum clock cycle frequency difference of approximately 20kHz between the two most outlying virtual prototype configurations of the micro-controller (VP C and VP E). The simulation speed may thus vary considerably depending not only on the size of the system but also on the type of models integrated.

Furthermore, the SoC Designer simulation performance is coupled to the host environment. From the observations in this investigation, the virtual platform simulation speed is limited to the usage of a single thread and is thus highly dependant on host system specifications on processing power and memory as well as system load which thus affects the simulation time. This can be observed in the results from table 7.3, where there is not a clear linear relationship between the simulated clock cycles per second and the total simulation time taken from the python script. Some of the non-linear behaviour can be attributed to the overhead that arises from the time lost when a failed reception of UART packets through TCP/IP occurs and the re-sending of these is required. However, larger differences as the ones observed could be attributed to system load and host environment kernel scheduling. Nevertheless, the simulation speed (in clock-cycles per second) is a good reference to the complexity of the simulation as this value is generated as an average of the SoC Designer scheduled time taken to simulate one full clock cycle of the system.

Virtual prototype simulation as described in this thesis, is most suitable to testing integration and debugging of individual sub-systems from a high-iterative development point of view. This is due to the simulation speed correlation to the number, size and complexity of the components in the system. Nevertheless, a virtual prototype could also be suitable for low-iterative full system HW/SW architecture verification runs to increase design confidence through simulation; e.g. evaluating worst case scenarios of system load. For the continuous verification of a full hardware-software system in late development stages an FPGA is the preferred method as the relatively low simulation speed achieved in the virtual prototype micro-controller is not realistic for any modern cellular IoT SoC; as it may not be able to meet the real time requirements of synchronization with a cellular network.

8.1.3 Clocking frequency and physical factors

As SoC Designer virtual prototype is a simulation, no physical repercussions of a target technology are accounted for as it with be done with a synthesized FPGA target library. For example, since the simulation speed or clock cycle frequency is determined by the models, the system and simulation environment steer the clock rate; as opposed to an ASIC or FPGA where a physical clock dictates the system behaviour. Furthermore, the simulated clock frequency cannot be set to a specific value, and it cannot realistically reach a clock cycle simulation frequency comparable to the clocking rates possible on a development board. Therefore, there is no good way of examining if a resulting model is viable more than functionally and from the clock or time granularity of an RTL perspective because no signal propagation times or levels are considered in the virtual prototype. Consequently, there is still a need for some type of physical implementation whether that may be ASIC or FPGA to verify the sanity of the design for any target implementation for a given technology and clocking frequency.

8.1.4 Implementation and architecture exploration

The main advantage of this virtual prototype methodology is the possibility of accurate hardware and software co-design from the early stages of the development cycle from a micro-controller maturity level. The co-simulation described not only allows for the software to be tailored and verified on the hardware under development, but allows the observation of altering the hardware configurations and observe the impact of hardware accelerators on software performance.

Employing virtual platform simulations provides ease of hardware configuration altering by simplifying the development iterations. No logic synthesis or bitfile gate logic programming on to the FPGA is required, leaving only C++ hardware description compilation is needed when doing internal module changes. Consequently, this methodology results in shorter development iteration time taken per hardware change. This is specially true if using HLS pre-synthesis models, as RTL regeneration is not strictly required per iteration and can be done once the functionality of the model has been achieved in the virtual system to verify RTL as a final stage once there is confidence in the high-level model. Nonetheless, the initial overhead of creating a CASI wrapper for a raw C++ or SystemC model must be accounted for, and any modifications that need be done to it due to port or protocol changes.

Another benefit of incorporating the virtual prototype usage unto the development or design cycle is reduced physical hardware requirements per target. A completely virtual co-simulation and verification environment entails the elimination of external dependencies as the system development and system execution are contained to only the host environment, instead of having to acquire an FPGA board with the specific custom hard IP required. Additionally, a virtual environment not only allows for a completely custom IP solution, but even removes the limitation that the size of the gate arrays presents in an FPGA board, as the number and size of hardware blocks that can be synthesized is vastly higher. The only factors to be considered thus is the memory size of the host environment and its capacity to simulate any given large scale system at a reasonable speed.

With no physical commitment to hardware configurations there is great flexibility and ease in redesign. The ease of exchanging the hardware elements thus leads to the possibility of optimizing the current system with a wider range of possibilities than otherwise possible with a FPGA development board or use of an ASIC, as even the effect of swapping a processor can be inspected with relative ease; assuming a verified RTL model is available of the IP that is to be integrated.

8.1.5 Debugging capabilities

With the use of SoC Designer, internal hardware module visibility is vastly greater as all internal signals and registers can be traced directly during simulation at any point, whereas in the FPGA implementation the visibility is limited to external block signals that are only visible when adding additional hardware such as ILA blocks; which require additional FPGA space to synthesize and effectively alter the design with unwanted hardware being implemented. Moreover, any modeled hardware can be extended with CADI at any stage to fit verification needs, which would for an FPGA implementation require hardware design changes if a physical debug interface were to be added.

Furthermore, SoC Designer includes views to visualize the usage of the debug functionality, whereas this could only be done with the use of both an implementation of a debug interface on the hardware component together with external software to control and view the debug interface. One of these debug views included in SoC Designer shows software flow through out the execution and present clock cycle count duration per function allowing for in-depth software execution investigation as well as ease of finding bottlenecks within the software.

Since SoC Designer deals with simulations, it is possible to add any custom software functionality to hardware models, even functionality which would require an additional hardware component. This is key in providing visibility into the execution stack for a processor model and the cycle-accurate model of the UART provided by ARM used in the microcontroller includes capability to interact over TCP which is not present in the physical hardware.

8.1.6 CASI and SystemC wrapped systems

The methodology described in this report to achieve the final virtual prototype system in SoC Designer includes two different types of models: native CASI implementing models and SystemC models wrapped with CASI. Although the possibility of combining CASI modules generated from RTL or not, with SystemC is convenient for early design and integration of HLS pre-synthesis model, this presents a significant performance drop for the simulation. As it can be observed in the results chapter, from the pure CASI model virtual micro-controller prototypes clock simulation speeds averaging 56.2 kHz in 7.3 the speed of the complete final prototype is reduced to 10.6 kHz.

The simulation speed decrease is mostly due to the inclusion of the CASI wrapped SystemC uplink and downlink HW accelerators. Logically, the addition of two subsystems to the micro-controller will decrease simulation speed, however the addition of native CASI subsystems in the development did not result in performance differences of such magnitude. It must also be considered that the transactor to SystemC signal translation chain developed may also introduced some unwanted complexity to the system as it must do translation for every clock-cycle and there are two instances of this in the final prototype which may account of a significant part of the simulation speed loss.

It is also worth noting that the usage of SystemC type signals in an SoC Designer simulation lacks for the support of break-points and signal value viewing on the wires, leaving this functionality to have to be implemented to the module via CADI for input signal values on every clock-cycles as registers. It is therefore more optimal to create a virtual prototype exclusively with CASI native models.

8.1.7 HLS pre-synthesis HW accelerator block latency

The difference in clock-cycle latency between the FPGA blocks illustrated in table 7.4 can be explained by the lack of library implementation specific information that is added during the Catapult HLS process. Firstly, all blocks that do not have an output signal register in the high-level description will receive one after the synthesis. Since both of the HW accelerators in question did not include such registers, this accounts for the one cycle latency difference measured in both blocks. Secondly, the target frequency of the blocks causes the generated uplink HW accelerator RTL to include a pipeline of two stages in order to ensure the implementation can have enough time slack between the clock-cycle period and the propagation delays in the critical signal paths. These delays, in addition to the output register, results in a total expected additional latency of three clock-cycles in the FPGA implementation.

Although the virtual prototype may not be considered cycle-accurate as per the definition coined in section 2.2, the simulation is fully cycle-accurate to to what the model it is executing describes. With knowledge of the target implementation latency, the discrepancy between the two the HLS pre-synthesis models could be adjusted to include the discussed changes that occur in the HLS and thus eliminate the cycle-count delta. Furthermore, when using SystemC wrapped HLS models in a SoC Designer simulation with CASI, it is vital to note that even if a completely functionally and behaviourally accurate model is achieved, internal coherence of the model in terms of such as structural and data organizational accuracy is not guaranteed.

Ideally, all models should thus be created from the final RTL regardless of HLS is being employed or not in order to guarantee model fidelity to a physical implementation. Thus, it must be stated that having a cycle-accurate platform will only result in a cycle-accurate system simulation if there is a very high degree of confidence in the model accuracy through verification.

8.2 Conclusions

The outcome of this thesis is condensed into the following statements regarding the virtual co-simulation platform developed in SoC Designer and the various virtual prototype configurations investigated:

A fully cycle-accurate hardware and software co-simulation platform for a cellular IoT system has been developed through a simulator with a cycle-by-cycle level granularity. This drives cycle-scheduled CASI C++ models generated from RTL, including a processor core which in turn stimulates the complete system which includes SystemC event-driven blocks in a cycle-by-cycle basis.

A virtual prototype with co-simulation capabilities can perform software execution with reliable functional performance and cycle-accurate clock counts, given that the models included of the target configuration are verified.

The utilization of behavioural model components in a virtual prototype can be used to model the desired software execution performance of a reference commercial physical target hardware micro-controller within a 5% error margin, in order to provide the constraints for IP required to match that performance. These simulations are performed at an average factor of 266 times slower than a development board clocked at 100 MHz.

The latency and functionality of C++ designed HLS pre-synthesis hardware blocks in the final virtual prototype is cycle accurate. However, in order for the latency to be equal to a physical FPGA or ASIC implementation the target frequency of the blocks must be taken into account as the resulting RTL in the HLS process which is synthesized for a specific library will add pipelines on critical signal paths in order to meet timing constraints.

SoC Designer as a virtual co-simulation platform to create virtual prototypes can provide significant reduction in development time by providing co-design capabilities from an early design phase with lower architectural commitment, no physical component requirements, shorter iteration times between hardware design changes, increased debug visibility and at the cost of significantly reduced system execution speed compared to an FPGA.

The final virtual prototype developed can fully co-simulate a RTOS driven hardware-software system with multiple threading and various hardware accelerators concurrently at a clock cycle simulation speed of 10.6 kHz, which could be further improved by removing the use of SystemC models in the simulation.

8.3 Future work

Building upon the results and conclusions reached in this thesis, the consequential task would be to expand the virtual prototype following the methodology proposed in this thesis, to form a complete cellular IoT digital modem system to perform basic cellular tasks such as band scan and cell search in a simulated way. The full system, should however consider the following:

Adopting an approach that utilizes models based exclusively from RTL whether it be generated through HLS or not, and thereafter utilizing ARM Model Studio to generate clock-cycle scheduled CASI components of the whole system to possibly achieve a more structurally accurate models that would result in a more cycle-accurate representation of the models. This results could then be compared to this investigation as well as standard RTL simulation in terms of speed, accuracy and implementation effort.

The utilization of ARM Fast Models to develop a parallel system in which overall simulation performance is improved at the cost of cycle-accuracy in order to allow efficient software debugging and to possibly accelerate the simulation by skipping none-debug system time sections such as system initialization sequences at boot. This would entail incorporating functional, loosely-timed models in addition to cycle-accurate models and could be used in combination to swap in between the two with the use of break-points in order to have cycle-accurate granularity only when required and higher simulation speed otherwise when higher model abstraction can be tolerated.

References

[1] J. Teich, “Hardware/Software codesign: The past, the present, and predicting the future”. Proceedings of the IEEE, vol. 100, no. Centennial-Issue, pp. 1411–1430, 2012.

[2] J. A. Rowson, "Hardware/Software Co-Simulation", 31st Design Automation Conference, 1994, pp. 439-440.

[3] A. Sayinta, G. Canverdi, M. Pauwels, A. Alshawa and W. Dehaene, "A mixed abstraction level co-simulation case study using SystemC for system on chip verification", 2003 Design, Automation and Test in Europe Conference and Exhibition, 2003, pp. 95-100 suppl.

[4] Platform Architect MCO, Synopsys, Inc, 2017 `https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html`, accessed 2017-10-04

[5] Transaction Level Modeling, Mentor, 2017
`https://www.mentor.com/esl/vista/tlm/`, accessed 2017-10-04

[6] Grant Martin, Brian Bailey, and Andrew Piziali. ESL Design and Verification: A Prescription for Electronic System Level Methodology. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 2007.

[7] N.S Voros and K, Masselos (eds.), System Level Design of Reconfigurable Systems-on Chip, 15-26, 2005, Springer.

[8] I.Kuon, J.Rose, “Measuring the Gap Between FPGAs and ASICs”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (Volume: 26, Issue: 2, Feb. 2007), 2007.

[9] N.Abdelli, A-M Fouilliart, N.Julien, E.Senn, “High-Level Power Estimation of FPGA”, Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on.

[10] FPGA vs. ASIC, Xilinx Inc, 2017.
`https://www.xilinx.com/fpga/asic.htm`, accessed 2017-10-04.

[11] L.Séméria, A.Ghosh, Methodology for Hardware/Software Co-verification in C/C++, Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific.

[12] T.Grötker, S.Liao, G.Martin and S.Swan, (2002). System design with SystemC. Boston : Kluwer Academic, c2002.

[13] P.Schaumont, A Practical Introduction to Hardware/Software Codesign, 22-24, 2012, Springer Science & Business Media.

[14] A.Khan, M.Vijayaraghavan, S.Boyd-Wickizer and Arvind, “Fast and Cycle-Accurate Modeling of a Multicore Processor”, Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on.

[15] P.Coussy, A.Morawiec, High-Level Synthesis: From Algorithm to Digital Circuit. New York: Springer, ©2008.

[16] C. Economakos, H. Sidiropoulos and G. Economakos, "Rapid prototyping of digital controllers using FPGAs and ESL/HLS design methodologies," 2013 19th International Conference on Automation and Computing, London, 2013, pp. 1-6.

[17] Catapult High-Level Synthesis, Mentor graphics, `https://www.mentor.com/hls-lp/catapult-high-level-synthesis`, accessed 2017-10-04

[18] IEEE Standard for the Standard SystemC Language Reference Manual, IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005), IEEE Computer Society, 2012.

[19] SystemC, Accellera System Initiative, 2016. `http://accellera.org/downloads/standards/systemc`, accessed 2017-10-04

[20] OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL, Open SystemC Initiative (OSCI), document version JA32, 2009.

[21] ARM SoC Designer, ARM Ltd. `https://developer.arm.com/products/system-design/cycle-models/arm-soc-designer`, accessed 2017-10-04.

[22] ARM IP Exchange, ARM Inc. `http://www.armipexchange.com`, accessed 2017-10-04.

[23] B.Neifert, Rob Kaye, “High Performance or Cycle Accuracy? You can have both”, ARM White paper, ARM Ltd. 2012.

[24] Cycle Model Studio, ARM Ltd. `https://developer.arm.com/products/system-design/cycle-models/cycle-model-studio`, accessed 2017-10-04.

[25] SoC Designer Version 9.1.0, ESL API Developer’s Guide. ARM Ltd. 2017 `http://infocenter.arm.com/help/topic/com.arm.doc.dui1090a/cycle_models_SoCD_ESL_API_Developer_Guide_v9_1_0_DUI1090A_en.pdf`, accessed 2017-10-04.

[26] Arbitration and locked transfers, AMBA Design Kit Technical Reference Manual. ARM Ltd. 2017. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0243c/I1020077.html`, accessed 2017-10-04.

[27] ARM v7-M Architecture Reference Manual (Issue E.b), ARM, 2014.

[28] Tailoring input/output functions in the C and C++ libraries, ARM Limited, 2014.
`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0808g/chr1358938930366.html`, accessed 2017-10-04.

[29] Target dependencies on low-level functions in the C and C++ libraries, ARM Limited, 2014-2016.
`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0808g/chr1358938930615.html`, accessed 2017-10-04.

[30] ARM Compiler 5 Documentation, ARM Developer, ARM Ltd 2017.
`https://developer.arm.com/products/software-development-tools/compilers/arm-compiler/docs/version-5`, accessed 2017-10-04.

[31] ARM compiler armlink user guide, ARM Limited, 2014.
`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0803g/pge1362065970010.html`, accessed 2017-10-04.

[32] CMSIS - Cortex Microcontroller Software Interface Standard, ARM.
`https://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php`, accessed 2017-10-04.

[33] GNU ARM Embedded Toolchain 5-2015-q4-major
`https://launchpad.net/gcc-arm-embedded/5.0/5-2015-q4-major`, accessed 2017-10-04.

[34] GNU make, Free Software Foundation, Inc.
`https://www.gnu.org/software/make/manual/make.html`, accessed 2017-10-04.

[35] FreeRTOS. Copyright (C) 2010-2016 Real Time Engineers Ltd.
`http://www.freertos.org`, accessed 2017-10-04.