Politecnico di Torino

Master Thesis Report

submitted towards fulfillment of the Double Degree academic program
Master of Science in Embedded Systems
at the **Department of Electronics and Telecommunications (DET)**
with **Kungliga Tekniska Högskolan (KTH)**

by
Giulia Cioffi
March, 2020

Title :

UVM Test-bench acceleration on FPGA

Internship Advisor: Yann Oddos, Apple Technology Services B.V. & Co. KG
Internship Co-advisor : Philipp Kadletz, Apple Technology Services B.V. & Co. KG
University Supervisor: Prof. Edgar Ernesto Sanchez Sanchez, Politecnico di Torino

Abstract

The complexity of integrated circuits is increasing much faster than what CAD tools can handle. Even at IP level test cases can run for several hours or even days. Researchers and developers have introduced several techniques to tackle this issue. One promising approach to speed-up the verification process is co-emulation.
Before diving into details, this thesis gives a background on Universal Verification Methodology (UVM) which allows Test-bench modularity and reusability. The available co-emulation techniques and their advantages and drawbacks are then discussed, with particular focus on the one used in this research project: Test-bench acceleration. The first part of the thesis concludes with the state of the art of existing Test-bench acceleration implementations and the evaluation of common weaknesses. The core of the thesis focuses on the description of the developed architecture and the adopted case study: a period jitter monitor. The SW Test-bench has been written in Python according to UVM. This allows to execute it from any workstation on which Python is installed. SW-HW communication is performed using Accellera's Standard Co-Emulation Modeling Interface (SCE-MI). While the available versions of the standard are written in C/C++, for this project a Python version of it has been defined and used. The platform used to perform hardware acceleration has been Arria 10 FPGA. To evaluate the performance of the designed infrastructure, the same test scenarios have been executed both in RTL simulation and Test-bench acceleration and the execution times have been compared. For 1 million of jitter measurements, RTL simulation requires several days. The same test case, executed with Test-bench acceleration on FPGA, completes in few minutes. The speed-up factors obtained with the developed co-emulation infrastructure are between 750x and 2000x.
This thesis concludes with some suggestions on how to additionally improve the infrastructure performance and speed-up the SW-HW communication.

Keywords: co-emulation, Test-bench acceleration, FPGA, SCE-MI, UVM

Acknowledgement

A lot of gratitude goes to Professor Ernesto Sanchez, who showed interest in my Master Thesis project from the beginning and supported me during these six months. I would like to thank him for the help and feedback that he provided me.
The internship at Apple has been the most challenging and enriching experience of my life. I would like to thank the Apple community for giving me such an opportunity.
Among the people who always gave me support, I would like to thank my advisor, Yann, who constantly and wisely guided me through these months. With his enormous experience he always gave me the right advice and suggestion at the right time to put me back on track whenever I was a bit lost.
I would like to thank Philipp, who supervised me during this research project. One of the most important thing that I learned from him is that having a clear method in mind is an optimal start to face any problem. Whenever I was blocked on something, he always found the time to sit at my desk and solve it together.
I would also like to thank Helder and Yifan, who closely worked with me for the realization of this project. It does not matter which problem of which kind you are facing, Helder always has the right solution for it. Yifan gave probably the most delicate and the same time crucial contribution to the project and I would like to thank him for this.
A special thank goes also to Rajen. With his wise advice he let me step outside from a blocking point which I was facing. I will probably remember this hint for the rest of my working career.
My gratitude goes to all the members of the Design & Verification team, who welcomed me from the very first day here at Apple. I learned a lot from all of you.

Un grazie, dal profondo del mio cuore, va alla mia famiglia, unico punto fermo in questo mio percorso di studi, ma soprattutto di vita. Grazie a mio padre, che mi ha trasmesso l'entusiasmo per la conoscenza: dalla passione per la lettura all'amore per la musica, di tutti i tipi, che sia una hit dei Beatles cantata in auto o un notturno di Chopin suonato al pianoforte. Ma soprattutto, grazie per avermi sempre sostenuta in tutte le decisioni che ho preso durante questo percorso. Grazie a mia madre, che ha avuto la forza di preparare "i pacchi da giú" e le valigie per tre figlie assetate della voglia di conoscere il mondo, forse un po' troppo da non riuscire a capire il vuoto che lasciavano a casa dopo la loro partenza. Sono cose che si capiscono col tempo, con l'instancabile amore dei propri genitori.
Grazie alle mie sorelle, senza le quali probabilmente non sarei neanche qui a scrivere queste

righe. Silvia, la più grande, il primo punto di riferimento in questo percorso. La persona da chiamare quando si ha bisogno di un consiglio, la prima che ha percorso questa strada piena di ostacoli e di soddisfazioni. Maria Chiara, la sorella con la quale ho condiviso tutto, dai giochi da piccolina agli amici, dal letto nella prima casa a Torino alle prime esperienze universitarie e professionali. Ne abbiamo affrontate di prove e di cambiamenti, ma condividerli con te ha reso tutto più semplice. E non tutti hanno questa fortuna.
Tra la mia famiglia, un bacione va alla mia nonnina, che era sempre pronta ad aspettarmi a casa dopo ogni sessione d'esami. E quando le dicevo che gli esami erano andati bene, quanto mi faceva tenerezza sentirmi dire che era perch "io ho fatto una preghierina per te".
In questi anni di studio tanti sono stati gli spostamenti e le persone che ho incontrato. Ma non importava dove io fossi, la mia Family torinese era sempre con me, pronta a support-armi. Agnese, grazie perché sei sempre pronta a strappare una risata (e sappiamo quando ce ne sia bisogno quando si studia ingegneria!) e per essere stata la migliore coinquilina che si potesse avere. Alessandro, grazie per essere semplicemente contro corrente come sei: sfido chiunque ad avere un amico che é pronto a portare sotto braccio una scala in Corso Duca all'1 di notte solo per cambiarti la lampadina della camera.
Un grazie di cuore va anche alla mia amica Giusi. Non ricordo neanche piú da quanti anni mi sopporti. Sei una delle prima persone con le quali sono felice di condividere le mie piccole vittorie e lamentarmi delle mie sconfitte. E ne sono successe di cose da quel famoso primo giorno di scuola!
Ed ora il turno dei mei fantastici Nardi's Angels. Cosa posso dire ragazzi? Un giorno senza le vostre perle di saggezza nel nostro gruppo é un giorno perso. Alb, tu sei il mio indiscusso life coach. Fiore, tutti dovrebbero avere il piacere di ascoltare i tuoi monologhi sulla vita. Nico, la tua energia e il tuo entusiasmo sono semplicemente invidiabili. Grazie per avere reso questi anni letteralmente indimenticabili. Alessio, da te invece aspetto ancora la famosa spadellata!
Grazie alla spensierata e insostituibile Sede Creek che mi fa immediatamente sentire a casa ogni volta che ritorno a Soleto.
Un grazie dal profondo del mio cuore va ad Alessandro, la mia roccia in questi ultimi anni di continui spostamenti. Non é stato semplice assecondarmi e starmi dietro, ma tu sei riuscito ad essere ancora piú forte di quanto io pensassi di essere. Hai sempre creduto in me, spingendomi a dare sempre il meglio e a non sottovalutarmi mai. Grazie per essere stato tutto ció di cui avevo bisogno senza che io te lo chiedessi.
Non riesco a contare le persone che ho avuto il piacere di incontrare durante questo percorso, da Torino a Stoccolma a Monaco. Sono convinta che tutte abbiano lasciato un'impronta nella mia vita ed un insostituibile contributo. Di ció sono infinitamente grata.

Contents

List of Figures

24 24

List of Tables

List of Abbreviations

A
API *Application Programming Interface*
AVM *Advanced Verification Methodology*

B
BFM *Bus Function Model*

C
CDC *Clock Domain Crossing*
CPU *Central Processing Unit*
CRC *Cyclic Redundancy Check*

D
DPI *Direct Programming Interface*
DUT *Design Under Test*

E
EDA *Electronic Design Automation*
eRM *e Reuse Methodology*
ES *Embedded Systems*

F
FPGA *Field Programmable Gate Array*
FSM *Finite State Machine*

H
HDL *Hardware Description Language*
HVL *Hardware Verification Language*
HW *Hardware*

I
IC *Integrated Circuit*
ICE *In-Circuit Emulation*
IEEE *Institute of Electrical and Electronics Engineers*
IP *Intellectual Property*

L
LFSR *Linear Feedback Shift Register*

LUT *Lookup Table*

M

MAC *Media Access Control*
MDI *Medium Dependent Interface*
MTBF *Mean Time Between Failures*

O

OS *Operating System*
OSI *Open Systems Interconnection*
OVM *Open Verification Methodology*

P

PCI *Peripheral Component Interconnect*
PCS *Physical Coding Sublayer*
PLI *Programming Language Interface*
PMA *Physical Medium Attachment*

R

RAL *Register Abstraction Layer*
RTL *Register Transfer Level*
RVM *Reference Verification Methodology*

S

SCE-MI *Standard Co-Emulation Modeling Interface*
SoC *System on Chip*
SW *Software*

T

TB *Test-Bench*
TSE *Triple Speed Ethernet*

U

UDP *User Datagram Protocol*
URM *Universal Reuse Methodology*
UVM *Universal Verification Methodology*

V

VMM *Verification Methodology Manual*

Chapter 1

Introduction

Contents

1.1 The need for verification

The impact of integrated circuits (ICs) in our lives is enormous. They are the main components of almost all electronic devices. They are fabricated in mass production and delivered for the most disparate applications.
During the production cycle of an integrated circuit, major attention is paid to the verification phase. When the IC is produced, it must work and be reliable. To ensure that, verification procedure must be adopted during the entire flow from the design specification to the actual IC's fabrication.
One of the main rules in ICs production is that the sooner a bug is found, the cheaper it is to fix it. Bob Gottlieb [Got06] analyzed the relationship between the time at which a failure is found and the related economic impact that it has. This is typically referred as "the rule of ten": at each step of the design flow, the cost to fix the bug increases by a factor of 10. As it is possible to see from Figure 1.1 (source: [Fos10]), finding a bug in the initial design phase is 10000 times (or even more) cheaper than finding it after the first silicon phase.

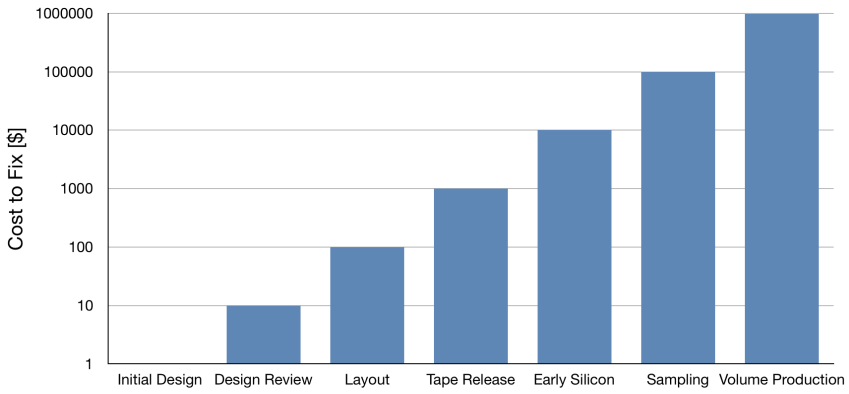


Figure 1.1: Relative cost of finding bugs.

Finding bugs as soon as possible is, indeed, critical for industries.

1.2 Design Flow and Pre-Silicon Verification

During the development of a digital or mixed signals IP, the design goes through multiple steps: from the original specifications to the final product. Each of these steps corresponds to a different description of the system, which incrementally has more details. A high level description of the typical design flow is shown in Figure 1.2.
In the industrial world some portions of the flow may be iterated several times. Moreover, some of them may be developed in parallel. This is indeed the case for design and verification phases.

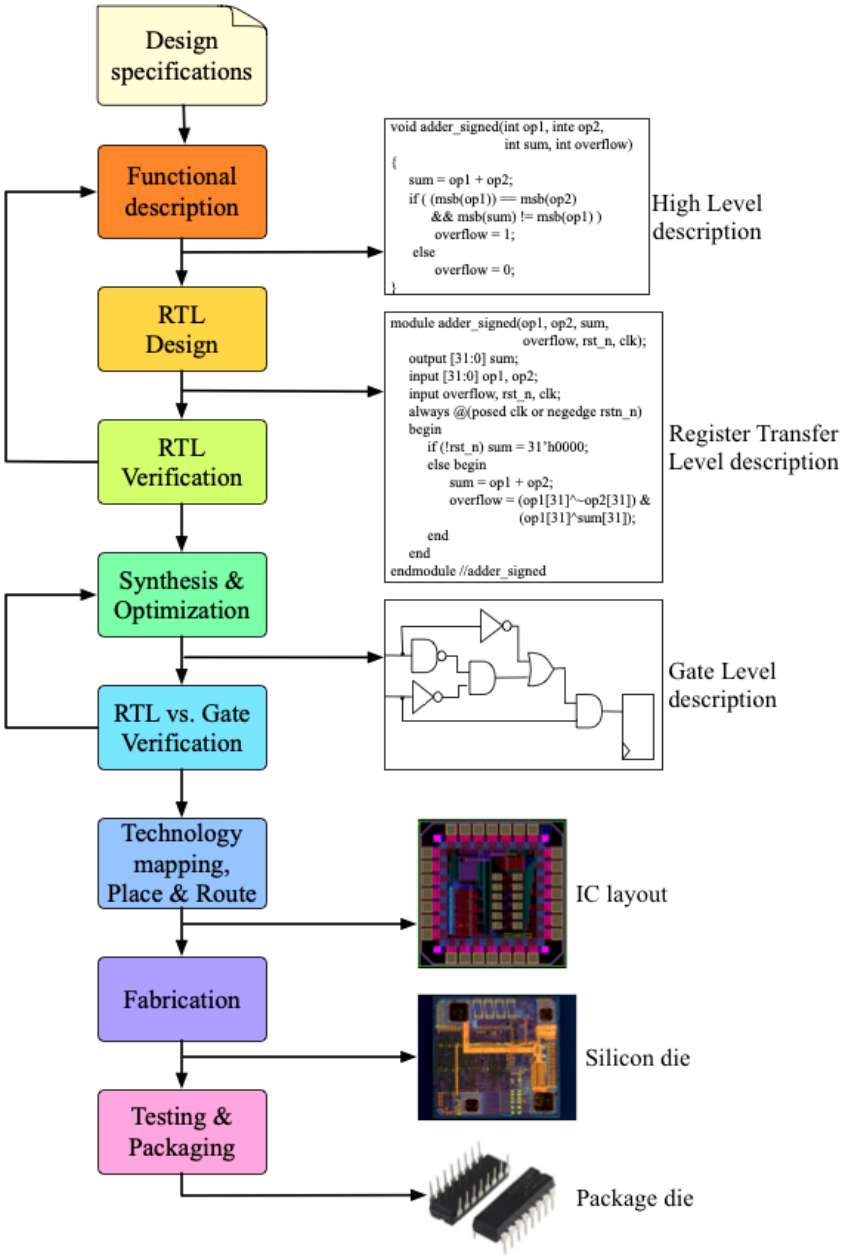


Figure 1.2: Design flow

Design specifications generally correspond to a description of constraints and functionalities that the device must satisfy and provide.
The functional description provides a high level model which can potentially include hardware/software tradeoffs and architectural features. Some of the design characteristics specified at this time of the flow are interfaces and communication protocols within internal blocks or with other external components.
The next phase is represented by the Register Transfer Level (RTL) design. The architectural description is defined using a Hardware Description Language (HDL). The clocking system is defined and attention is paid to timing and power analysis.
Once the initial version of the RTL design is completed, the RTL verification begins. This phase consists in stressing the design so that its functionality is proven in every possible (or at least almost every possible) scenario. Every time a design error is found, it must

be corrected by updating the RTL design. While in Figure 1.2 RTL verification appears as an isolated phase, it actually works in parallel with design.
Verification is generally carried out at two levels: partition and chip level. The former can be either a module, an IP, or a collection of IPs. Each partition is verified independently, with tests that include the production of input patterns to stimulate the module and the verification of the obtained output with respect to what is expected. The latter involves the verification of the entire design as a whole. The goal is to verify the proper interaction among the modules. This can be done only after a fair confidence about the correctness of each single module.
The next step is to perform synthesis optimisation. At this point, a gate level netlist is generated from the design and timing, power or area constraints are applied to meet the specifications. A new verification phase (RTL vs. gate level) is needed to evaluate if the applied optimisations introduced functional errors.
If the outcome of the gate level verification is positive, it is possible to proceed to technology mapping, placement and routing. This phase will allow to obtain a geometrical description of the layout which is used during fabrication.
Once the silicon die has been produced it is tested and packaged.
The design flow of an electronic device can be divided in two big phases: pre silicon and post silicon. During the former, the verification phase occupies a large portion of time, as shown in Figure 1.3.

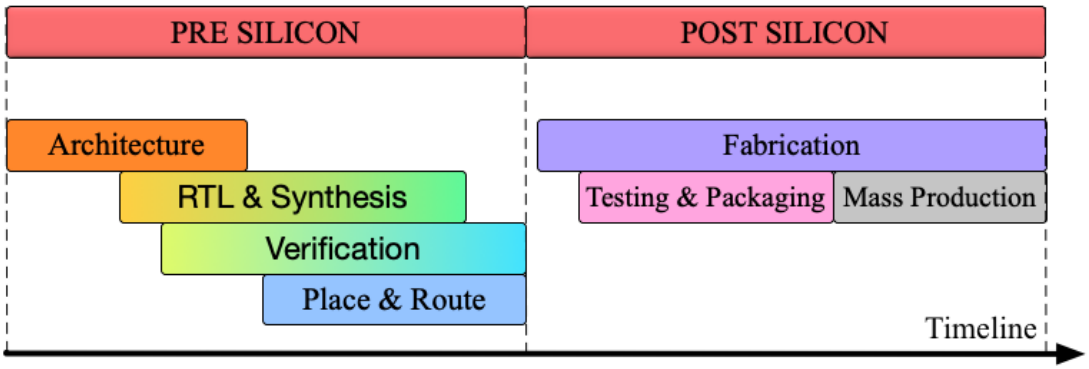


Figure 1.3: Design flow timeline

RTL verification is a crucial phase in electronic system development. The market is imposing always shorter production times with respect to device complexity. Then, in order to keep up, verification must be carried out with a valid methodology which encourages the reuse and the adoption of common practices.

1.3 Problem statement

Over the past years the complexity of circuits has increased much faster than what CAD tools can handle. The Test-benches used to verify them are also becoming more and more complex. EDA tools are, instead, lagging behind this progress. While the complexity of designs and tests increases, the performance of available simulators is lagging behind (Figure 1.4). This is referred as *verification gap* [AV19].

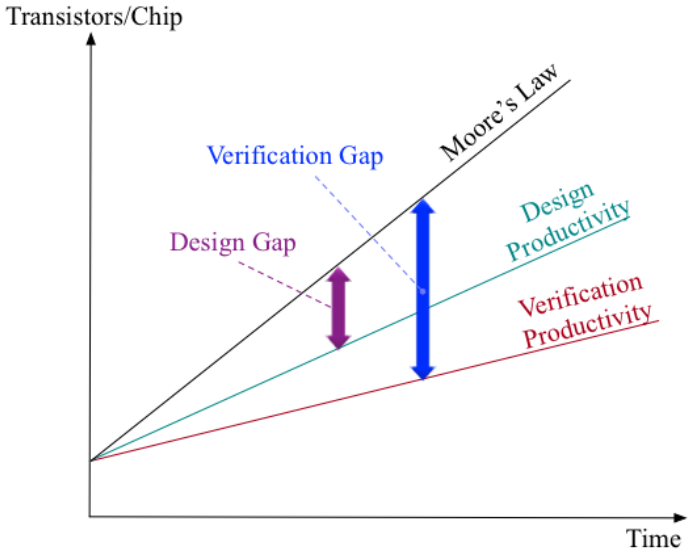


Figure 1.4: Verification Gap

Running the simulation of complex IPs can require several hours or even days, due to the fact that SW can run at a contained frequency speed. Timing can be crucial also when dealing with small IPs on which a high number of register accesses needs to be performed. RTL verification is carried out by means of RTL simulators. A wide variety of them is available on the market, provided by different vendors. They are easy to use, cost effective, and have sophisticated debugging capabilities. However, their execution speed is contained.
Let's consider that it is required the verification of a design which has a size of hundreds of millions of gates. Not only the DUT itself is very complex, but also the test case executed for its verification should perform a very high number of operations in order to reach a certain coverage. Executing this on a RTL simulation could even be unfeasible.
The main problem lies in the fact that RTL simulators can run at a frequency of few kHz. While software has this limited speed, hardware can run at a higher frequency. In this regard, co-emulation techniques have been developed over the past years [vdSS17].
Among the diffused co-emulation methodologies, one of them is called Test-bench acceleration.

1.4 Co-emulation goals

The problem related to reduced speed of RTL simulators can be faced by executing a portion of the test case on an environment running at higher speed. This is the main intent of co-emulation. This methodology is based on the co-operation of the software running on the workstation and a hardware domain executing on an emulator. The former executes the software Test-bench, while the latter is a physical board onto which some tasks are loaded and executed at a higher speed.
The main goal of adopting co-emulation over RTL simulation would be to achieve a speed-up of few orders of magnitude. Figure 1.5 shows the time shortening in both Testbench and Design domains obtained by adopting co-emulation.

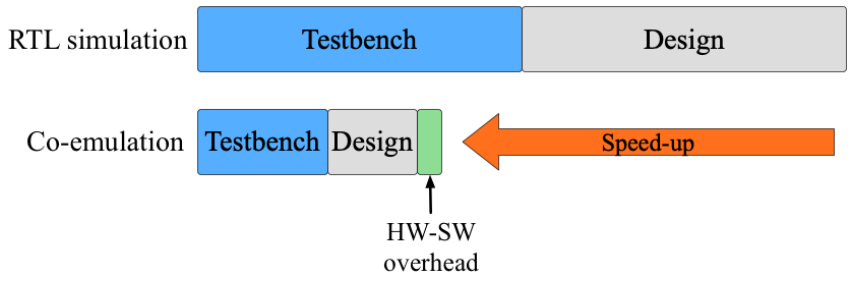


Figure 1.5: Performance speed-up from RTL simulation to co-emulation

In particular, adopting a co-emulation methodology called Test-bench acceleration, a speed-up factor between 50x and 1000x (or even more) [vdSY15] could be obtained. This would allow to run test scenarios in a much shorter time. Moreover, it will also allow to run some test cases which execution was unfeasible in RTL simulation.
It is available a wide variety of papers regarding co-emulation implementations [HKPS05, TJY14, WDBP11]. The obtained performance improvements are very different.
The achievable speed-up factor from RTL simulation to co-emulation relies on three main factors:

- adopted emulator and maximum HW speed
- HW-SW partitioning
- HW-SW communication

Examples of emulators are Field Programmable Gate Arrays (FPGAs) or complex emulation platform. The hardware speed is typically higher than the software one, but it strongly varies according to the HW platform itself [Riz14].
This research project aims at providing an implementation of the Test-bench acceleration co-emulation methodology. The adopted use case is the verification of a period jitter monitor. The main idea is to port the DUT and some of the verification tasks of the SW Test-bench onto FPGA. This will unload the software tool of some time-consuming operations, letting them execute on reprogrammable hardware.
Moving the RTL code from the SW environment onto the programmable board will also allow to move a step forward from pre-silicon verification and test the design on real hardware.
The aimed result of this research project is to achieve a speed-up factor higher than the results presented by the past implementations. To reach this, the definition of the infrastructure and the SW-HW flow control play key roles. The architecture, on both software and hardware domains, must be DUT-independent. This will allow to use it for the verification of any IP.

Chapter 2

Background

Contents

2.1 Verification with UVM

The increasing complexity of the IPs over the past years inevitably led to structure more and more complex Test-benches for their verification. Adopting the known verification techniques without a good methodology, conducted in most of the cases to write very complex but inefficient and unreusable Test-benches. For this purpose, the Universal Verification Methodology (UVM) has been defined: it allows to realize complex Test-benches in a faster, more efficient and scalable way.
UVM has been derived from other existing methodologies. Figure 2.1 (source: [Dou15]) shows its family tree and the vendors from which they have been originated.

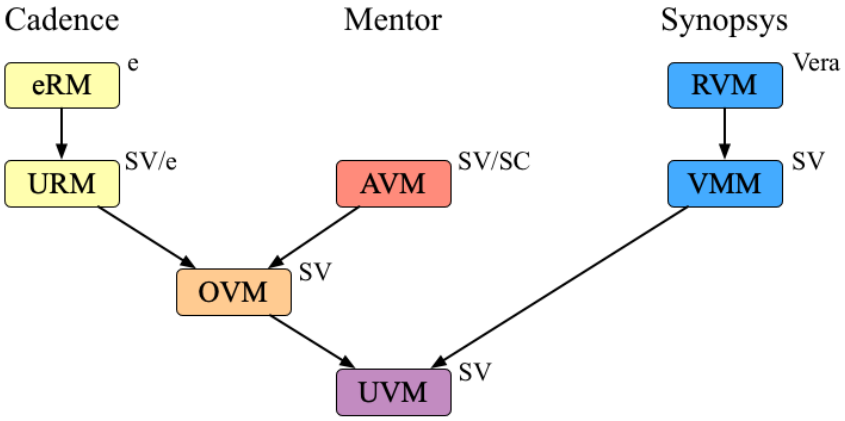


Figure 2.1: UVM family tree.

It has been directly derived from OVM (Open Verification Methodology) [Ayn] and VMM (Verification Methodology Manual), derived in turn by Vera's RVM (Reference Verification Methodology). OVM has been, instead, originated from Cadence's URM (Universal Reuse Methodology) and Mentor's AVM (Advanced Verification Methodology). From 2017, UVM is a standard approved by IEEE as a methodology that can improve interoperability and facilitate the reuse of verification components.
The main benefits of UVM are:

- providing standard Test-bench structure
- coming with a methodology
- splitting two problems: defining test scenarios and implementing pin-level protocols

The building blocks needed to develop a well-constructed and reusable Test-bench in SystemVerilog are provided by the UVM Class Library [Acc11a]. Its structure is shown in Figure 2.2 (source: [Acc11b]).

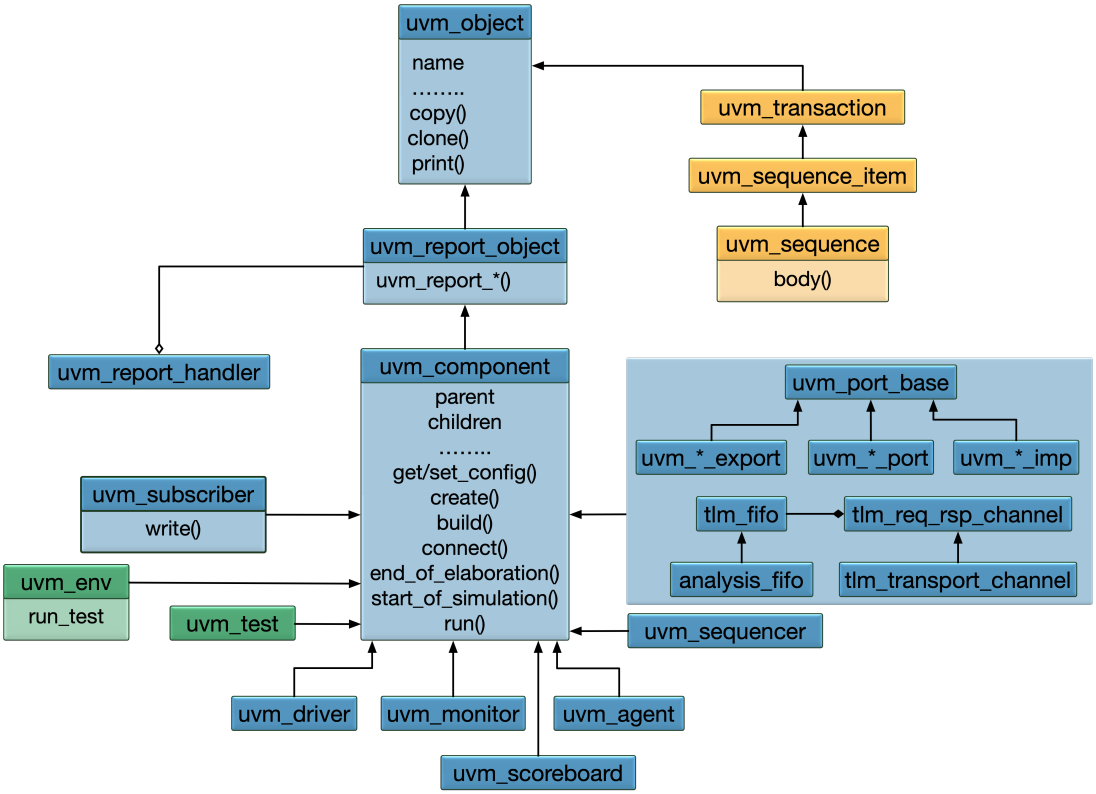


Figure 2.2: UVM Class Library

2.2 Test-bench structure

The typical architecture of a UVM Test-bench is shown in Figure 2.3.

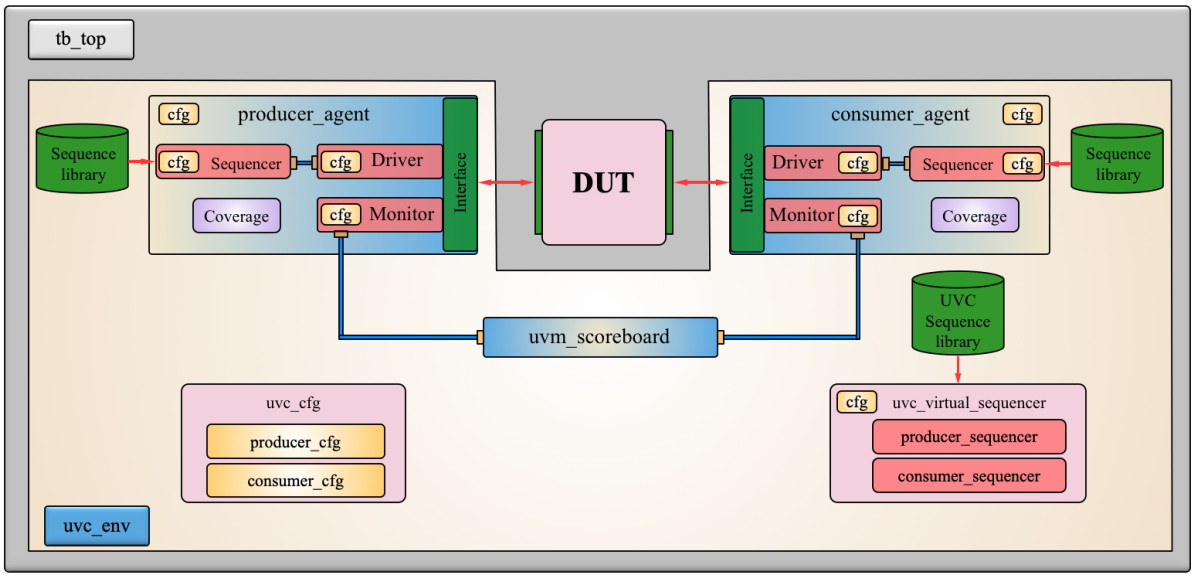


Figure 2.3: UVM Test-bench structure

Crossing its hierarchy from the top, it is composed of a Top-level wrapper (*tb_top*) that connects the test-bench and the DUT. The test-bench environment (*uvc_env*) encap-

sulates the needed UVM components.
Inside the environment, multiple *agents* can be instantiated according to the DUT interfaces. An agent can be active (containing a sequencer, a driver and a monitor) or passive (containing only a monitor that samples DUT signals).
To interface the DUT, proper sequences of stimuli need to be generated. *Sequence items*, consisting of data fields required to generate the stimuli, are randomized and inserted into *sequences*. The *sequencer* controls the flow of sequence items that need to be provided to the *driver*. This, in turn, converts the sequence items into pin level stimuli towards the DUT. At the same time, the *monitor* records the signals from the DUT. According to the interface protocol, it converts the signal changes back into sequence items.
The *scoreboard* is then responsible for checking the functionality of the DUT by comparing the DUT's outputs with the expected values.

2.3 Co-emulation methodologies

When adopting co-emulation, the verification engineer can choose among different methodologies. They mainly differ in the used hardware emulator, the SW-HW partitioning and the communication interface.
As Rizzati mentions in his article [Riz14], the two main choices for hardware-assisted verification are based on emulation platforms or on FPGA[Yas11]. Advantages and disadvantages of these two platforms are analyzed in section 2.4.
AbdElSalam and Salem explain in their paper [AS16] how different co-emulation methodologies can be implemented according to the used platform: three of the main ones are *hardware acceleration*, *in-circuit emulation* and *test-bench acceleration*. Table 2.1 summarizes the comparison among these three co-emulation methodologies.

Method	Platforms	Blocks on acceleration platform	Blocks on host workstation
Hardware acceleration	Emulation platforms	Test-bench + DUT CPU + OS can be present	SW debugger
In-circuit emulation	Emulation platform + FPGA (if needed)	Test-bench + Emulating circuit for Target system + DUT + Speed adapter	Software debugger
Test-bench acceleration	Emulation platform or FPGA	DUT + BFMs for Drivers and Monitors	Test-bench + SW communication layer

Table 2.1: Comparison between co-emulation methodologies

2.3.1 Hardware acceleration

In this methodology, [WBW01] an operation can be typically executed faster on an application-specific hardware than on a general purpose processor.
Hardware acceleration is a technique already largely adopted in many fields. Some of them are listed in the Table 2.2 [HWa20].

Application	Hardware accelerator
Digital signal processing	Digital signal processor (DSP)
Computer networking	Network on a chip (NoC)
Computer graphics	Graphics processing unit (GPU)
General computation tasks	Field programmable gate arrays (FPGA), Application-specific integrated circuits (ASIC), Systems-on-Chip (SoC)
Verification tasks	Emulation platforms

Table 2.2: Applications and related HW accelerators

Hardware acceleration implies that the entire system is loaded onto the hardware accelerator.

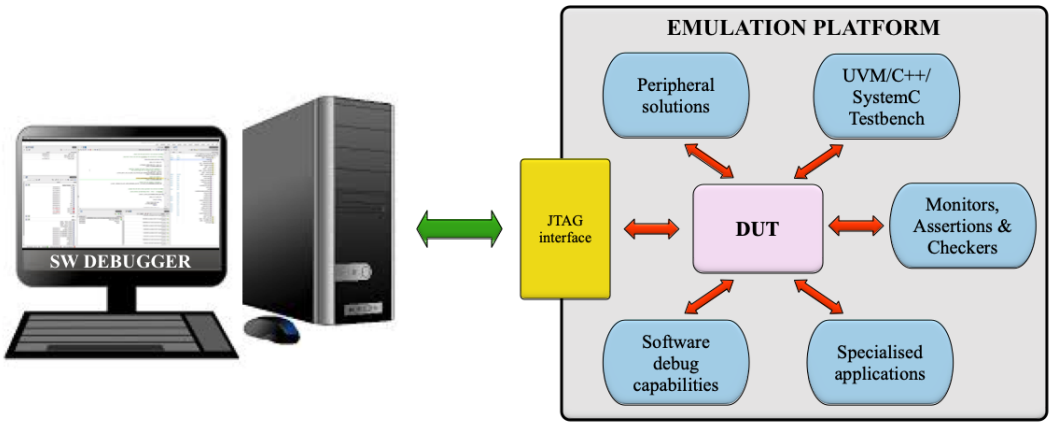


Figure 2.4: Hardware acceleration with emulation platform

2.3.2 In-circuit emulation

In-circuit emulation has been widely used for debugging embedded systems [Wal12]. It allows to connect the host workstation to the embedded system. The user can load programs through the emulator onto the target embedded system. It is also possible to set breakpoints, display memories etc. A speed adapter is required to bridge the fast side of the real-device with the low speed side of the emulated circuit [AS16].
The software program (Test-bench) is executed by the emulation platform, like in Hardware acceleration. The user tracks the verification process using a software debugger on the workstation.
If a user wants to perform digital circuit verification from the RTL description of the DUT, since the *target system* is not a physical device yet, another emulator is needed. An FPGA can then be used to map the synthesized netlist on its hardware.
Historically the platforms used for ICE were *bond-out processors.*Their name derive from the fact that their internal signals and bus were brought out to external pins for debugging purposes. Nowadays standard chips are available on the market and they are provided with a JTAG-based on-chip debugger[BB03].

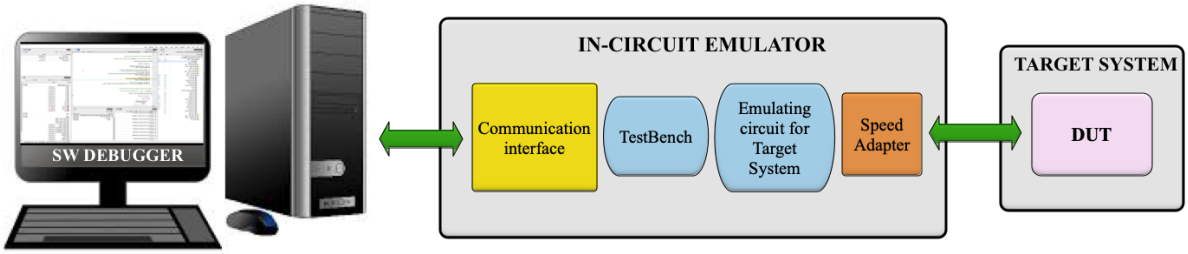


Figure 2.5: In-circuit emulation methodology

2.3.3 Test-bench acceleration

Adopting this co-emulation methodology, not only the DUT, but also the synthesizable part of the Test-bench (FSMs for Drivers and Monitors) run on a hardware emulator. They interact with a test-bench running on a host. The workstation executes the non-synthesizable portion on the Test-bench, written in a Hardware Verification Language (HVL) [vdSY15].
A framework composed of a software layer and a hardware layer is required to manage the interaction between the two domains.
When adopting this methodology, there is no request for specific and advanced interactive capabilities on the emulator side, since verification tasks such as the ones executed by sequencers and checkers are still executed in software. For this reason, any FPGA can be used instead of an expensive modern emulation platform.
One drawback of using FPGA boards for Test-bench-acceleration is that it would be possible to perform only black box verification. Indeed, it would be very complex to manage internal signal probing on the FPGA and transfer the information back to the Test-bench side on the workstation. On the contrary, this feature can be supported by emulation platforms [AS16].

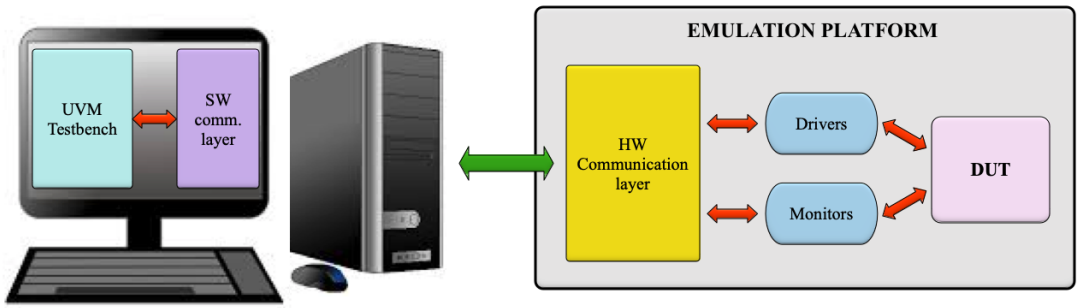


Figure 2.6: Test-bench acceleration methodology

2.4 Platforms for co-emulation

Two of the main platforms available today are emulation platforms and Field Programmable Gate Arrays. About the former ones, their performance and required tools are highly dependent on their vendor and they are typically used only at industrial level,

due to their high cost. The latter ones are widely diffused and switching from one board to another, provided by the same vendor, is almost immediate.
Table 2.3 summarizes the comparison [AS16, Riz14] among standard RTL simulation and co-emulation with emulation platforms and FPGAs.

Method	Design capacity	HW frequency	SW frequency	Bring-up effort	Debug capabilities	Cost
RTL Simulation	∼100M gates	/	∼10kHz	low	high	low
Emulator-based co-emulation	∼10-15B gates	∼2.5MHz	/	medium	medium-high	high
FPGA-based co-emulation	∼100M gates	∼100MHz	∼10kHz	high	medium-low	medium-low

Table 2.3: Comparison between RTL simulation and co-emulation methodologies

2.4.1 Emulation platform

Emulation platforms are widely used for their large capacity: they can support very large designs, such as entire System-on-Chips (SoC).
An emulation platform combines hardware architecture, operating system, specialized applications and peripheral solutions to deliver a comprehensive verification environment. They provide good debug visibility: users can download triggers, set breakpoints, advance clock signals, etc. They also provide simulation use models for assertion and coverage based verification.
The DUT is compiled and converted into a gate level netlist which is mapped onto the platform. The Test-bench, if not synthesizable, is handled by the microprocessor and Operating System internally available on the emulation platform. Specialized applications are provided by the vendor to interact with the emulation platform OS.
The maximum speed at which an emulation platform can run is around a few megahertz, which is still a good improvement with respect to RTL simulation. Having a simulator that runs at a frequency around 10kHz, the overall speed-up can be a few orders of magnitude. This is typically less then what other platforms (like FPGAs) can provide, since they can run at a frequency of hundreds of megahertz.
Their cost can vary according to the platform, but is generally very high compared to the cost of FPGA boards.
Emulation platforms can deliver all the three mentioned methodologies: HW Acceleration, In-Circuit Emulation and Test-bench Acceleration.

2.4.2 FPGA

FPGA boards provide higher speed at a reduced cost with respect to a full emulation platform.
There are several families of FPGAs, manufactured by different companies (Altera, Xilinx, Atmel etc.). Despite their differences in architecture and features, they follow a common

architecture [Bis19]. Its structure is shown in Figure 2.7 (source: [ea12]).
It consists of three main parts:

- Configurable logic blocks (CLB), implementing logic functions
- Programmable interconnect: switch boxes (SB) and connection boxes (CB), implementing routing
- Programmable I/O blocks, connecting with external components

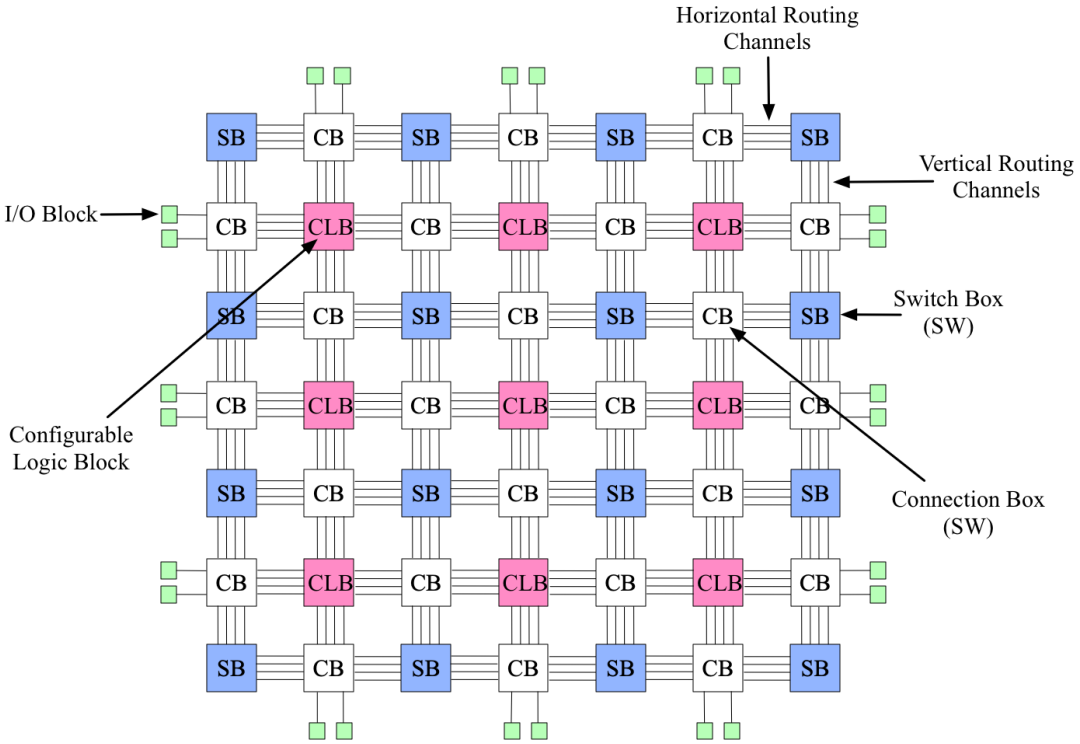


Figure 2.7: Generic architecture of FPGA

CLBs are arranged in a two-dimensional grid and are connected through programmable interconnects. The routing network is mesh-based and consists of horizontal and vertical tracks which are connected through switch boxes. Connection boxes are used to connect the routing network to the logic blocks. A high-level view of a programmable interconnect is shown in Figure 2.8.

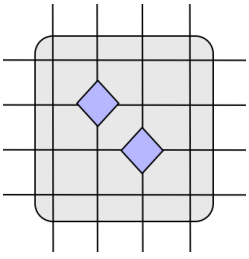


Figure 2.8: interconnect of FPGA

The design loaded onto the FPGA is decomposed in basic logic functions. Each of them is executed by a CLB, which works in parallel to all the others. According to the vendor, a CLB can be composed of different elements. The most diffused CLB structure is based on look-up tables (LUT-based). It provides a good trade-off between too fine-graned and too-coarse graned logic blocks.
A CLB can be composed of a single Basic Logic Element (BLE) or of a cluster of them. A BLE consists of a look-up table (LUT), a D Flip-Flop and multiplexers, as it is shown in Figure 2.9 (Source: [ea12]). A LUT with k inputs contains 2^k configurations and can implement any k-input logic function.

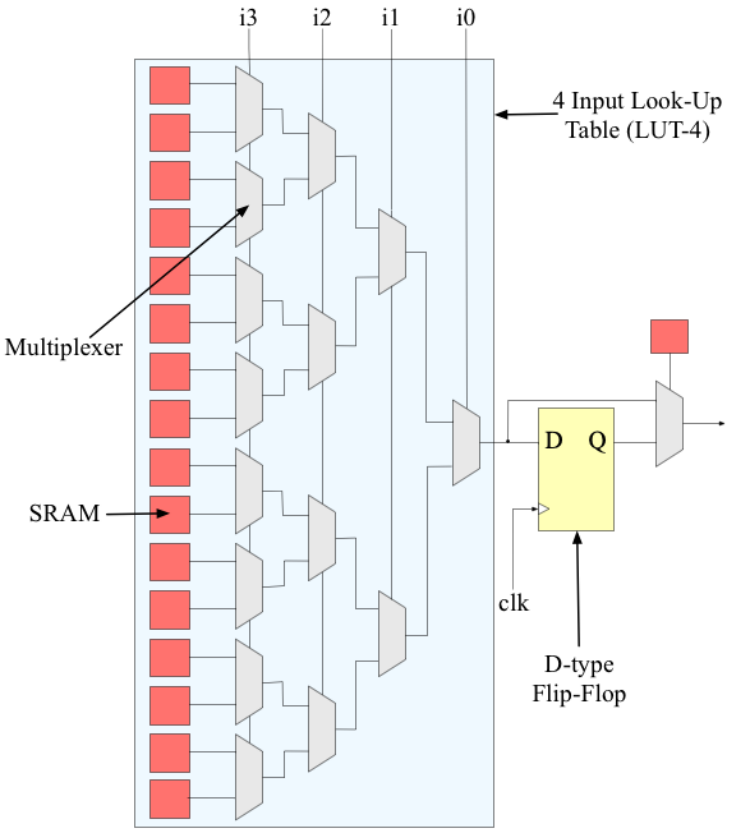


Figure 2.9: Basic Logic Element (BLE) of FPGA

The routing network occupies 80-90% of the total area, while logic elements occupy only 10-20% of it. The routing network consists of horizontal and vertical tracks connected through switch boxes (SW). Then, CLBs are connected to the routing network through connection boxes (CB).
Neighboring logic cells may be routed with high speed dedicate lines to improve performance.
I/O blocks sit at the periphery of the grid. They are used to interface the CLBs and the routing architecture with external components.

Among the three methodologies mentioned in section 2.3, FPGA is adopted for Test-bench Acceleration, which allows to run portions of the Test-bench at frequencies in the order of hundreds of MHz, leading to a speed-up of a few orders of magnitude with respect to RTL simulation.

In RTL simulation the Test-bench directly interacts with the DUT (Figure 2.10), directly implementing pin-level protocols.

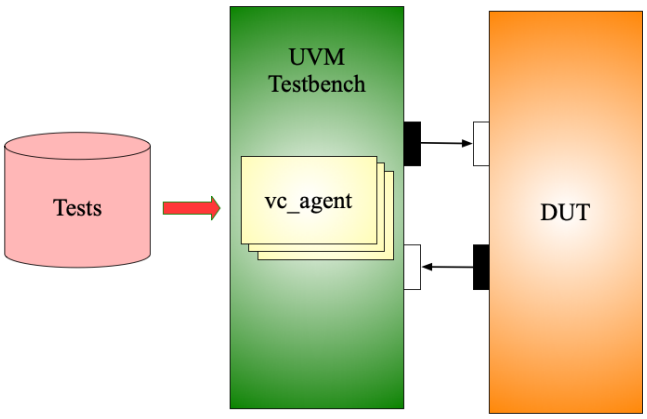


Figure 2.10: Test-bench - DUT interaction in RTL simulation

In FPGA Test-bench acceleration a communication layer between the host workstation and the FPGA is required (Figure 2.11). This introduces an overhead due to the communication between the two domains. If this additional time is not significant with respect to the time spent in HW (FPGA) and the time spent in SW (host workstation), this overhead can be leveraged.

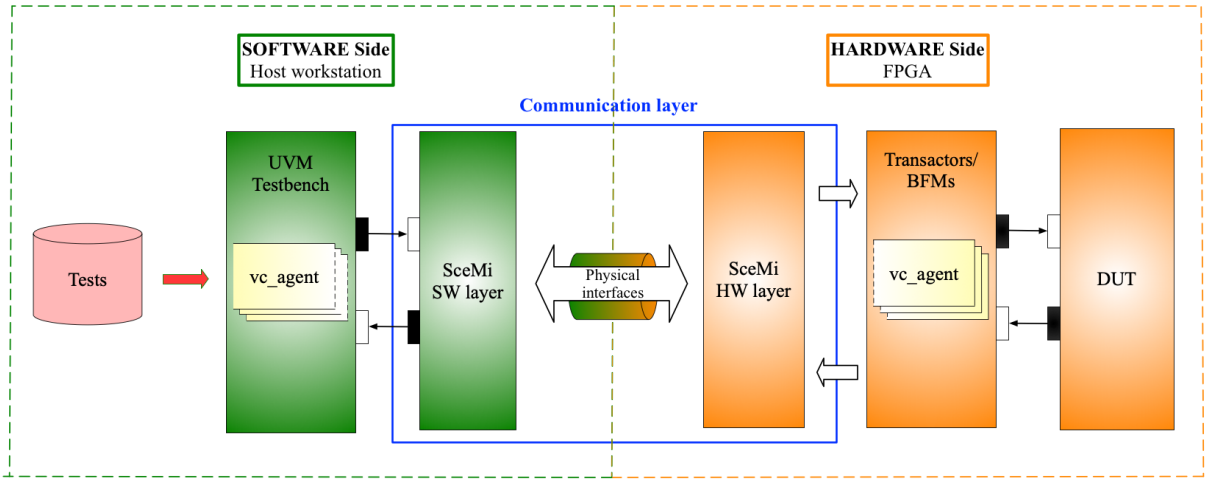


Figure 2.11: Test-bench - DUT interaction in FPGA Test-bench acceleration

One possibility to carry out the communication between SW and HW is by means of the SCE-MI interface. This standard describes the implementation of this communication interface, which is divided into a software and a hardware layer. Its bring-up requires an initial demanding effort: the infrastructure must be designed to be reusable and non-DUT dependent.
Debugging the DUT and the infrastructure on the FPGA is difficult, given the limited debug capability offered by the available logic analyzers. However, once the bring-up phase of the infrastructure is completed, it is highly reusable: only dedicated BFMs for the DUT interfaces need to be designed.

2.5 UVM Test-bench acceleration with FPGA

Co-emulation is a very useful methodology for the verification of DUTs with time-demanding RTL simulations or for the execution of particular stress test cases. However, it is not meant to entirely replace RTL simulation.
One of the key points of this co-emulation project is the need to maintain the pre-existing verification environment used for RTL simulation. Only few modifications have been done to support the co-emulation features.
The methodology adopted in this research project is *test-bench acceleration* of UVM test-benches, performed using the Altera FPGA Arria 10 as acceleration platform [Int18].
As van der Schoot and Yehia explain in their article [vdSY15], three steps are needed to create a unified dual domain framework for UVM test-bench acceleration:

- definition of dual software and hardware domains
- modeling of the timed test-bench portions in the HW domain
- definition of a transaction-based intra-domain API

2.5.1 Dual SW and HW domains

The HW domain runs on the emulator, containing the synthesizable portion of the verification framework: DUT, clock and reset generators and Finite State Machines (FSMs) to interface the DUT.
On the other side, SW domain is strictly untimed and contains the non-synthesizable behavioral test-bench code. It generates transaction-level stimuli and contains analysis components like generators, scoreboards and coverage collectors.
This domain partitioning is facilitated by the UVM abstraction and layering principles: upper test-bench layer components delegate timing control to lower layer components such as UVM drivers and monitors.
Figure 2.12 shows the layered structure of a UVM test-bench. Three sections can be identified: a *Test-bench layer*, a *Transaction layer* and a *RTL layer*. It is possible to notice that most of the UVM test-bench is naturally untimed (Test-bench layer). The timed portion of the Test-bench is implemented by Bus Function Models (BFMs) loaded onto the FPGA (section 5.5).

2.5.2 Modeling of timed HW domain

The *Transaction layer* is in charge of converting untimed transactions into cycle-accurate clocked events applied at the interface of the DUT and vice versa. As shown in Figure 2.12, Drivers and Monitors sit on both the *Test-bench layer* and the *Transaction layer*: on the former side they send/receive sequence items, while on the latter the Driver converts sequence items into pin-level stimuli and the Monitor does the opposite.
The functionalities of Drivers and Monitors related to the interaction with the DUT interface signals can be ported on the HW domain: they are implemented as Bus Function Models (BFMs).
On the UVM test-bench side, "*uvm_driver*" and "*uvm_monitor*" instances will still be present, but they will not be responsible anymore for clocked stimuli generation.

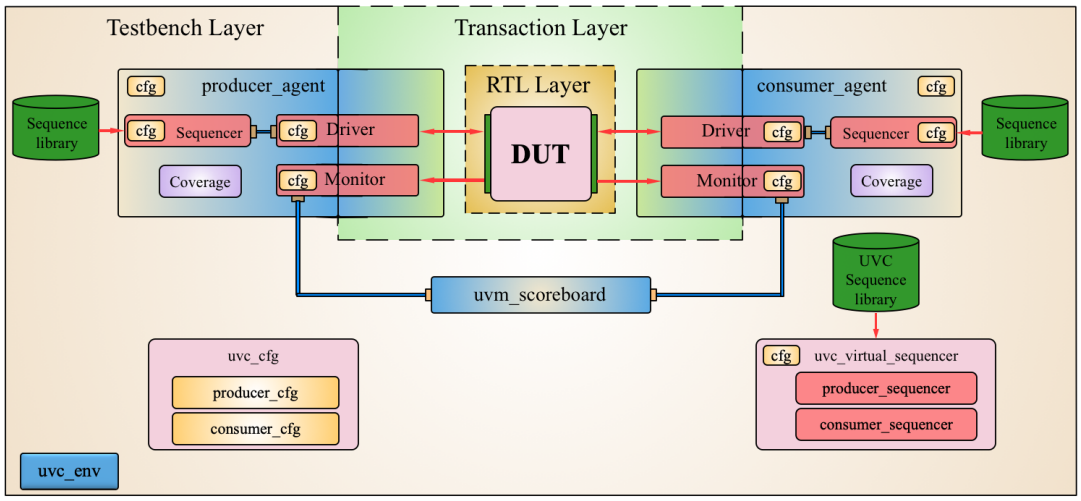


Figure 2.12: UVM layered test-bench

Moving the BFMs from SW to HW domain means that UVM test-bench objects can now access DUT signals only indirectly, by means of a DUT proxy model.

2.5.3 Definition of transaction-based SW-HW API

After partitioning the test-bench, a communication interface between the two domains needs to be defined. SW domain can interact with HW domain by following a remote proxy design pattern [vdSY15].
The communication between the two domains can be carried out by means of an intermediate layer. One way to implement the intra-domain communication is by using the Standard Co-Emulation Modeling Interface (SCE-MI)[Acc16], which provides protocols and channels that allow to connect the software to the hardware.
Figure 2.13 shows how a UVM test-bench for acceleration is structured, highlighting how the layered structure shown in Figure 2.12 is maintained.

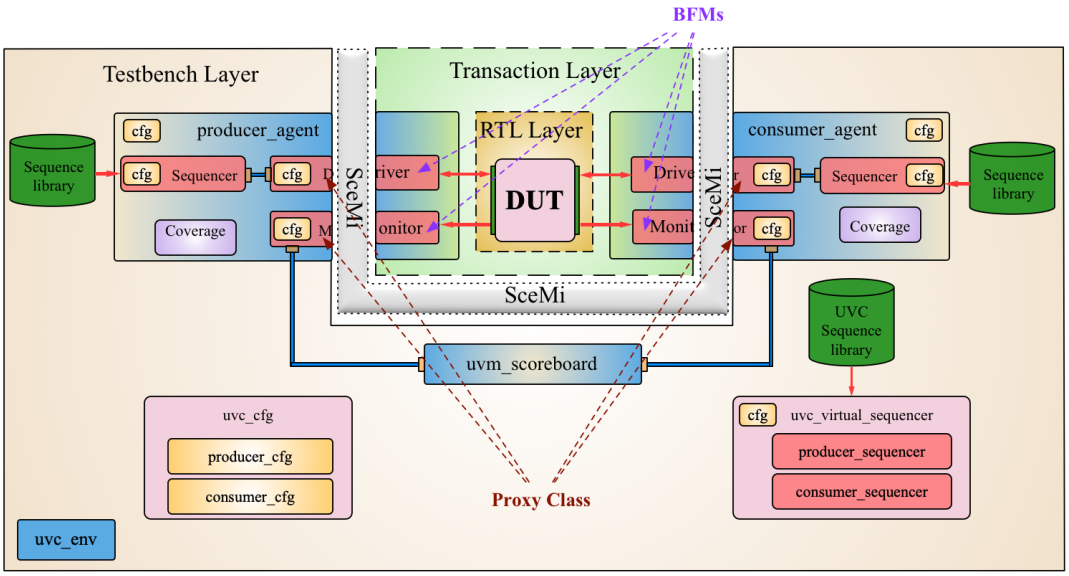


Figure 2.13: UVM test-bench for acceleration

The SCE-MI interface sits right in between the *Test-bench layer* and the *Transaction layer*, working as a communication channel between them.

2.6 Standard Co-Emulation Modeling Interface

The Standard Co-Emulation Modeling Interface (SCE-MI) is an Application Programming Interface (API) developed by *Accellera*. It provides multiple channels of communication. Each channel allows software models to connect to hardware models.
The main purpose is to interface purely untimed software models with a register transfer level (RTL) or gate level DUT. Given the speed difference between the two end points, the focus of the interface is to avoid communication bottlenecks which might compromise the performance of the emulator.
The SCE-MI interface has been defined in three versions:

- Message-passing Macro-based interface
- Function-based interface
- Pipes-based interface

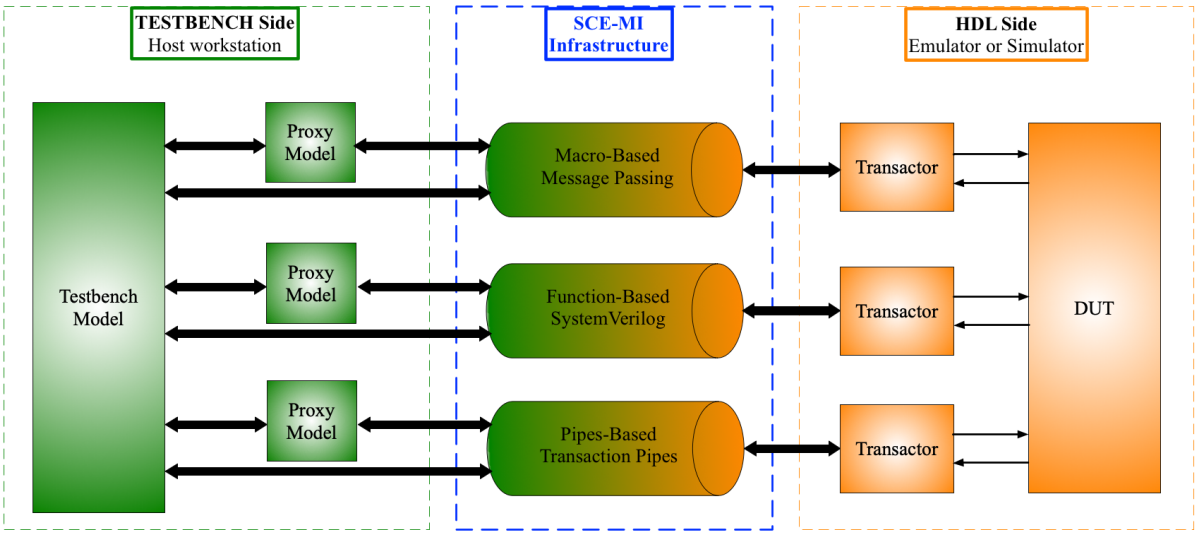


Figure 2.14: Three SCE-MI interfaces

2.6.1 Function-based SCE-MI

The *Function-based interface* [Tom] is based on SystemVerilog Direct Programming Interface (DPI), which was designed to provide an easy way to use inter-language communication mechanism based on function calls. The user can create his/her own API by defining functions in one language and calling them from the other. Although SistemVerilog provides a wide set of data types, only a small subset is compatible with the SCE-MI standard.
Function calls provide an intermediate level of abstraction which is suitable to be connected to both the emulator and host workstation domains.

2.6.2 Pipe-based SCE-MI

The *Pipe-based interface* [Acc16, PGM11] supports constructs called *transaction pipes* which are accessed through function calls and stream transactions to and from the HDL side.
To perform send and receive operation over a transaction pipe, two function calls very similar to the write and read calls to UNIX sockets have been defined. Pipelining streaming items allows to reach an optimal streaming throughput, leveraging the round trip latency issue.
The transaction pipes are unidirectional, which means that it is guaranteed that the data sent from one end of the pipe are received in the same order by the other end.

2.6.3 Macro-based SCE-MI

In the Macro-based interface the interconnects between the transactors on the emulator side and the test-bench model on the workstation are provided in the form of message channels. Each message channel has two ends: the one on the software side is called *message port proxy*, which gives API access to the channel, while the one on the hardware side is the related transactor's interface.
Message channels should be seen as network sockets which use message passing protocols. The transactor decomposes messages arriving at the input channels into sequences of cycle-accurate events and, for the other direction of flow, recomposes sequences of events coming from the DUT back into messages sent via output channels.
The SCE-MI infrastructure works as a transport layer that delivers messages between the host workstation side and the emulator side, as is it shown in Figure 2.15

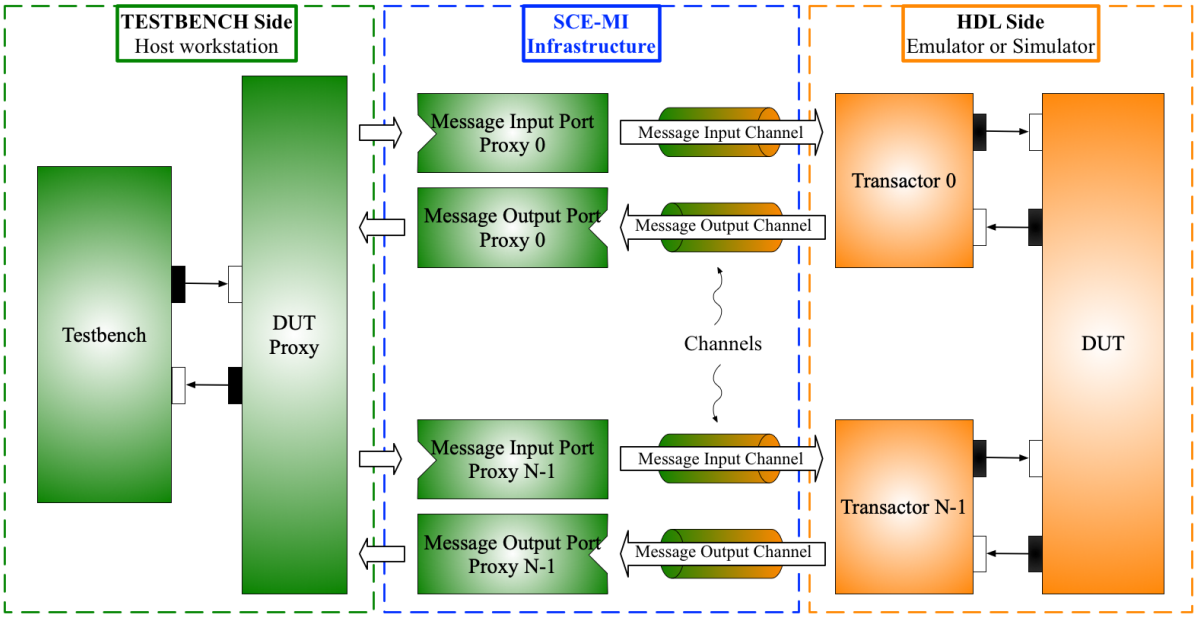


Figure 2.15: Macro-based architecture

The DUT proxy receives untimed messages from the Test-bench models. It then sends them to the proper message input port proxy. Each port is defined by a transactor and port name. It has a defined bitwidth but is data agnostic. Messages are not associated

with clocked events, but they are considered as data types that can be transported among software models. Before messages are sent over a message input channel, they are serialized into a bit vector by the DUT proxy model.
For the other flow direction, the HDL side constructs messages that are sent through the message output channel. The DUT proxy monitors the message output port proxies to check for arriving messages and de-serializes them back into abstract data types.
The work flow for system verification through SCE-MI environment can be divided in three phases:

- Software model compilation: the models that run on the workstation are compiled and linked with the software side on the SCE-MI infrastructure
- Infrastructure linkage: during this process, the macro-based SW layer reads a description of the hardware containing information of the transactors (total number, names, widths of the messages) and elaborates its structure
- Hardware model elaboration: a netlist is generated, loaded onto the emulator and prepared for binding to the software

Chapter 3

State of the art

The need for simulation speed-up has been faced since many years from companies that design complex hardware. They need to ensure good functionality and performance of their devices in the shortest possible time frame. Hans van der Schoot and John Stickley describe in their article [vdSS17] the history of this effort, from the old In-Circuit Emulation (ICE) methodology to the advanced transactor-based co-emulation techniques.
The first methodology developed to speed-up the verification process has been In-Circuit Emulation. It has been extensively used until the 1990s. It was a quite expensive technique and only the richest companies producing processors could afford it. Their processor was connected with an enormous amount of cables to big hardware emulators. Because of its structure, the engineers called it *ICE spaghetti cables*. Despite its expensive cost, it had a mean time between failures (MTBF) of a few hours. Every movement of one of the cables could have alterated the verification process.The final solution was to use ICE only at the final verification stage, to reduce the tape-out risk. An RTL simulation environment was instead used in the precedent phases.
To bridge the gap between ICE and simulation, different methodologies started to be adopted in the mid-1990s. In-circuit emulators have been virtualized into functionalities such as Programming Language Interfaces (PLIs), Application Programming Interfaces (APIs), Direct Programming Interfaces (DPIs) and so on.
Early co-simulation techniques via the Verilog PLI started to be adopted, sold by emulation vendors. However, they were very difficult to use and they were signal-oriented. Because of the inefficient communication through the signal-level interface, the simulation speed could achieve only few kilohertz. The emulator was, instead, capable of running at frequencies in the order of megahertz.
This performance degradation has been reduced by elevating the abstraction level of the test-bench, which were written in C/C++. PLI-based communication has been replaced by API-based interfaces. Figure 3.1 and Figure 3.2 (source: [vdSS17]) show the performance improvements obtained using co-simulation with Verilog and C/C++ Test-benches respectively.
However, co-simulation did not allow to exploit the full speed of the emulator, which was held back by the Test-bench. A solution to this problem has been introduced when co-simulation has been replaced by co-emulation (Figure 3.3). Test-bench acceleration methodology started to be adopted, dividing the Test-benches into two parts as described in section 2.3.3.

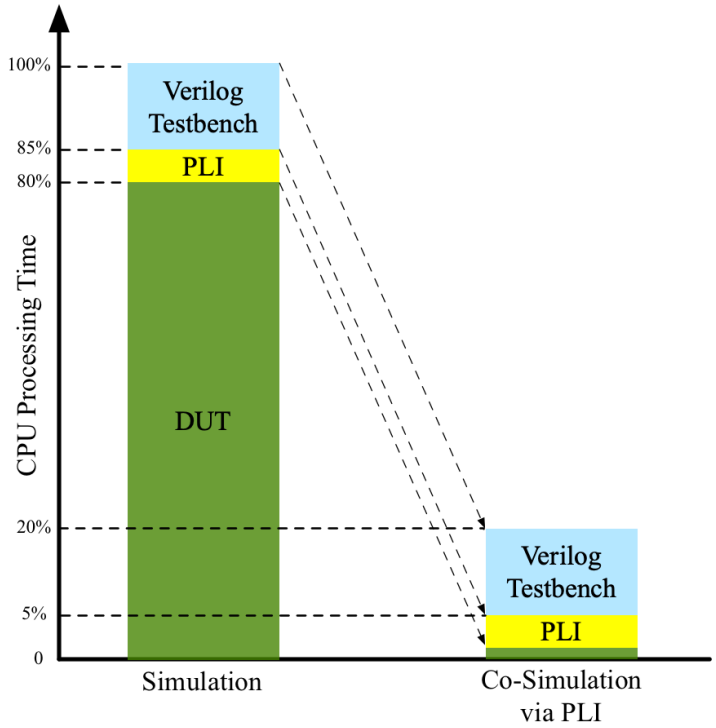


Figure 3.1: CPU Processing time in simulation and co-simulation with Verilop PLI

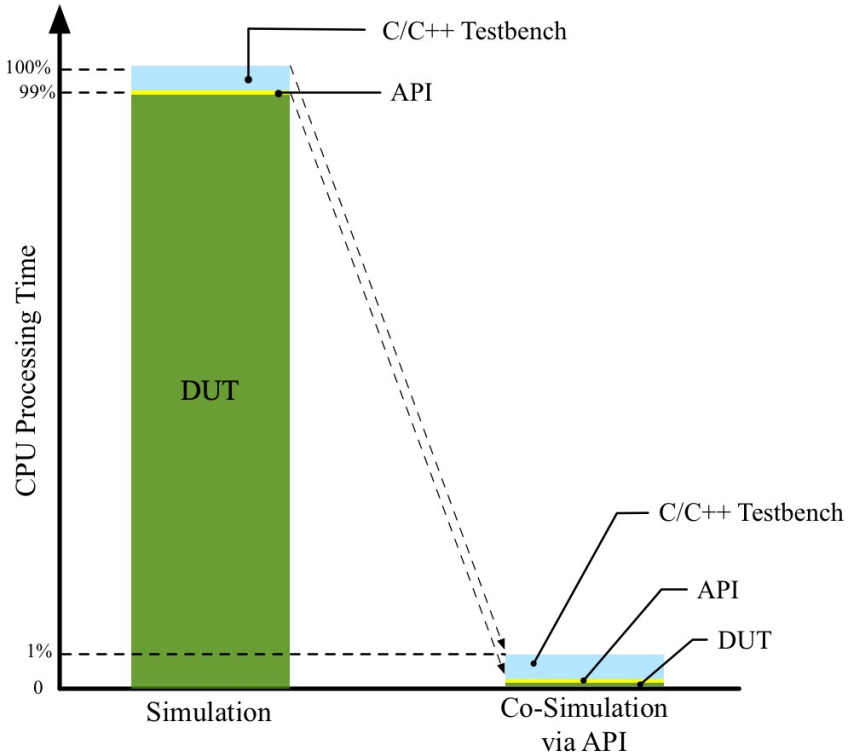


Figure 3.2: CPU Processing time in simulation and co-simulation with API

An extensive number of papers regarding Test-bench acceleration implementations have been written in the past years.

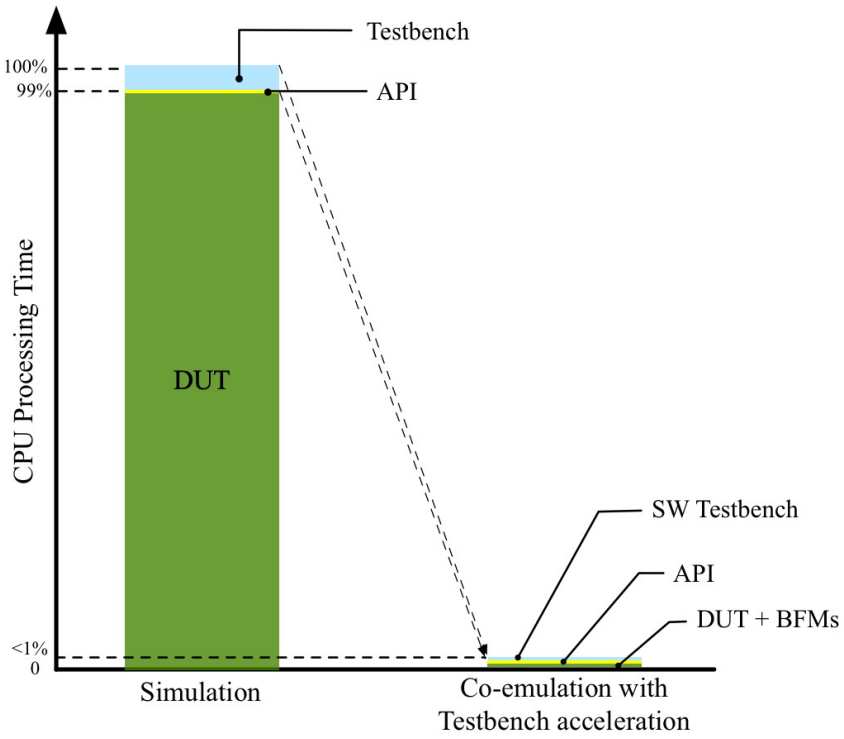


Figure 3.3: CPU Processing time in simulation and co-emulation with Test-bench acceleration

Hassoun, Kudlugi, Pryor and Selvidge give in their paper [HKPS05] an analytical computation of the speed-up ratio that could be obtained by using transaction-based instead of cycle-based communication. Having:

- L_t: latency of one transaction measured in time units
- C_{user}: clock cycles of work generated by the transaction
- L_{sc}: latency of one cycle-based instruction
- π: clock period

The speed-up ratio can be computed as

$$\text{ratio} = \frac{(C_{\text{user}} \times \pi + L_t)}{(\pi + L_{\text{sc}})} \tag{3.1}$$

Keeping L_t low and performing many cycles of work per transaction (C_{user} high) will result in good performances.
The main topic of their paper is the description of a layered architecture used for both simulation and emulation. An analysis of their results with respect to the expected ones is then provided.
They adopt the Test-bench acceleration methodology, highlighting two domains:

- Driving Environment(DE): contains the user application, the *application adapter* (which provides an API) and the drivers

- DUT environment: contains the netlist of the DUT, the transactors and *co-modeling primitives*

To perform intra-domain communication they did not use any standard, but they developed their own API with different C routines. The *application adapter* is the block responsible for providing these routines.
For simulation, DE and DUT environments are implemented as two processes and connected via UNIX-based socket.
For co-emulation, the DE is implemented on the host workstation and the DUT environment is loaded onto the emulator (Altera Flex 10 KE FPGA [Alt03]). The connection between the two is done through PCI-IB.

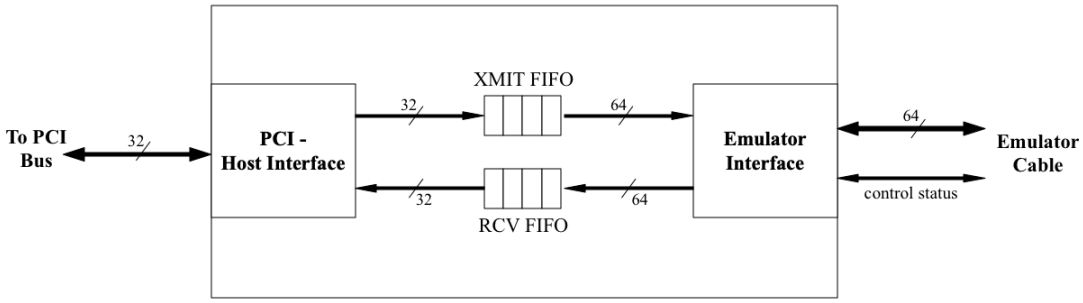


Figure 3.4: Block diagram of PCI-IB

This interface provides that data received and sent have a fixed length, as it is shown in Figure 3.4 (source: [HKPS05]). This has the disadvantage that in case of transactions with multiple bytes, several PCI transactions are also required.
On the FPGA side, the PCI interface is directly connected to input/output macros, which in turn are connected to the transactors. Such a structure does not provide any interactive flow control capability. Hardware and software will for sure run at different frequencies. The only way to be sure that no packets are lost and that the software buffer and hardware FIFOs do not get full is to always wait for a certain number of transactions to go around the full loop (from host workstation to FPGA and way back). This of course degrades the overall co-emulation performance.
The two metrics used to evaluate the performance of the emulator were:

- round-trip latency: time required to perform one transaction without the latency introduced by the FIFOs within the PCI-IB card. It can be computed as the addition of the following metrics:

  1. delay within the workstation: time spent to execute an API write or read
  2. delay transferring the data from the workstation to the PCI-IB card
  3. delay within the PCI-IB card
  4. delay transferring the data from the PCI-IB card to the emulator
  5. delay within the emulator co-modeling "*Primitives*"
  6. delay within the emulator transactors
  7. delay within the DUT

- communication bandwidth (CB): number of frames that can be inserted and removed from a full pipeline per second. The limiting CB could be one of the following two paths:

  1. between the workstation and the PCI-IB card
  2. between the PCI-IB card and the transactors

The PLI simulation time has been compared with the emulation one. The authors verified two different IPs and the obtained speed-up factors were 240x for the first one and 320x for the other one.

Tomas, Jiang and Yang discuss the difference between signal-based and transaction-based communication. In their paper [TJY14] they briefly introduce the functionalities and protocols of the SCE-MI interface (3.6.3) and then the topic of their work: SoC Scan-Chain verification with emulation platform using both signal-based and transaction-based communication.
They use the functional verification platform Aldec Riviera-PRO [Ald]. During simulation acceleration with signal-based communication, the DUT has been loaded onto the emulation platform. Connection with the host workstation is done by means of a PCIe connector. The emulator compiler automatically analyzes the RTL sources and distinguishes between synthesizable and non-synthesizable code. The former is loaded onto the emulation platform, the latter is executed on the simulator. Their design has 7 inputs and 7 outputs, which are mapped to single-ended lines on the platform. It is clear that this solution cannot be feasible for DUTs with a large number of I/O pins.
Moving to the transaction-based methodology, some modifications were required. In order to use the SCE-MI interface, the test-bench on the software side has been written in SystemC. On the hardware side, message ports and transactors are designed to convert messages into signals towards the DUT. They provide two different versions of transactors implementation. One called ”pass-through” transactors, which do not manipulate the data coming from the software. The other one is FSM-based, with transactors serially shifting in the data received from the software. The obtained speed-up factors with respect to RTL simulation were:

- 2.11x with simulation acceleration (signal-based)
- 2.43x with pass-through emulation (transaction-based)
- 5.8x with FSM emulation (transaction-based)

The implementation proposed by Tomas, Jiang and Yang [TJY14] provides good results if we consider that an emulation platform is used. The maximum frequency that it can provide is tied to few megahertz (8 MHz in case of Aldec Riviera-PRO). The authors of the paper were able to reach a speed-up of 5.8x with respect to RTL simulation. This factor can be good in case of simulations that run for minutes or even few hours. On the other hand, for test cases that require a longer simulation time, a higher spee-up factor might be needed. To reach that it is necessary to use a faster platform like an FPGA.

Another implementation of Test-bench acceleration has been developed by Wrona, Barcik and Pietrasina [WDBP11]. Their co-emulation environment was composed of a SystemC Test-bench, the SCE-MI interface and an FPGA platform. The project was intended to verify a high resolution VGA interface controller. The DUT has been loaded onto the programmable board together with the hardware SCE-MI layer and the transactors.
The USB port [Mur] was used to perform the physical connection between the host workstation and the platform. A SCE-MI interface domain was used to decode the received messages and interact with a FIFO-domain. Message ports were placed between FIFOs (containing input/output messages) and the transactors. Other blocks were used to control the DUT clock signals and manage the external memory connection.
The authors describe how software and hardware transactors are automatically generated. The SystemC description of the transactors and the message frames (XLM language [Qui11]) can be divided by a parser into a C++ transactor proxy and a Verilog synthesizable transactor description (Figure 3.5, source [WDBP11]).
They executed seven testcases and compared the overall time needed in RTL simulation and SW-HW co-emulation. The second implementation was 70 times faster than the first one.

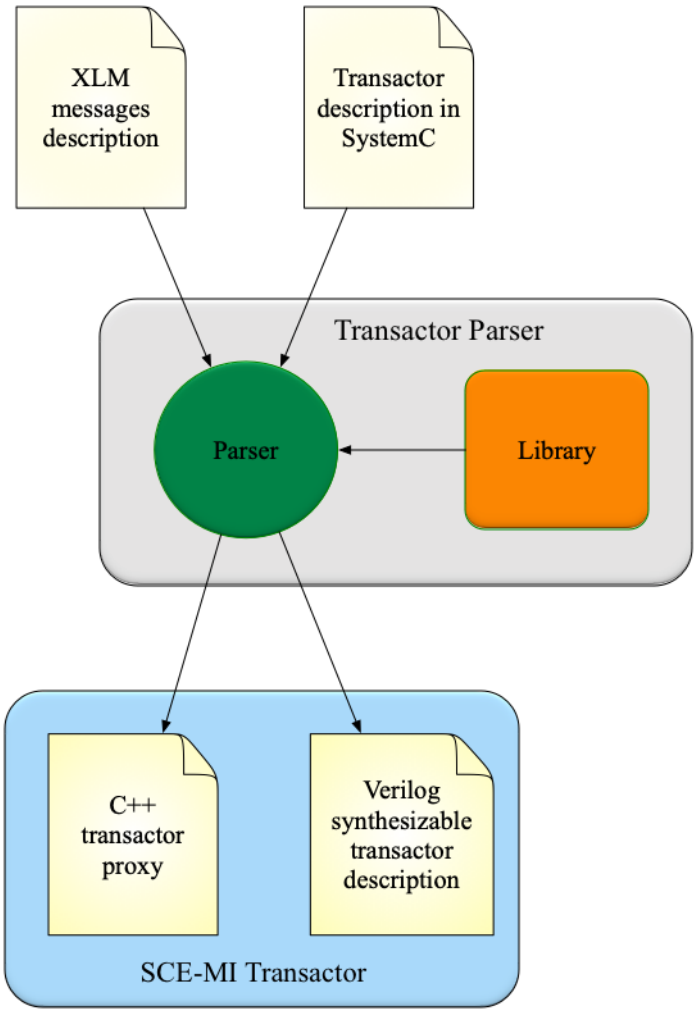


Figure 3.5: Transactor designing process

One common weakness of the implementations presented in the mentioned papers [HKPS05, TJY14, WDBP11] is that the authors do not discuss about SW-HW intra-domain flow control. This could be due to the fact that the SCE-MI standard [Acc16] provides that whenever a transaction is sent from the SW to the HW, a reply should be generated by the hardware and sent back to the software side. At each transaction, a handshake mechanism is established between the two side. This approach slows down the overall architecture, since the fast side should anyway wait for the slow one.
An alternative approach would be to implement a flow control technique that allows to continuously stream messages on both direction, preventing that software buffers or hardware FIFOs get full.
Moreover, as already mentioned before, in order to achieve a higher speed-up, FPGA boards should be chosen over emulation platforms. This can be noticed by looking at Figure 3.6. It contains a comparison between the results obtained by Wrona, Barcik and Pietrasina [WDBP11] (70x) and Hassoun, Kudlugi, Pryor and Selvidge [HKPS05] (240x, 320x) with the ones obtained by Tomas, Jiang and Yang [TJY14] (5.8x).

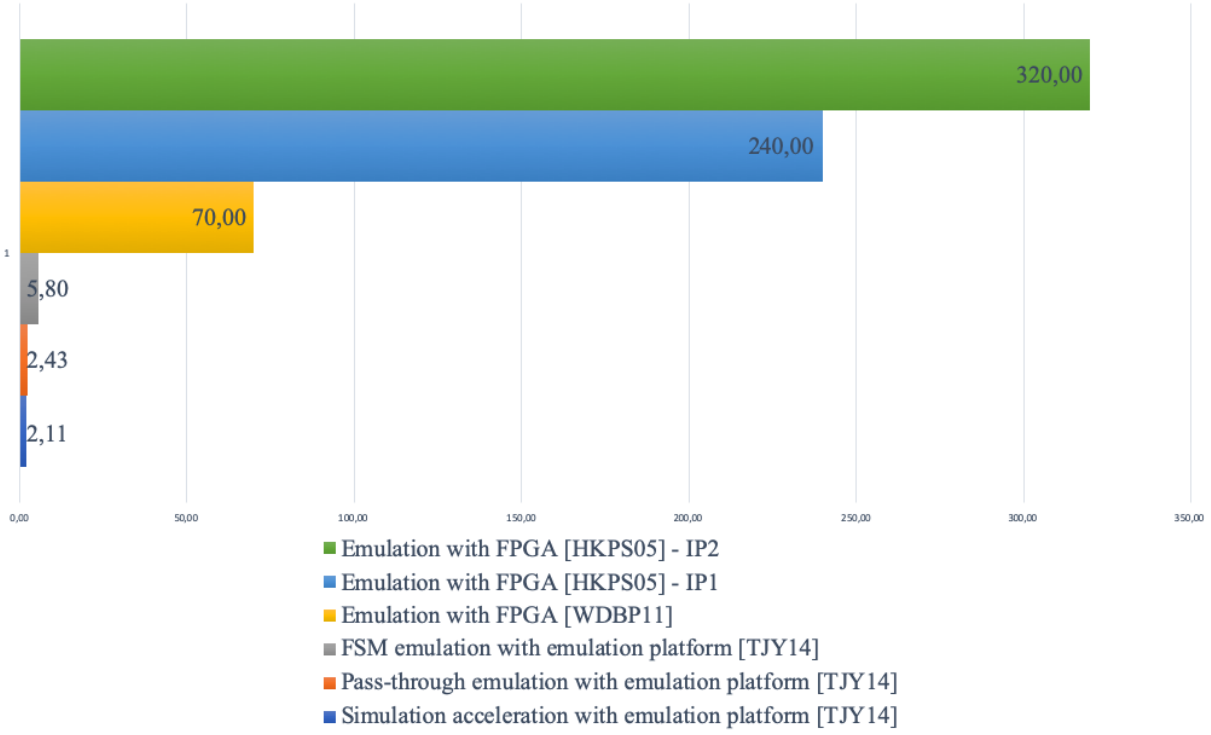


Figure 3.6: Speed-up factors obtained with discussed implementations

Chapter 4

Case study

The DUT which has been verified with Test-bench acceleration is a Period jitter monitor. A high level description of the architecture of the DUT is shown in Figure 4.1. It consists of a measurement block configured by a control logic.

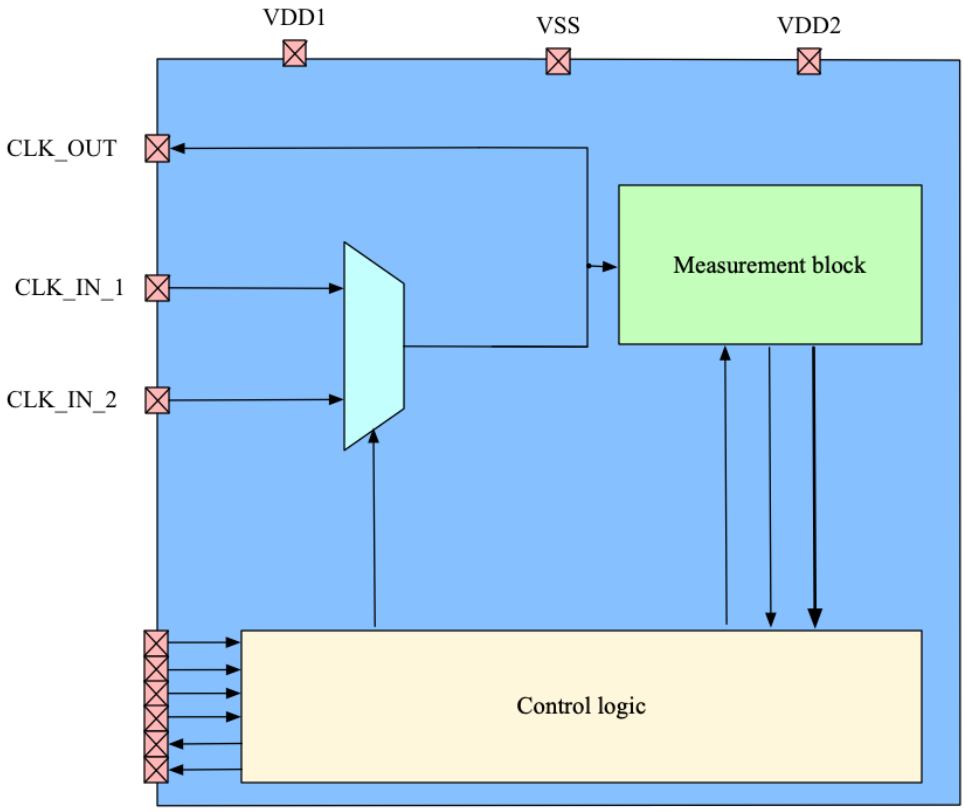


Figure 4.1: DUT architecture

The input clock can be chosen between two sources: an external source (*CLK_IN_1*) and an internal one (*CLK_IN_2*). A multiplexer selects one of the two paths and provides the output clock to the measurement block. The input clock can be observed externally through *CLK_OUT*.
The input clock jitter can be analyzed by collecting many measurements. The root-mean-square (RMS) random period jitter can be calculated from the collected values. The higher the number of samples, the more accurate is the calculation of the RMS.

Chapter 5

Co-emulation architecture

Contents

5.1 General SW-HW architecture

The general SW-HW architecture developed to perform Test-bench acceleration is based on the Standard Co-Emulation Modeling Interface [Acc16]. The implementation used is the Macro-based message passing interface. As already mentioned in section 2.6.3, it allows to connect transactor models on the emulator (hardware) side to untimed models on the workstation (software) side. A high level view of the architecture is provided in Figure 5.1. The physical connection between software and hardware domains has been performed using Ethernet.

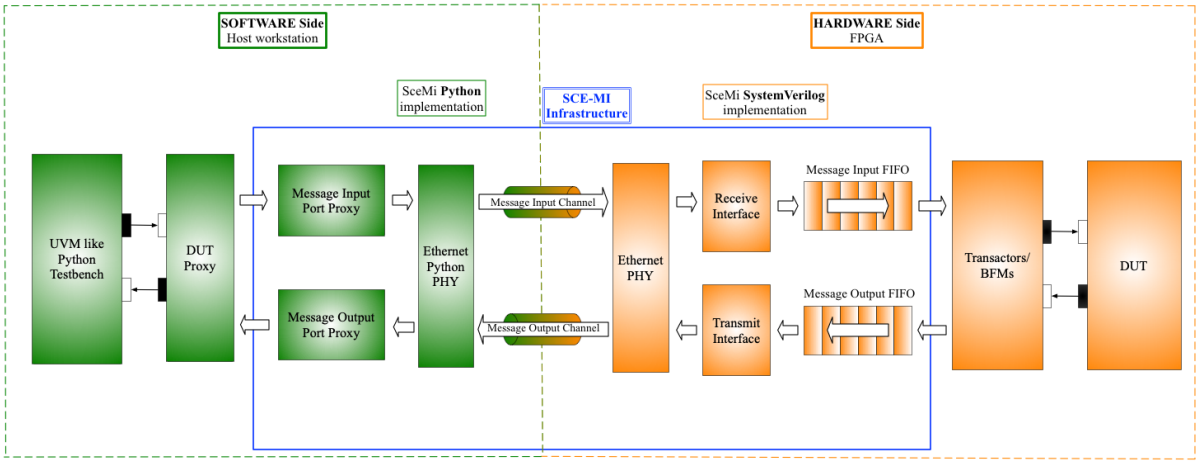


Figure 5.1: High Level view of HW-SW communication through SCE-MI infrastructure

The host workstation side contains:

- Test-bench: it is structured as a UVM TB, but written in Python, allowing us to run it on any workstation where Python is installed
- DUT proxy: it serializes messages into a large bit vector, before transporting them through the input channel. It de-serializes them back into a Python data type, after they are received on an output channel
- Message input/output Port Proxies: send/receive messages to/from hardware side
- Python PHY: it allows to open sockets to communicate over the physical connection with the hardware side

The FPGA side contains:

- PHY: embedded on the programmable board
- Receive/Transmit Interfaces: they respectively receive from and send messages to the PHY
- Message FIFOs: where messages received from or sent to each transactor are stored
- Transactors: they implement BFMs that interface to the DUT. A transactor decomposes messages arriving at input channels from the software side into sequences of cycle-accurate events. For the other direction of flow, a transactor recomposes sequences of events coming from the DUT back into messages to send to the software side

5.2 Software block

The software layer, depicted in green in Figure 5.1, includes the Python Test-bench and the software side of the Macro-based SCE-MI interface. A description of them is provided in the next sections.

5.2.1 UVM-like Test-bench

The structure of the Python Test-bench has been created close to the one defined in the UVM standard. The hierarchy of the Test-bench is shown in Figure 5.2.

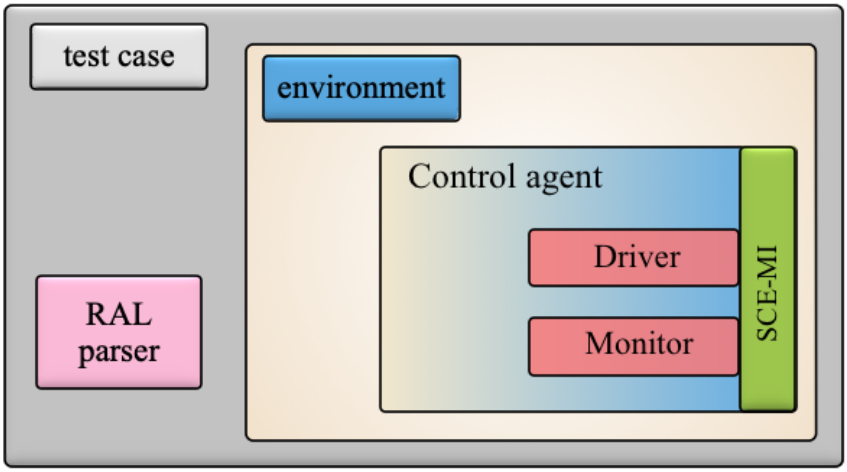


Figure 5.2: UVM-like Python Test-bench structure

It starts with the test case, which contains the Register Abstraction Layer (RAL) parser and the environment. Only the control agent is instantiated inside the environment. In contrast to the UVM guidelines, this agent does not implement a sequencer since for this specific use case it is not required. Another difference to common UVM TBs, is the absence of UVM ports. Instead, queues and methods are used to forward transaction items between components.
This TB is purely transaction-level based, to be able to interact with the Sce-Mi SW-HW intra-domain API.

The test case can be potentially any test that extends the base test class. The base test instantiates the RAL parser, the environment and provides the following methods:

- *run_phase()*: this method calls the *run_phase* of the control agent, the *run_test_seq* and the *stop_all_threads* method, which needs to be implemented by the child class
- *write(reg_name, val)*: it triggers a write transaction by creating a sequence item of type *seq_item*. This transaction item is then forwarded to the *send_transaction* method of the environment
- *read(reg_name, non_blocking, timeout_sec=5)*: it triggers a read transaction by creating a sequence item of the type *seq_item*. This transaction item is then forwarded to the *send_transaction* method of the environment. Only in case the argument *non_blocking* is set to False, the *read method* waits for the response of the read

transaction. A timeout is triggered when there is no transaction for more than *timeout_sec*. The default value is 5

- *stop_all_threads()*: it calls the same method from the environment
- *power_up()*: this method configures the DUT's registers
- *dump_csv_file*: it extracts the value of the DUT's measurements and dump them into a csv file

The sequence item class contains all the information required for a transaction and implements three methods. Two of them are used to set the register name, the address and the access type of the read/write instruction.
In case the access type is read, *is_read_transaction()* returns true.

The RAL parser takes a *.json* file as input and returns a RAL object. In the current application, the functionality of the RAL object is very limited. It provides only one method *get_reg_address(reg_name)*. However, this simple method allows to do register accesses by name.

The environment component is a container of the control agent and it has three methods. *send_transaction(seq_item)* is used to forward the sequence item to the agent. Vice-versa, *get_transaction(timeout_sec)* returns the sequence item received from the agent.
stop_all_threads() is used to call the same method from the control agent.

The control agent instantiates the driver and the monitor. It implements two methods used in the environment:

- *send_transaction(seq_item)*: it puts the sequence item into the in_port queue of the driver
- *get_transaction(timeout_sec)*: it tries to get a sequence item from the out_port queue of the monitor. In case of expired timeout, an exception is issued

The driver is defined as an extension of another driver class. This defines two queues: *in port* forwards the transaction items to the Test-bench and *ready port* is used for flow control.
The main function of the driver is to send a write or read transaction, depending on the sequence item type.
It also defines the *ready* method. This is a callback method, called by the SCE-MI layer whenever a pre-defined READY packet is received from the hardware. This *ready* method fills the *ready_port* queue with the expected number of freed slots. If this number is higher than zero, the driver sends transactions to the FPGA.

The monitor extends the *AeonvMonitor* class, which defines two queues: *out_port* and *recv_port*.
The monitor implements the *recv* method, which is called by the SCE-MI layer whenever the hardware returns a transaction. This function maps the *SceMiMessageData* to the

sequence item and puts it into the *recv_port* queue. In the *run_phase* the monitor moves the sequence item from the *recv_port* queue to the *out_port* queue.

5.2.2 SCE-MI SW layer

The original Macro-based SCE-MI standard was only specified in C and C++. A Phyton version of it has been defined and used in this research project. Its implementation is identical to the C/C++ existing ones.
The software side interface is described by the following classes:

- class SceMiEC: used for error handling
- class SceMi: this represents the software side of the infrastructure and contains methods for global interface operations (initialization, shutdown, binding...)
- class SceMiMessageInPortProxy: it presents to the application a proxy interface to interact with a transactor's driver on the hardware receive side. The flow direction is from software to hardware
- class SceMiMessageOutPortProxy: it presents to the application a proxy interface to interact with a transactor's monitor on the hardware transmit side. The flow direction is from hardware to software
- class SceMiParameters: it provides access to the interface parameters
- class SceMiMessageData: it represents the vector of message data that will be transferred from the port proxy to the hardware and vice-versa

The API gives the Test-bench access to a set of input/output ports that communicate to the hardware side on the SCE-MI interface. To gain this access, the software side shall first initialize its infrastructure, then bind to the hardware side.
Initializing the SCE-MI means loading a parameter file containing the information to establish the connection to the hardware side through the port proxies. Each port has a set of parameters defined in the *SceMiParameters* class, which is initialized by a configuration file.
The *SceMiParameters* class allows to:

- define the communication channel (Test-bench acceleration or HDL simulation)
- define the number and names of transactors
- define names and widths of message ports

Class *SceMi* defines the main method of the infrastructure: the *ServiceLoop()*. It is called by the SW Test-bench and it is used to service the port proxies by checking for arriving or pending messages, dispatching input-ready information and receiving callbacks.
Messages transmitted over the communication channels should be compliant with a pre-defined format. The frame of a generic SCE-MI message is shown in Figure 5.3.

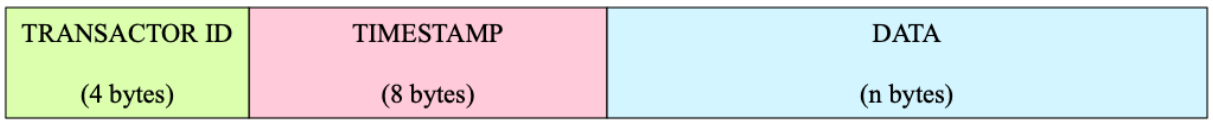


Figure 5.3: Generic SCE-MI frame

It contains following fields:

- Transactor ID: this is a unique identifier for the transactor and related port to which it is destinated
- Timestamp: timing information should be encapsulated in this frame
- Data: the message that shall be sent to the transactor

5.3 SW to FPGA communication channel

The communication between two systems, based on a standard protocol, is described by the Open Systems Interconnection model (OSI model) [Neu05]. The OSI model presents a structure of the communication system into several abstraction layers (Table 5.1).

	Layer	Function
7	Application	High-level APIs
6	Presentation	Translation of data (encoding, encryption..) to allow the communication between user application and network
5	Session	It manages communication sessions between two nodes
4	Transport	Responsible for end-to-end communication over a network
3	Network	It manages multi-node network: it performs addressing, routing and traffic control
2	Data link	It handles the movement of data between two nodes into and out from a physical medium
1	Physical	Transmission and reception of raw bits over a physical medium

Table 5.1: Layers of OSI model

To choose a communitation protocol, the available interfaces on the board have been analyzed. The programmable board used in this research project, Arria 10 FPGA, offers three interface options. Table 5.2 shows a comparison among them.

Interface	Maximum connection speed	Number of interface pins	Protocol complexity
RS232	10-100 Mbps	5	Low
Ethernet	1.25 Gbps	9	Medium
PCIe	8 Gbps/lane, max. 64 Gbps full-duplex	38	High

Table 5.2: Features of Arria 10 interfaces

All the three described interfaces are serial. RS232 has a very low speed which could corrupt the main intent of the project: consistently speed-up the test execution. PCIe would provide the highest possible speed compared to the other two. However, implementing its communication protocol on pure hardware would require a high effort. It is indeed based on a Requester-Completer protocol [PCI].
On the other side, Ethernet does not provide the highest possible speed. However it allows to implement a simpler protocol, such as the User Datagram Protocol (UDP) [Fai08].

5.3.1 User Datagram Protocol

Among the various Ethernet protocols, IPv4 UDP has been used. It requires a minor number of frame fields compared to other ones.
UDP uses a connectionless communication model (differently from TCP), without handshaking or acknowledgment mechanisms. This means that there is no certainty that the packet has been delivered. However, this makes the protocol easy to reproduce, having a shorter frame (in comparison to TCP) with a faster transmission over a serial interface.
UDP is a simple message-oriented transport-layer protocol. The SW application running on the host PC can communicate with the FPGA by opening a UDP socket and specifying the target IP address and port.
An Ethernet frame is composed of:

- Preamble (7 bytes, each byte set to 0x55): it indicates the beginning of a new transmission
- Start Frame Delimiter (SDF, 1 byte = 0xD5): it indicates the beginning of the Ethernet frame
- Destination MAC Address (6 bytes)
- Source MAC Address (6 bytes)
- Length/Type (2 bytes): according to the protocol used, it may indicate the length of the message or the protocol type. In our case, since UDP uses the IP protocol on the Network layer. The type is fixed to **0x0800**
- MAC CLIENT DATA (min 46 bytes): it contains the message that is willing to be transmitted, preceded by the headers required by the protocol used

- PAD: this field may not be present if the length of the MAC CLIENT DATA field is already higher than 46 bytes. If it is not, the PAD field (all bytes 0) is added to reach the minimum length of 46 bytes
- FRAME CHECK SEQUENCE (4 bytes): it is cyclic redundancy check (CRC) error detection code

To send a UDP packet, its protocol (implemented at Transport layer) must interact with the Network layer below. For this reason, the Ethernet frame must contain a IPv4 header and a UDP header.
The format of a general Ethernet frame and a UDP frame are shown below in Figure 5.4.

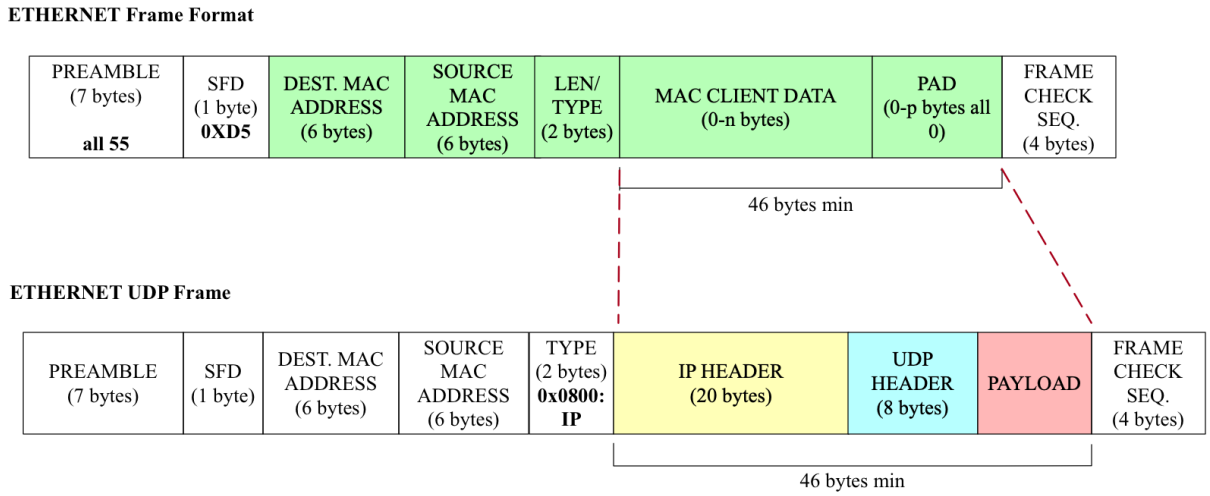


Figure 5.4: Ethernet frame and UDP frame format

5.3.2 Address Resolution Protocol

To send UDP packets from the host PC to the FPGA it is needed to just open a socket and specify target address and port. However, in order to correctly build the frame, the OS needs also the information about the target MAC address. To retrieve this information and perform the two-point connection, the Address Resolution Protocol (ARP) [Har17] is used.
As soon as the workstation tries to send the first packet to the target IP address (FPGA), if this is unknown to the OS, an ARP request packet will be sent first. This packet contains information about the sender and about the target IP that it wants to reach. It is broadcasted to all the devices connected on the network. The format is shown in Figure 5.5.

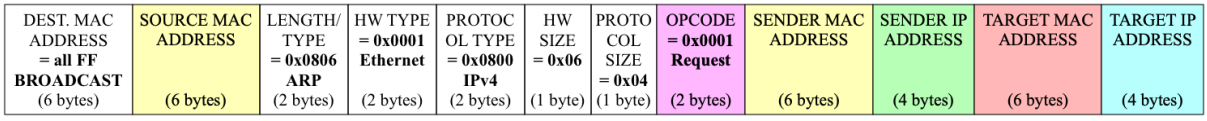


Figure 5.5: ARP Request frame format

Only the device which has IP address equal to the one listed in the TARGET IP ADDRESS field must answer with a ARP Reply frame. The content of the reply is shown

in Figure 5.6. Fields that have equal colors in Request and Reply frames have same content.

DEST. MAC ADDRESS (6 bytes)	SOURCE MAC ADDRESS (6 bytes)	LENGTH/ TYPE = 0x0806 ARP (2 bytes)	HW TYPE = 0x0001 Ethernet (2 bytes)	PROTOC OL TYPE = 0x0800 IPv4 (2 bytes)	HW SIZE = 0x06 (1 byte)	PROTO COL SIZE = 0x04 (1 byte)	OPCODE = 0x0002 Reply (2 bytes)	SENDER MAC ADDRESS (6 bytes)	SENDER IP ADDRESS (4 bytes)	TARGET MAC ADDRESS (6 bytes)	TARGET IP ADDRESS (4 bytes)

Figure 5.6: ARP Reply frame format

Only after the ARP Reply is received and the sender has registered the Target Mac address, UDP packets can be sent.
Figure 5.7 shows the communication protocol.

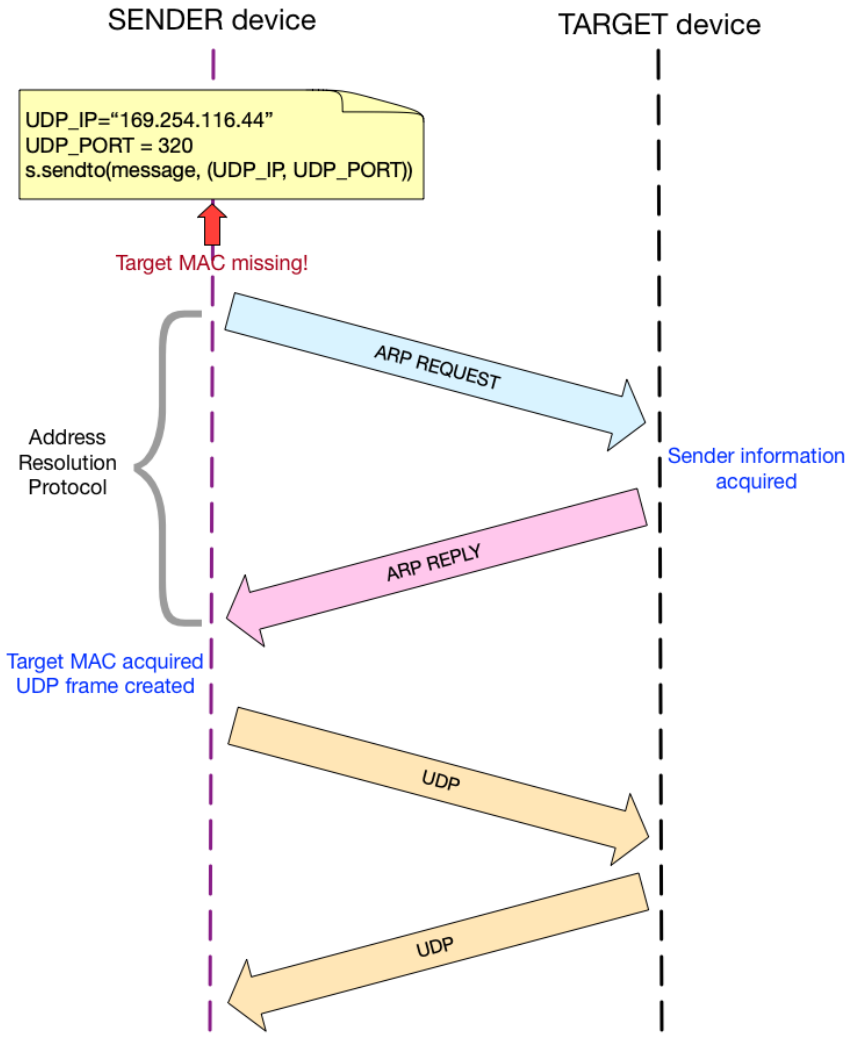


Figure 5.7: ARP protocol before UDP packets transmission

As soon as the ARP protocol terminates, a new ARP entry is inserted in the ARP table of the OS, where information like IP address, MAC address and interface are stored. The OS has a timeout, after which it deletes the ARP entry and eventually asks again a new ARP request.
To avoid designing an ARP detector on the FPGA side, another solution can be adopted:

it is possible to add a permanent entry to the ARP table of the OS, such that the ARP synchronization does not need to take place. The information needed to register a new entry in the ARP table are:

- IP address of the target IP (FPGA): decided by the designer
- MAC address of the target device: decided by the designer
- name of the interface on the host workstation

5.3.3 Ethernet interface on FPGA

Arria 10 GX platform has an embedded Marvell 88E1111 PHY [Mar13]. This PHY supports 10/100/1000 base-T Ethernet connection in combination with the Altera Triple-Speed Ethernet IP [Int19c], provided by the Intel Library.
Marvell 88E1111 PHY uses 2.5 V and 1.0 V power rails and requires a 25 MHz reference clock. This clock signal is driven by default from an embedded 25 MHz oscillator.
A RJ45 connector communicates with the Ethernet PHY through a MDI [20119] interface. The PHY-to-FPGA connection employs SGMII at 1.25 Gbps transmit and receive. In 10-Mb or 100-Mb mode, the interface still runs at 1.25 GHz, but the packet is repeated 10 or 100 times.

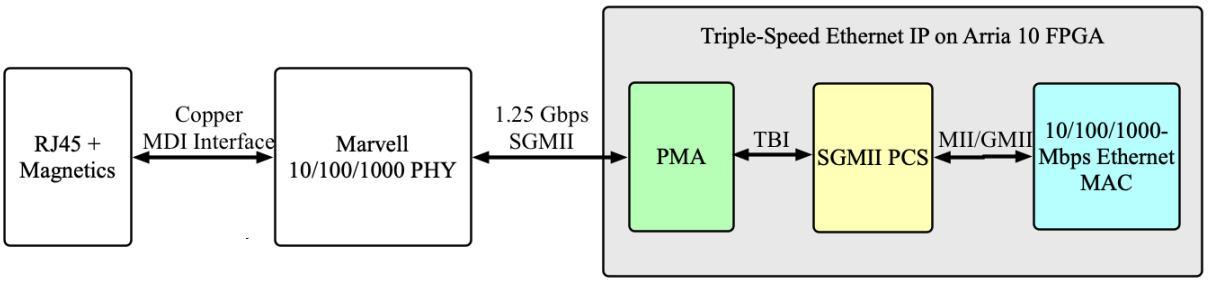


Figure 5.8: Ethernet communication interface

The Triple-Speed Ethernet IP is responsible for processing the incoming/outgoing Ethernet frames. It has been configured to contain:

- Physical Medium Attachment (PMA): responsible for the deserialization/serialization of the incoming/outgoing data
- Physical Coding Sublayer (PCS) [Fra98]: performs the 8B/10B encoding/decoding and is also responsible for the auto-negotiation process
- Media Access Control (MAC): its main functions are to append/check the FCS, discard malformed frames and attach/remove preamble, SDF and padding

5.3.4 Ethernet configuration register blocks

The Triple-Speed Ethernet IP can be configured by updating the content of its register space. It goes from address 0x00 to 0xFF and it is divided in the following way:

- Address space 0x00 - 0x7F contains MAC control registers

- Address space 0x80 - 0x9F contains PCS control registers
- Address space 0xA0 - 0xFF contains MDIO control space and registers for additional optional functionalities

The interface on the TSE to its control registers is an Avalon-MM interface [Int19a]. In order to upload the intended content in the TSE configuration registers, two blocks have been designed (Figure 5.9):

1. **Instruction_mem**: this block contains a memory-like structure in which the instructions that should go through the Avalon-MM interface are stored
2. **Configuration_reg_bridge**: this block passes the instructions read from the previous block to the Avalon-MM interface

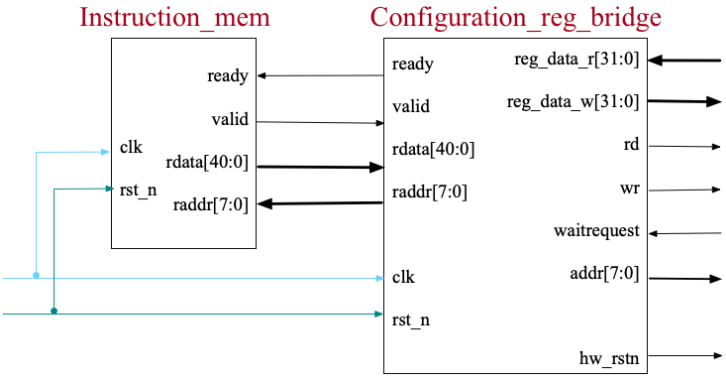


Figure 5.9: Configuration register blocks

The *Instruction_mem* block contains an array where the operations that should be performed on the TSE's configuration registers are saved. Each row of the array contains a 42-bit packed data as shown in Figure 5.10.

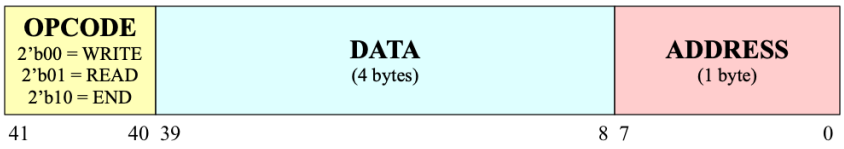


Figure 5.10: Array row format in Instruction_mem block

The opcode field can be *READ*, *WRITE* or *END*, meaning that the operations to be performed on the configuration register space of the TSE are finished.
The loaded instructions can be divided in two subgroups:

- PCS register space configuration:
  1. Enable SGMII interface
  2. Enable auto-negotiation
  3. PCS reset
- MAC register space configuration:

1. MAC address configuration: 00-1C-23-17-4A-CB
2. Maximum frame length and inter-packets gap length
3. Pause quanta timeout
4. MAC SW reset and enable TX/RX paths

Instruction_mem_block has been implemented with a 5-states FSM (Figure 5.11):

- IDLE: reset and default state
- MEMORY_LOAD: all the instructions are loaded into the array rows
- WAIT_TRANS: wait for an instruction request by *Configuration_reg_bridge*
- SEND_DATA: *valid* is asserted and the instruction is sent to *Configuration_reg_bridge*
- WAIT_READY_DEASSER: wait for *Configuration_reg_bridge* to release the current request and be ready for another one

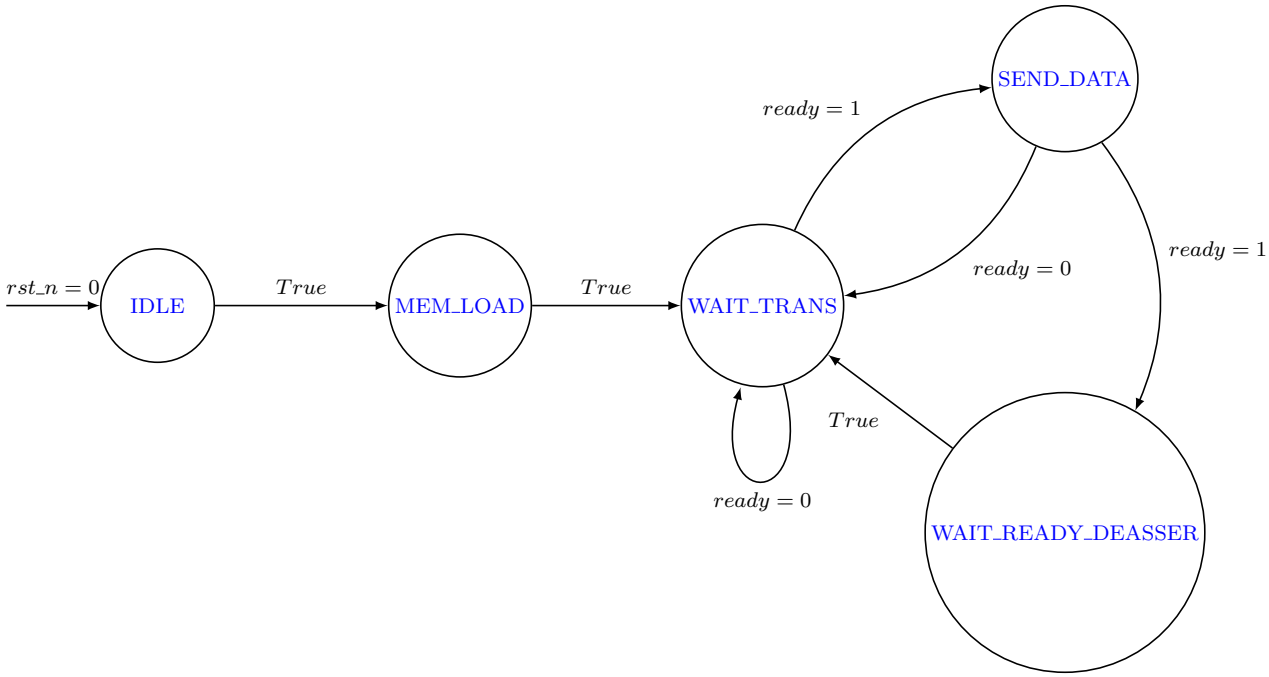


Figure 5.11: FSM of *Instruction_mem* block

The *Configuration_reg_bridge* reads the instructions from the *Instruction_mem* block and decodes them to perform the correct operations towards the TSE configuration registers. It has been implemented a 7-states FSM (Figure 5.12):

- IDLE: reset and default state
- READ_INSTR: read the instruction from a row of the *Instruction_mem* block, whenever *valid* = 1
- INSTR_REC: the instruction is decoded

- WRITE: write a register of the TSE's register space
- READ: read a register of the TSE's register space and compare its content to the expected value. If the value does not match, the register is re-read until its content matches the expected value. This feature is mainly used for reset bits, which could take some time to be cleared
- NEXT_INSTR: the counter to fetch another instruction from the *Instruction_mem* block is incremented
- END_INSTR: the valid instructions in the *Instruction_mem* block have all been read, so the FSM blocks in this state

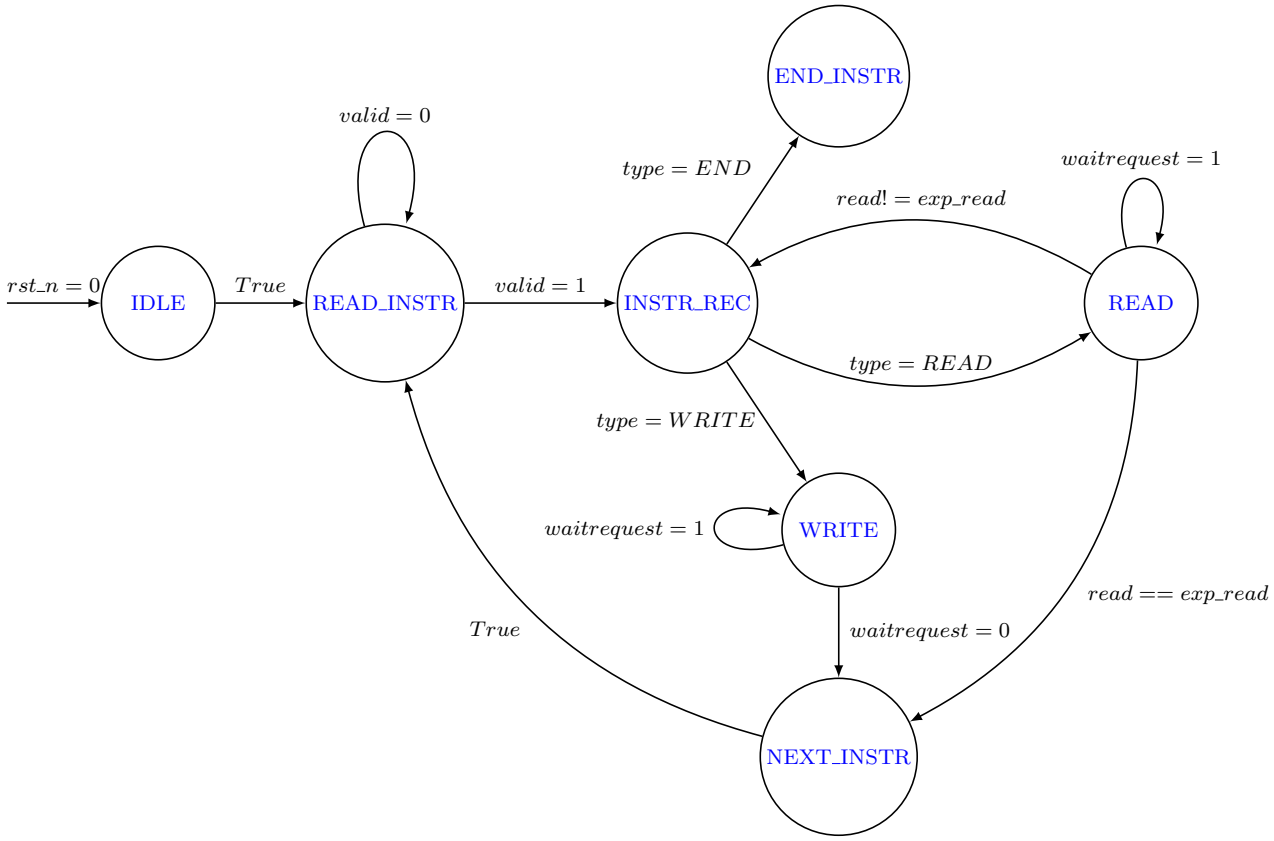


Figure 5.12: FSM of *Configuration_reg_bridge* block

5.4 Hardware structure

This portion of the architecture contains the hardware side of the infrastructure. It is loaded on the FPGA and is meant to connect the Ethernet interface to the DUT. The complete schematic of the architecture loaded onto the FPGA is shown in Figure 5.13

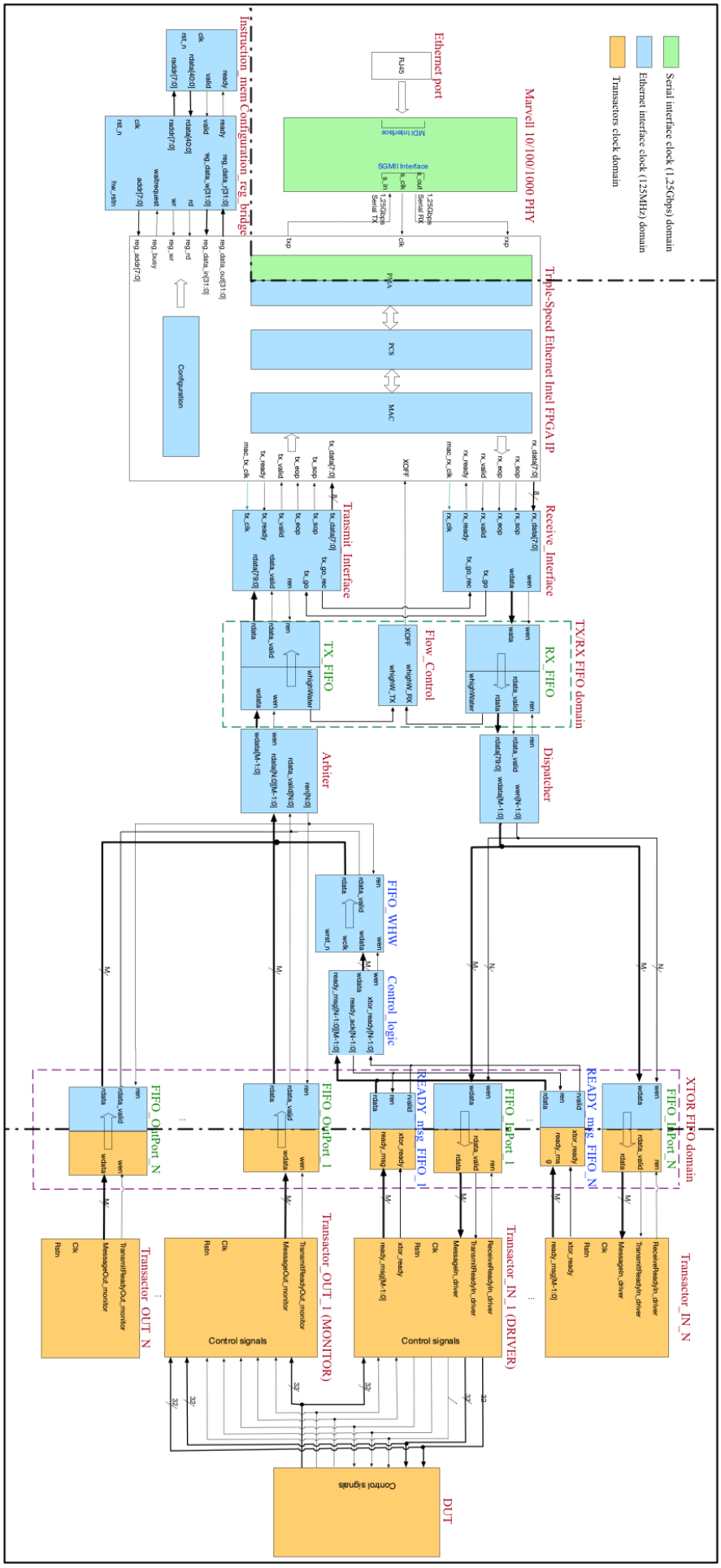


Figure 5.13: Infrastructure on FPGA

5.4.1 Receive interface

The *Receive interface* block is responsible for receiving packets from the TSE. Over the Ethernet connection, not only SCE-MI packets can flow. It is very likely that other packets used for synchronization and control are also transmitted. The Receive interface must then identify the SCE-MI and propagate only them to the rest of the infrastructure. On one side it is connected to the Avalon Streaming interface [Int19a] of the TSE and, on the other side, to the *RX FIFO*. Its block diagram is shown in Figure 5.14.

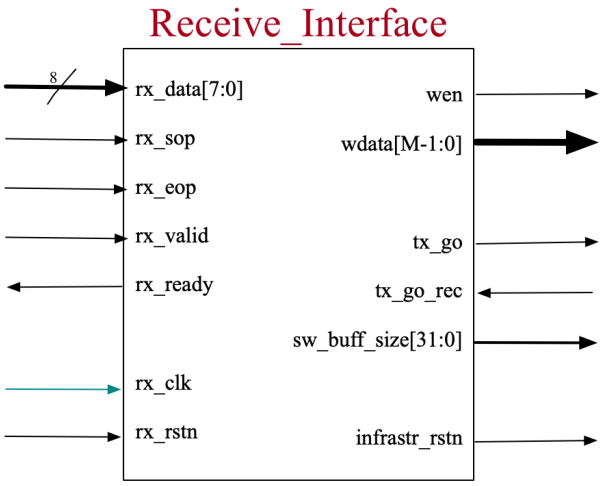


Figure 5.14: Block diagram on *Receive_interface*

It is configured from the top level design with the following parameters:

- FRAME_WIDTH: Number of bytes of an IPv4 UDP Ethernet frame, as the one shown in Figure 5.4
- MEX_WIDTH: Number of bytes of the extracted SceMi packet containing an instruction, as the one shown in Figure 5.3

This block collects the bytes streamed out by the Avalon Streaming Source RX interface of the TSE. The first byte is collected as soon as the *rx_sop* signal is asserted, while the last byte is sent in correspondence with the assertion of the *rx_eop* signal. Transmission happens from the Most Significant byte to the Least Significant byte.
The Receive interface must detect the entity of the packets received. There are three types of packets that can be propagated:

- Reset packets: used to force the infrastructure reset from the SW side. They can be identified by ITEM_TYPE = 0xA5A5A5A5.

TRANSACTOR ID (4 bytes)	TIMESTAMP (8 bytes)	ITEM_TYPE = 0xA5A5A5A5 (4 bytes)	DATA (4 bytes)	ADDRESS (4 bytes)

Figure 5.15: Reset packet

- SceMi packets for transactors: they carry valid instruction for the transactors. The *Receive_interface* block checks the ITEM_TYPE field first, then if it is neither

a Reset or Flow control packet, the TIMESTAMP field is checked. It is used as identification field for transactors messages, by setting it to a pre-defined value. The Standard defines that the TIMESTAMP field should carry timing information. In our architecture, this field was unused so, instead of creating another identification field for the SceMi transactions, the TIMESTAMP has been used.

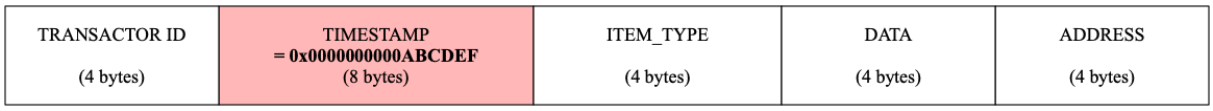


Figure 5.16: SceMi packets for transactors

- Flow control packets: they are sent from the SW to the FPGA to inform the hardware side of how many slots the SW has available. If sent right after the Reset packet, the Flow control packet has also the function of informing the FPGA that it can start sending. These packets are identified by the field ITEM_TYPE = 0xFFFFFFF. The field normally used as ADDRESS now contains the information about the number of available slots in the SW buffer.
  The number of available slots is communicated by the Receive interface to the Transmit interface.

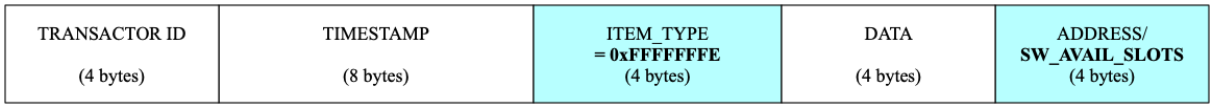


Figure 5.17: Flow control packets

This block has been designed as a 7-states FSM (Figure 5.18):

- IDLE: reset and default state
- RUN_IF: the Receive interface asserts the *rx_ready* signal and waits for the start of a packet transmission (*rx_sop* = 1). Then *infrastr_rstn* signal is released
- READ_PKT: a packet transmission has been initiated, so the Receive interface saves the incoming bytes into a frame
- END_PKT: *rx_eop* has been asserted, so the packet has been fully received. The Receive interface detects the packet type: if it is not a valid packet, the FSM returns back to the RUN_IF state
- WRITE_FIFO: the packet received is a SceMi packet, so it is written in the RX_FIFO
- TX_GO_FRAME: the packet received is a Flow control packet, so a dual-ready protocol with the Transmit interface is initiated
- RST_INFRASTRUCTURE: the packet received is a Reset packet, so the infrastructure reset is enabled (*infrastructure_rstn* = 0) for one clock cycles and released right after

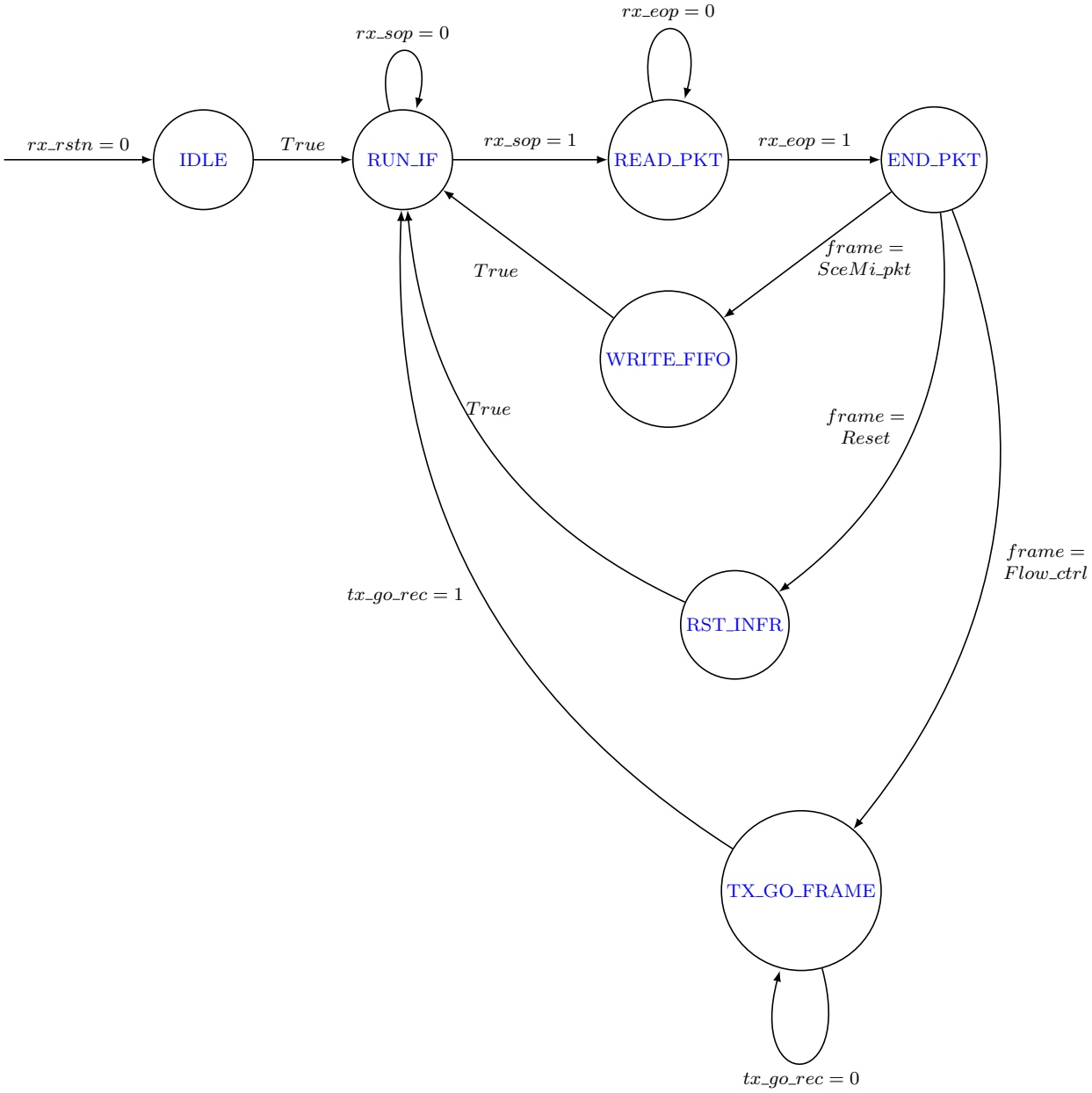


Figure 5.18: FSM of *Receive_interface* block

5.4.2 TX/RX general FIFOs

These FIFOs contain all the messages coming from the Receive interface or going to the Transmit interface. They can be parameterized according to their depth, width, highWaterLevel and lowWaterLevel. These parameters are set in the top level:

- DEPTH_TX_RX_FIFO: depth of the FIFOs
- MEX_WIDTH: width in bytes of the FIFOs
- WHW_TX_RX_FIFO: high water level of the FIFOs

AsyncFifoFlop_adl FIFOs are designed to have read-side and write-side running at different frequencies. In this case, both sides run at the interface speed (125MHz).
The clock signals that drive the rx/tx sides of the hardware interface come from the TSE, which is clocked with a 125MHz clock signal embedded on the board. The IP generates in turn two clock signals *tx_clk* and *rx_clk* both running at 125MHz.
RX_FIFO has the write-side driven by *rx_clk* and the read-side driven by *tx_clk*.
TX_FIFO has both sides driven by *tx_clk*.
The remaining part of the hardware infrastructure running at 125MHz is clocked by *tx_clk*.

5.4.3 Flow control for TX/RX FIFOs

Whenever a FIFO domain is present in the infrastructure, a related Flow control mechanism must be put in place. Looking at Figure 5.13, two FIFO domains are present:

1. TX/RX FIFO domain
2. Transactor FIFO domain

In the same way, flow control has been divided in two stages:

1. First level: it relates to the TX/RX FIFO domain
2. Second level: it relates to the transactor FIFO domain

In the first level, the *Flow_Control* block (Figure 5.19) checks if RX_FIFO or TX_FIFO reach their *highWaterLevel* mark. Whenever one of the two FIFOs reaches it, it asserts the *whighWater* signal.

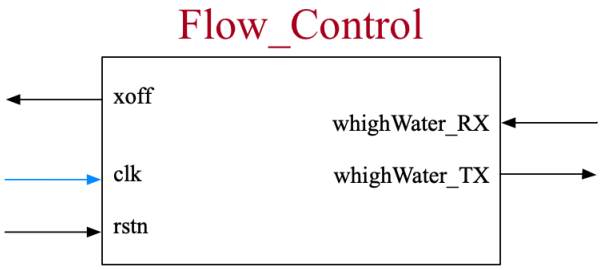


Figure 5.19: Block diagram of *Flow_Control*

The *Flow_Control* block continuously samples these two signals and, whenever one (or both) is high, it asserts the *xoff* signal at the interface of the TSE IP. This automatically triggers the generation of an XOFF packet, which is the pause frame [Eth20] compliant to the IEEE 802.3 Ethernet standard. Its format is shown below.

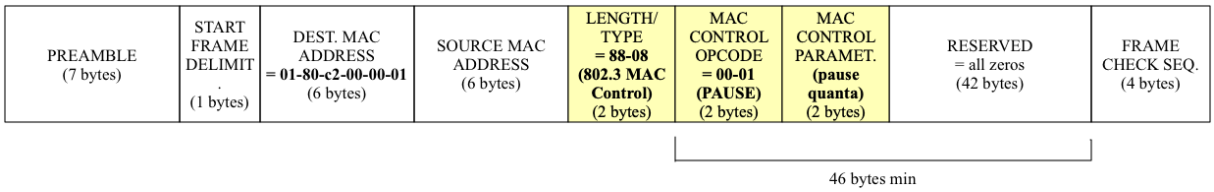


Figure 5.20: Ethernet XOFF frame

5.4.4 Dispatcher

This block reads a message from the general RX_FIFO, extracts the TRANSACTOR_ID field and writes the message into the FIFO related to the proper transactor. Its block diagram is shown below in Figure 5.21.

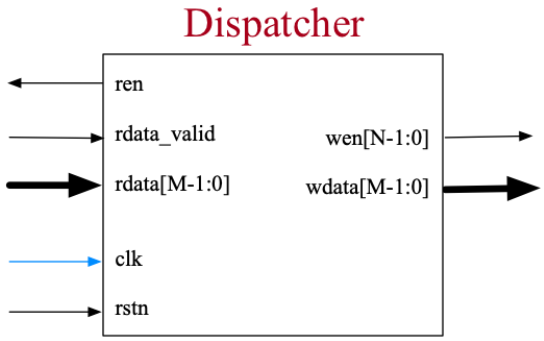


Figure 5.21: Block diagram of *Dispatcher*

This block is parameterized from the top level:

- N_TRANSACTORS: number of transactors in the infrastructure. For the current case study, there is only one: the transactor
- MEX_WIDTH: number of bytes of the extracted SceMi packet containing an instruction

In Figure 5.21, N = N_TRANSACTORS and M = 8*MEX_WIDTH.
wen[N-1:0] is an array where each i-th bit is connected to the *wen* signal of the FIFO related to the transactor with $TRANSACTOR\_ID = i$. IDs for transactors are then supposed to be assigned in increasing order. For the current case study, having just 1 transactor, *wen* reduces to a single bit signal and $TRANSACTOR\_ID = 0$.
wdata is routed to all the transactors FIFOs. Only the one with its *wen* signal asserted will push in the current data at its write interface.
The Dispatcher block has been implemented with a very simple 2 state FSM (Figure 5.22), whose states are:

- IDLE: reset and default state
- READ_RX_FIFO: data is read from the general RX_FIFO and routed to the dedicated transactor's FIFO

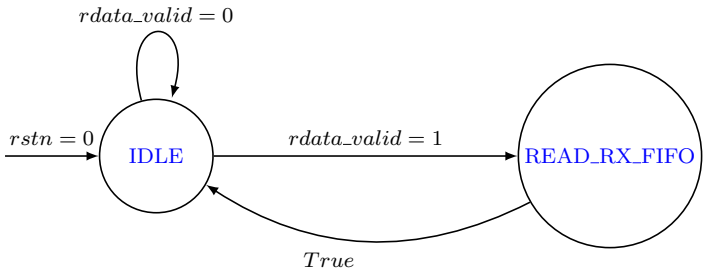


Figure 5.22: FSM of *Dispatcher* block

5.4.5 Transactor FIFO domain

In this domain the clock domain crossings (CDCs) between the infrastructure clock and the transactors clocks takes place. Each FIFO in this domain has the write-side driven by the interface clock signal (*tx clk* = 125 MHz), while the read-side is driven by the clock signal of its own transactor. The depth of these FIFOs should be smaller than (or at least equal to) the one of the general RX/TX FIFOs. They can be parameterized according to their depth, width, highWaterLevel and lowWaterLevel. These parameters are set in the top level:

- DEPTH_XTOR_FIFO: depth of the transactors FIFOs
- MEX_WIDTH: width in bytes of the transactors FIFOs
- WHW XTOR FIFO: high water level of the transactors FIFOs

The flow control for these FIFOs is managed by the transactors themselves which exchange "READY" packets with the software. More details about this are provided in section 5.4.7. For this reason, *full* and *whighWater* signals are not used.
Constraints for CDC must be applied. They can be generated with the help of the *Timing Analyzer* tool provided by Intel: from Quartus command line click *Tools* > *Timing Analyzer*. Once the GUI opens, select the constraints that are needed from the *Constraints* tab in the command line.

5.4.6 Transactors

SceMi transactors are divided in a software part and in a hardware counterpart. Communication between them happens through messages whose format is compliant to the SceMi standard.
Each transactor is identified by a unique ID, which is shared by the Driver and the Monitor (if both present) of the same transactor. Drivers and Monitors are implemented as Bus Function Models (BFMs). They communicate on one side with their dedicated FIFO and on the other side with the DUT.
For the current case study, the transactor is the only one present and it is implemented with a Driver and a Monitor. Its TRANSACTOR_ID is 0.
In case of multiple transactors, they can obviously run at different speeds: the same infrastructure can contain a very fast transactor and a very slow one. Dedicated FIFOs and the second level of Flow control are indeed needed to prevent to have messages for the fast transactor stuck behind the ones for the slow one.

5.4.7 Flow control for transactor FIFOs

This section of the architecture implements the second level of flow control, as mentioned in Section 5.4.3. It is composed of three blocks: *READY_MSG_FIFO*, *Control_block* and *FIFO_WHW*.

READY_MSG_FIFO

The main function of this FIFO is to implement the CDC between the clock domain of the transactor and the clock domain (write side) of the infrastructure (read side, 125 MHz).

Since each transactor can run at a different clock speed, a READY_MSG_FIFO block must be instantiated per each transactor.
The content of the message stored in this FIFO is a "READY" message (Figure 5.23), defined by the Standard [Acc16]. This message is used for flow control: whenever a transactor is ready to receive, it sends this message to the SW side.

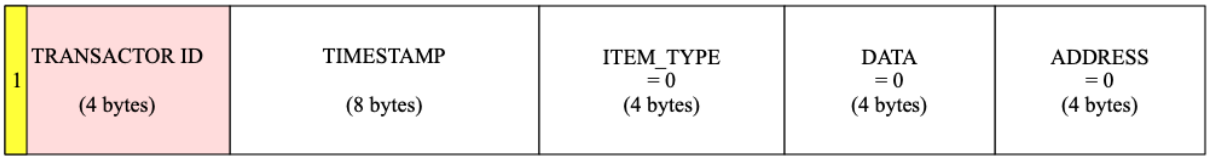


Figure 5.23: Ready message frame

A "READY" message contains the information of the TRANSACTOR_ID field, with its most significant bit set to 1. This field allows the SW side to understand from which transactor the "READY" message comes from. All the other fields are present, but ignored by the infrastructure.

Control_block

It checks by increasing order if the READY_MSG_FIFO blocks have valid READY messages. Whenever this happens the message is copied into the next fifo: FIFO_WHW. Its block diagram is shown in Figure 5.24.

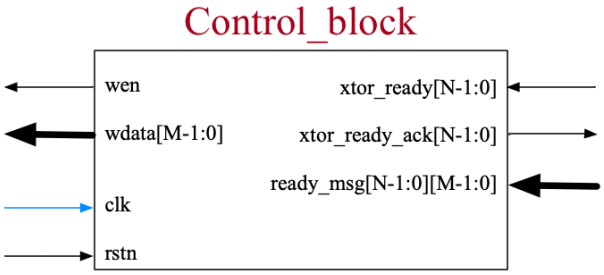


Figure 5.24: Block diagram of *Control_block*

Its parameters are the same ones of the other blocks, defined in the top level: N=N_TRANSACTOI
M=MEX_WIDTH.
xtor_ready[N-1:0] and *xtor_ready_ack[N-1:0]* signals are two arrays where each i-th bit is connected to the related signal of the transactor's Driver with TRANSACTOR_ID = i.
ready_msg[N-1:0][m-1:0] is a matrix where the READY message of the transactor with TRANSACTOR_ID = i is stored in each i-th row.
The *Control_block* has been implemented with a 3 states FSM (Figure 5.25) with states:

- IDLE: reset and default state
- CHECK_XTOR_READY: checks if the i-th READY_MSG_FIFO has a valid message
- SEND_READY: the READY message is read from the current READY_MSG_FIFO and stored in FIFO_WHW

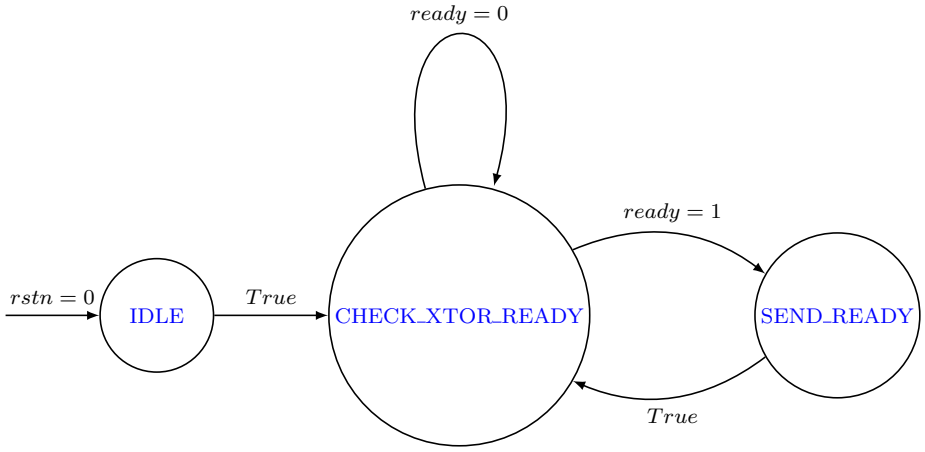


Figure 5.25: FSM of *Control_block* block

FIFO_WHW

This FIFO has both read and write sides connected to the same 125 MHz clock signal. The main function of this block is to store the ready messages of all the transactors and provide them to the Arbiter block.
FIFO_WHW has highest priority with respect to the other FIFOs connected to the Arbiter. As soon as an element is present in FIFO_WHW it is immediately written in the TX_FIFO.

5.4.8 Arbiter

This block runs at the infrastructure speed (125MHz) and is connected to:

- FIFO_OUT_XTOR blocks, from which it reads the monitored instructions
- FIFO_WHW, from which it reads the transactors READY messages
- TX_FIFO, into which it writes the read messages

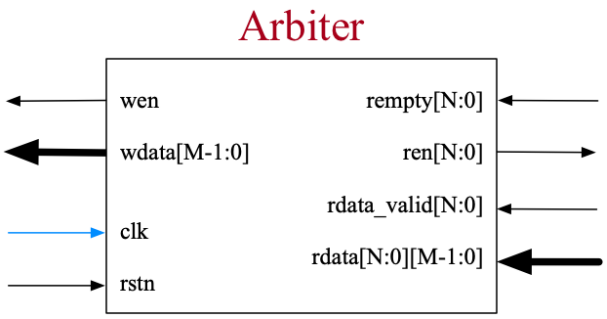


Figure 5.26: Block diagram of *Arbiter*

The *ren[N:0]* signal is an array where each i-th bit is connected to the *ren* signal of the FIFO_OUT_XTOR related to the TRANSACTOR_ID = i. The most significant bit of the array (*ren[N]*) is connected to the related signal of FIFO_WHW, containing the READY

messages.
Same connections are performed also for *rdata_valid* and *rdata* arrays.
FIFO_WHW has the highest priority in the arbitration process: the Arbiter always checks first if a message is available in this fifo. Only if FIFO_WHW is empty, the Arbiter checks, by increasing order of the transactors IDs, the *rdata_valid* signals of the transactors FIFOs.
This block has been implemented with a 6 states FSM, which states are:

- IDLE: reset and default state
- CHECK_READY: *rdata_valid[N]* is read to check if a READY message is available
- READ_READY: the READY message is read from FIFO_WHW
- CHECK_XTOR_MEX: the index i is updated, *rdata_valid[i]* is checked
- READ_XTOR_MEX: a message is read from the i-th FIFO_OUT_XTOR

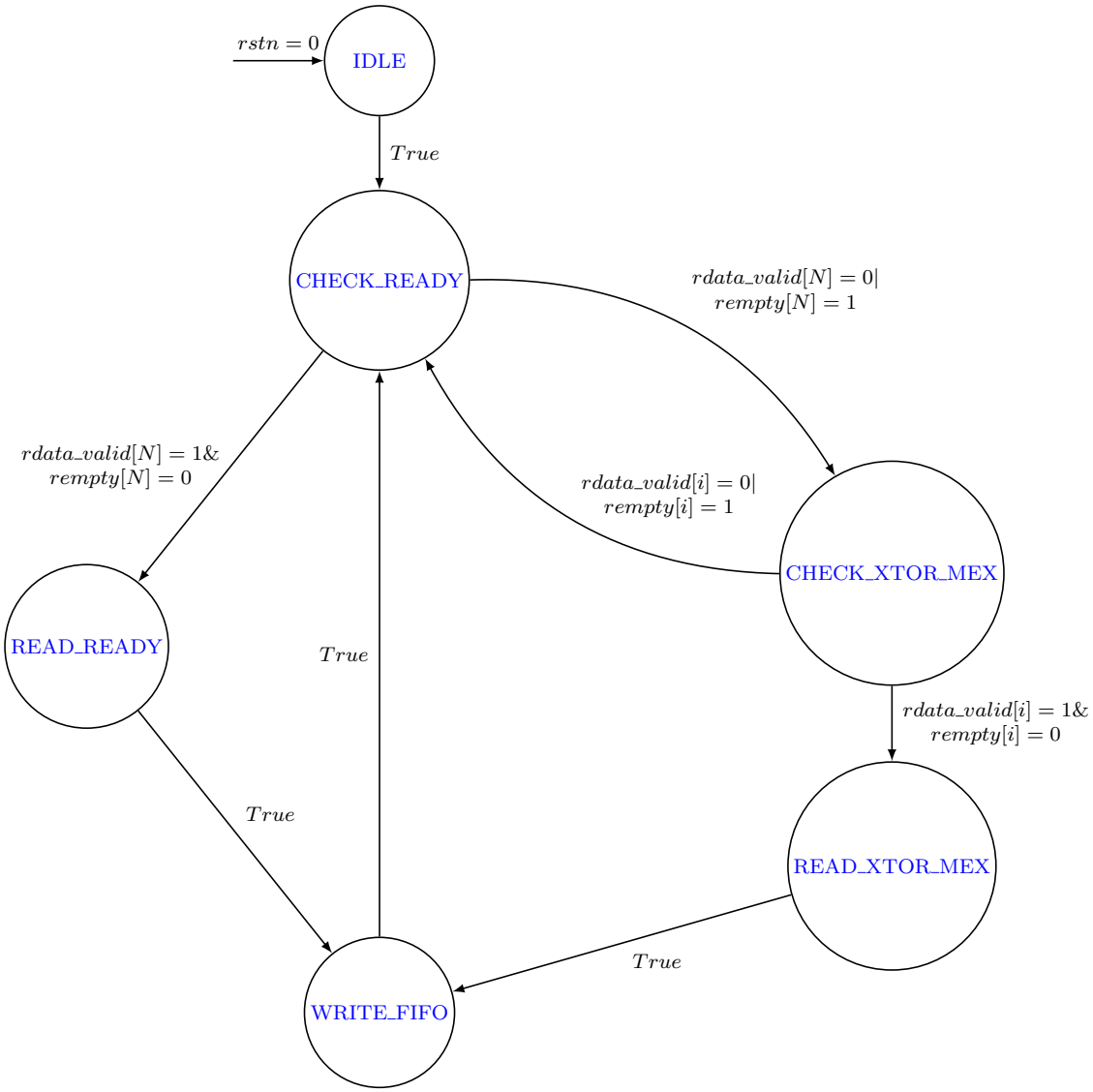


Figure 5.27: FSM of *Arbiter* block

5.4.9 Transmit interface

The *Transmit_if* block is responsible for receiving SceMi messages from the TX_FIFO, packing them into IPv4 UDP frames (as shown in Figure 5.4 of section 5.3.1) and forwarding them, 1 byte at a time, to the Avalon Streaming Transmit interface on the TSE. The block diagram of Transmit_if is shown in Figure 5.28.

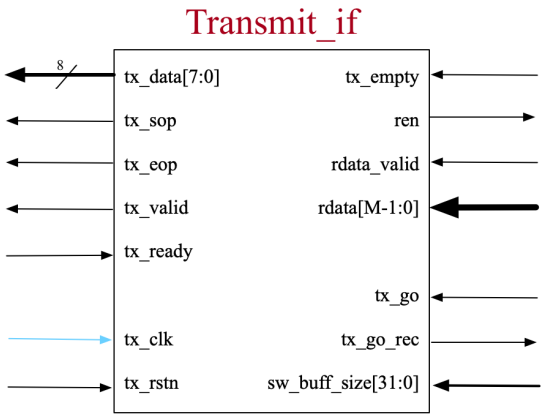


Figure 5.28: Block diagram of *Transmit_if*

Messages coming from the TX_FIFO are SceMi packets containing instructions for the transactor or READY messages. The Transmit interface block encapsulates them into IPv4 UDP packets. The SceMi message occupies the PAYLOAD field of the Ethernet packet, as shown in Figure 5.29.

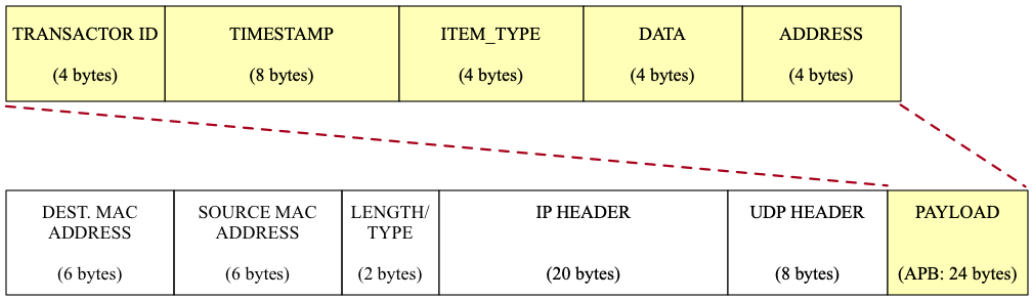


Figure 5.29: UDP frame formatting by *Transmit_if*

The transmit interface is also responsible of managing the flow of packets from the FPGA to the host workstation. The FPGA cannot send packets without taking control of the availability of the SW side to receive. Assuming that the HW is faster than the SW, if the FPGA continuously sends packets to the PC, the SW buffer will fill up. To avoid this, the Transmit interface keeps a count of the slots available on the SW buffer. The information about the number of available slots is sent by the SW to the FPGA inside a Flow control packet (Figure 5.17).
The flow control procedure goes in the following steps:

1. The SW side is ready to receive and sends a Flow control packet containing the information about the number of available slots

2. The *Receive_if* block receives the Flow control packet and starts a dual-ready protocol with the *Transmit_if*, sending to it the number of slots available

3. The *Transmit_if* block updates its internal counter
4. *Transmit_if* decrements its counter every time it sends a packet to the SW. If the counter reaches 0, even if there are messages to send, it waits for another Flow control packet to restart the transmission

This block has been implemented with a 8-states FSM (Figure 5.30). Its states are:

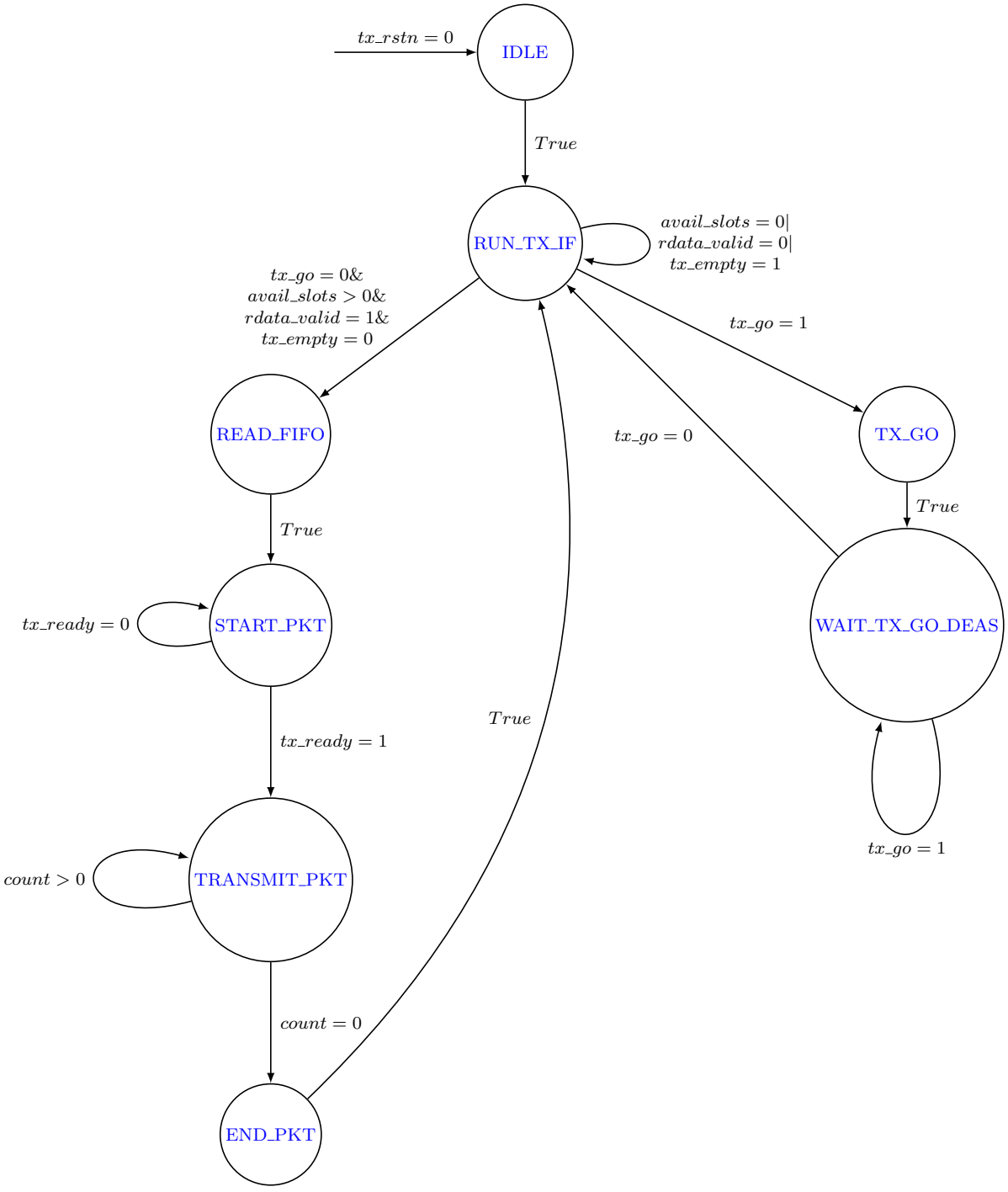


Figure 5.30: FSM of *Transmit_if* block

- IDLE: reset and default state

- RUN_TX_IF: the Transmit_if waits for a valid message to be read from TX_FIFO
- TX_GO: the number of available slots on the SW side is received from the Receive_if block. The internal counter is updated and *tx_go_rec* signal is asserted for the handshake with Receive_if
- WAIT_TX_GO_DEASSERTED: waits for the Receive_if to release the *tx_go* signal
- READ_FIFO: a message is read from TX_FIFO and the full UDP frame is created. The counter of transmitted packets is updated
- START_PKT: *tx_sop* is asserted and the first byte is sent to the TSE IP
- TRANSMIT_PKT: all the bytes (except the last one) are sent and the byte counter is constantly decreased
- END_PKT: *tx_eop* is asserted and the last byte is transmitted

5.5 Driver

The current case study has just a control interface, meaning that only a pair of transactors is needed. They are:

- Driver: it drives the interface signals to perform read and write instructions on the DUT . It is also responsible of generating the READY messages
- Monitor: it observes the interface signals to track back the performed instructions

The block diagram of the *Driver* is shown in Figure 5.31.

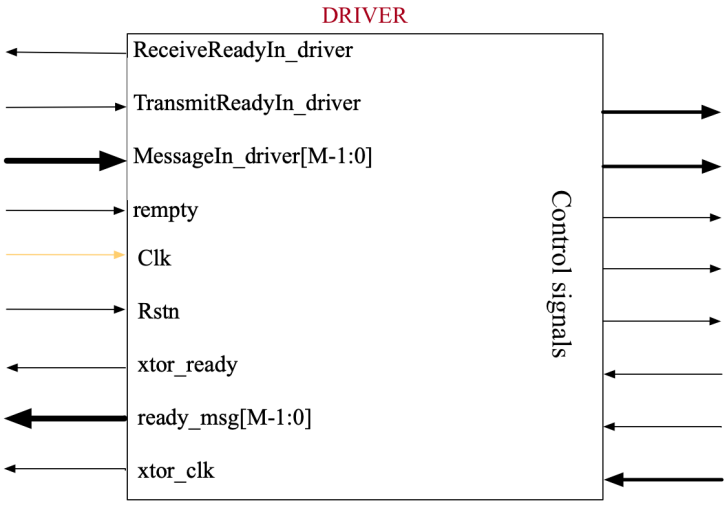


Figure 5.31: Block diagram of *Driver*

It is connected to:

- its FIFO_IN_XTOR, from which it reads messages containing instructions
- its READY_MSG_FIFO, into which it writes its READY message

- control interface on the DUT

When the Driver is ready to receive a new instruction, it raises the *ReceiveReadyIn_driver* signal and waits for the assertion of *TransmitReadyIn_driver*, meaning that the FIFO is ready to send a new message. As soon as the message is received by the driver, *ReceiveReadyIn_driver* is released.
The messages received from the FIFO are in the following format (Figure 5.32):

- *MessageIn_driver[191:160]*: transactor ID
- *MessageIn_driver[159:96]*: timestamp
- *MessageIn_driver[95:64]*: transaction type (1 : WRITE, 0 : READ)
- *MessageIn_driver[63:32]*: pwdata (data to be written)
- *MessageIn_driver[31:0]*: paddr (address of the target register)

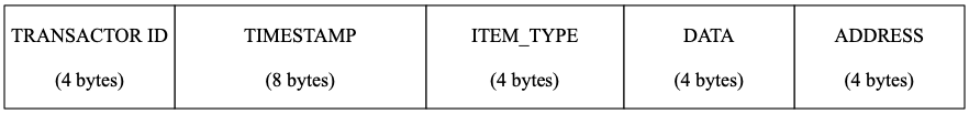


Figure 5.32: SceMi instruction frame

The Driver is also in charge of managing the second level of flow control which prevents that its own FIFO_IN_XTOR gets full. To do so, the Driver counts the number of packets that it receives. Whenever it reaches a predefined amount set in the top level entity as N_PACK_RX, the driver sends a READY message. The frame has been shown in Figure 5.23 and discussed in section 5.4.7.
The READY message is sent also to establish the initial handshake between SW and HW right after a Reset packet from the SW. In case of multiple transactors, all of them will send a READY message after their reset signals have been released.

5.6 Monitor

The *Monitor* (Figure 5.33) tracks the events at the DUT interface, detects instructions and recomposes the SceMi message.

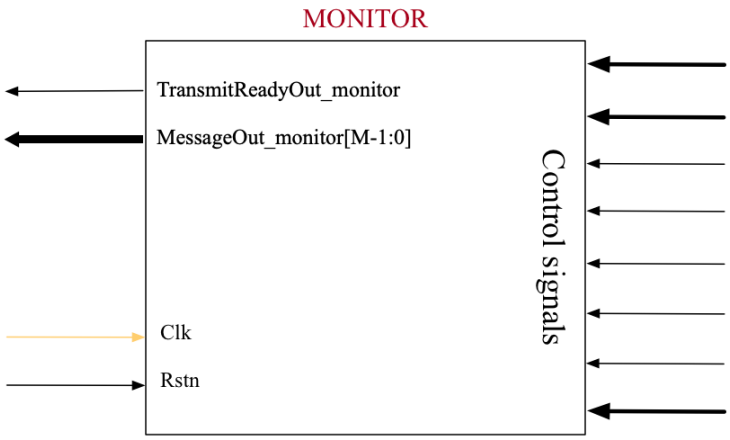


Figure 5.33: Block diagram of *Monitor*

It is connected to:

- its FIFO_OUT_XTOR into which it writes the messages containing the monitored instructions
- control interface on the DUT

When the monitor is ready to write a new message into its dedicated FIFO, it asserts *TransmitReadyOut_monitor*. The message is packed as shown in Figure 5.32.

Chapter 6

Experimental Results

Contents

6.1 Test-bench acceleration set-up

To evaluate the project performance, the total time required by RTL simulation has been compared with the one required by Test-bench acceleration.

Software set-up

The Test-bench and SW SCE-MI layer have been executed on a MacBook Pro with Intel i7 processor at 2,3GHz.
The Python test case is intended to run a set of measurements on the DUT. It contains a start-up sequence (19 write instructions to control registers) and a measurement phase. Per each measurement, the following instructions are sent:

- 1 WRITE to set the *start* bit in a dedicated register
- 1 READ to read the value of the measurement from the *result* field of a dedicated control register

These two instructions are repeated every time a measurement must be acquired. If a test-case requires 100k measurements, 200k instructions are sent from the host workstation to the FPGA.

Hardware set-up

The HW side of the infrastructure ran on Arria 10 GX FPGA. The board is connected to the host workstation through an Ethernet cable. Before running the software test case, the board must be programmed loading onto it the *.sof* file of the synthesized project's netlist.
Once the board has been programmed, the synchronization and auto-negotiation phases between FPGA and PC starts. These tasks are automatically handled by the PC's OS and the Triple-Speed Ethernet IP loaded on the board. The completion of this process is identified by a green LED on the board. Only at this point, the test case can be executed.

6.2 Simulation vs. co-emulation results

In order to evaluate how the speed-up factor scales with time, 7 different sets of increasing number of measurements have been considered: 1, 10, 100, 1k, 10k, 100k, 1M. RTL simulation with 100k measurements was killed after 65284 measurements, within 7h 55m.
The Python Test-bench has been tweaked several times during the performance evaluation phase. Some changes to the software have been made to improve the overall timing of the test case execution. Test-bench acceleration performance has been evaluated with three different revisions of the SW Test-bench.

Revision 1

The SystemVerilog UVM Test-bench used in RTL simulation contains some print instructions. To fairly compare the obtained timing results, print instructions have also

been added to the first version of the Python UVM Test-bench. The obtained results are shown in Table 6.1.

Nr. measurements	RTL sim time	TB acc time (Revision 1)	Speed-up factor
1	26s	14.517ms	1791x
10	39s	24.014ms	1624x
100	1m 27s	134.94ms	645x
1k	6m 39s	1s 198ms	333x
10k	48m 39s	12s 247ms	238x
100k	~7h 55m for 65284	2m 1s 471ms	/
1M	4d 3h 8m 46s	19m 22s 687ms	307x

Table 6.1: RTL Simulation times vs. Test-bench acceleration times

Figure 6.1 displays the results in a logarithmic scale.

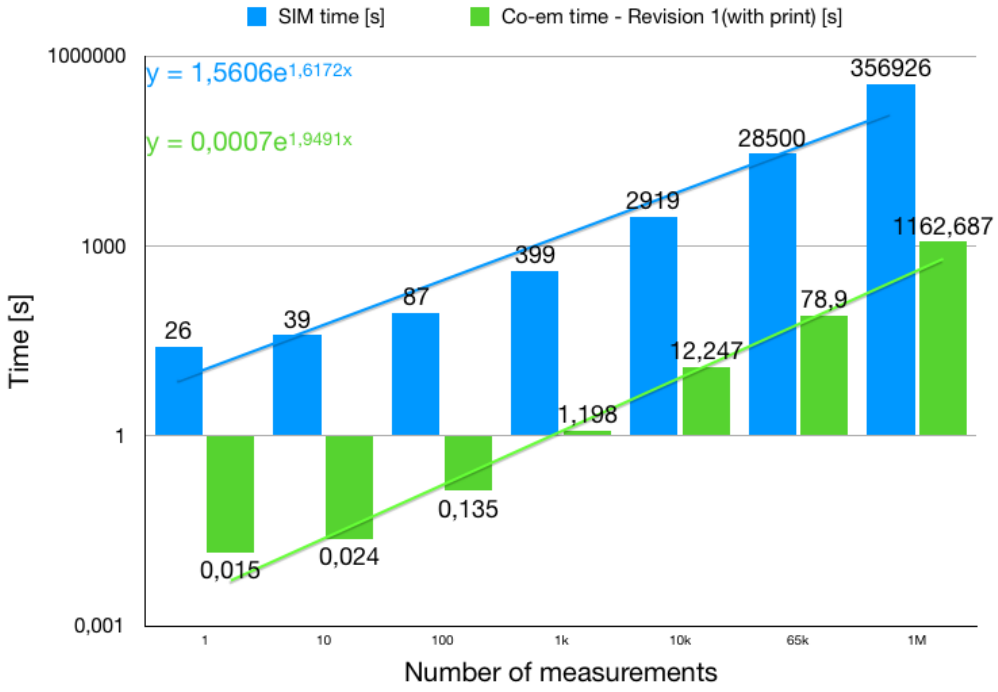


Figure 6.1: Comparison between simulation and co-emulation time results

As it can be seen from Table 6.1, for a very low number of measurements (1 or 10) the co-emulation methodology is more than one thousand times faster than RTL simulation. For higher number of measurements, the speed-up factor is around 300x.

Revision 2

Removing the print instructions has been the first improvement done to speed-up the Test-bench acceleration. The obtained results are shown in Table 6.2.

Nr. measurements	TB acc time (Revision 1)	TB acc time (Revision 2)	Ratio
1	14.517ms	12.5ms	1.15
10	24.014ms	16.6ms	1.47
100	134.94ms	63.2ms	2.13
1k	1.198s	538.4ms	2.23
10k	12s 247ms	5s 429ms	2.26
100k	2m 1s 471ms	51s	2.38
1M	19m 22s 687ms	9m 41s	2.00

Table 6.2: Test-bench acceleration times with and without print instructions

As it is possible to notice, displaying messages on the terminal slows down the DUT co-emulation of a factor equal or higher than 2.
The results in Table 6.1 and Table 6.2 have been obtained with a Python UVM Test-bench structured in such a way that every time a measurement instruction was sent, the test waited for the result and dumped it in a csv file. A large portion of time was indeed wasted by the Test-bench to wait.

Revision 3

To improve the overall performance, the task to dump the results (*dump_measurements*) have been moved at the end of the Test-bench execution.
All the obtained results are received by the software monitor and stored inside a queue. The test sends N measurement instructions to the hardware and only after that it dumps the collected results into the csv file.
Table 6.3 contains the times measured by moving the *dump_measurements* task at the end of the Test-bench.

Nr. measurements	TB acc time (Revision 2)	TB acc time (Revision 3)	Ratio
1	12.5ms	12.5ms	1
10	16.6ms	15.0ms	1.105
100	63.2ms	49.8ms	1.269
1k	538.4ms	379.9ms	1.417
10k	5s 429ms	3s 889ms	1.396
100k	51s	37s 519ms	1.359
1M	9m 41s	6m 19s	1.532

Table 6.3: Test-bench acceleration times without print with two options to dump the measurement results

Results comparison

Figure 6.2 shows a summary of the results presented in Table 6.1, 6.2 and 6.3. The Time axis has a logarithmic scale and timing values are in seconds.

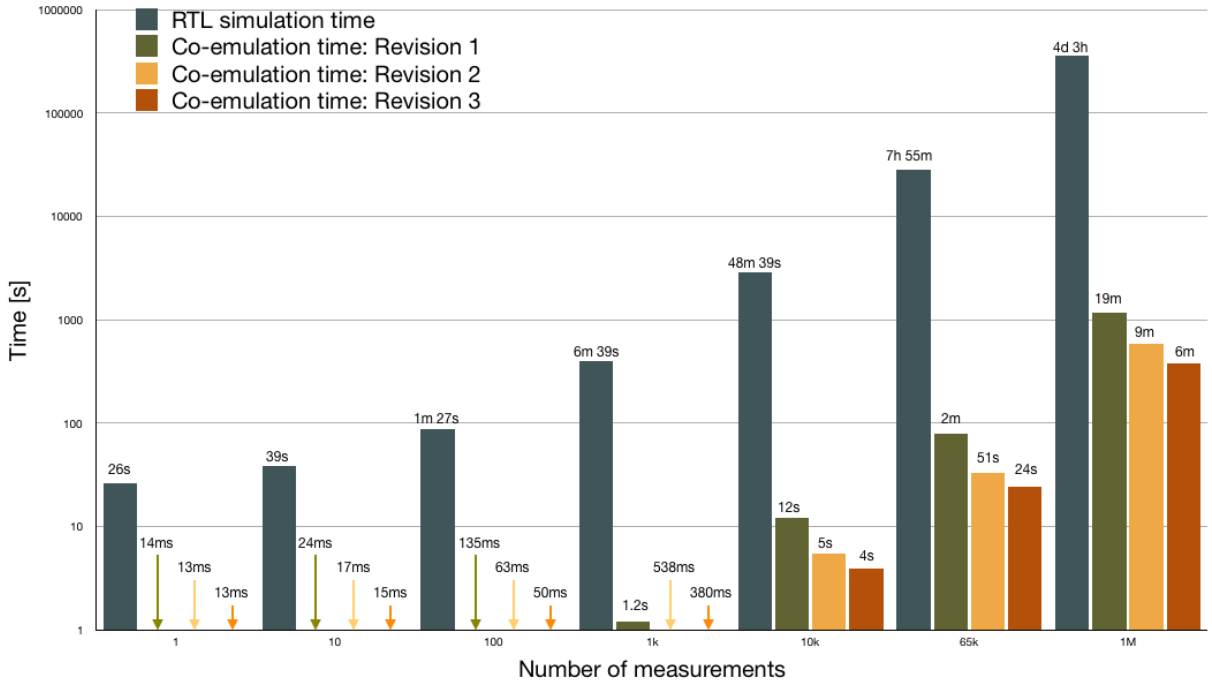


Figure 6.2: Comparison between three Test-bench versions

If we compare the timing results obtained by RTL simulation with the ones obtained by co-emulation with the third revision of the SW test-bench, we obtain much higher speed-up factors. They are shown in Table 6.4

Nr. measurements	RTL Sim time	TB acc time (Revision 3)	Speed-up factor
1	26s	12.5ms	2080x
10	39s	15.0ms	2597x
100	1m 27s	49.8ms	1746x
1k	6m 39s	379.9ms	1050x
10k	48m 39s	3s 889ms	751x
100k	~7h 55m for 65284	37s 519ms	/
1M	4d 3h 8m 46s	6m 19s	941x

Table 6.4: RTL simulation times vs. co-emulation time in Revision 3

The speed-up factors in Table 6.1 and 6.4 are respectively obtained by comparing revision 1 and revision 3 with the RTL simulation time. Their comparison among the listed factors is shown in the histogram in Figure 6.3.

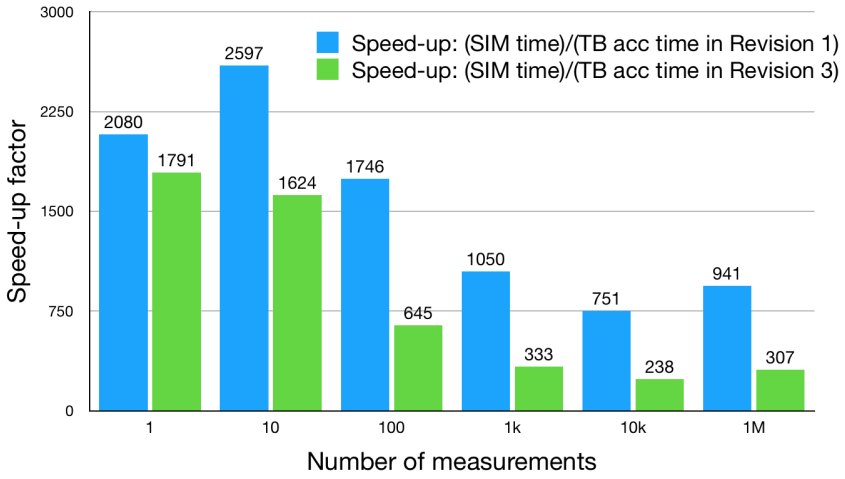


Figure 6.3: Speed-up factors comparison between Revision 1 and 3

For the current case study, acquiring a high number of measurement results is crucial for the verification of the DUT. Running one million of measurements can provide a very good analysis of the behavior of the DUT. Doing it with RTL simulation would require more than 4 days, while with co-emulation results are available after 6 minutes.

6.3 Co-emulation Timing Analysis

A timing analysis related to the operating mode of the co-emulation architecture has been carried out from Revision 3. The main idea was to evaluate the impact of the transactor's clock on the overall performance. The timings analyzed in the previous section are related to the architecture with the transactor domain running at 48MHz. One of the first idea to speed-up the verification would be to increase the frequency of the clock signal of this domain. For this purpose, the architecture has been modified and tested with the following transactor's clock frequencies: 75MHz, 100MHz, 125MHz and 150MHz.
Theoretically, increasing the transactor's clock frequency, the overall time should decrease. However, this did not happen: for each set of measurements, changing the transactor's speed the overall time changed of irrelevant factors. All the results are shown in Table 6.5.

Number of measurements	Transactor's clock speed: 48 MHz	75 MHz	100 MHz	125 MHz	150 MHz
1	12.5ms	11.7ms	11.5ms	11.8ms	11.7ms
10	15.0ms	14.9ms	14.8ms	15.1ms	15.96ms
100	49.8ms	49.3ms	52.1ms	50.3ms	54.9ms
1k	379.9ms	388.2ms	384.4ms	384.0ms	380.3ms
10k	3s 889ms	3s 822ms	3s 809ms	3s 920ms	3s 827ms
100k	37s 519ms	37s 716ms	38s 111m	37s 777ms	38s 304ms
1M	6m 19s	6m 24s	6m 13s	6m 10s	6m 11s

Table 6.5: Co-emulation times in Revision 3 with increasing transactor's clock speed

Speeding-up the hardware and not seeing timing improvements means that the time needed by the software to send a packet might be longer than the time needed by the hardware structure to receive it and send it back. Then, making the control interface faster would not improve the overall communication.
Another factor that impacts on the overall timing performance is given by the contribution of the OS. To see the impact of this, 100 consecutive test cases have been executed and their times have been collected. Revision 3 has been used, with the transactor's clock running at 48MHz on the hardware side. In each test case, 10k measurements have been collected. The obtained results have been plotted in a graph shown in Figure 6.4. The average value was equal to 3.34835s.

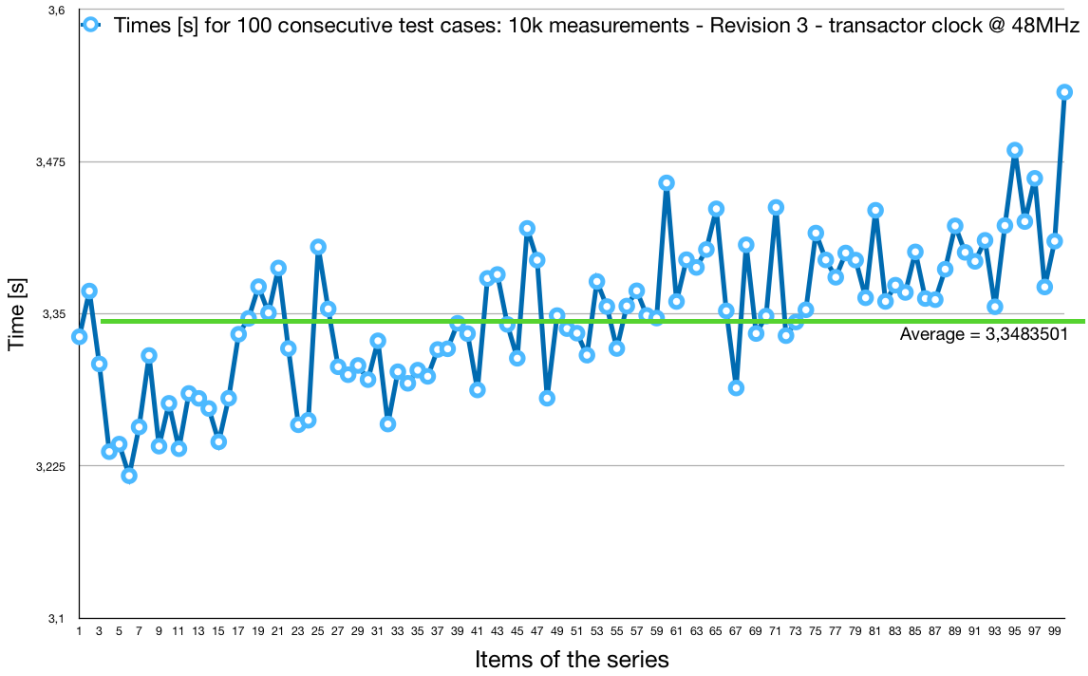


Figure 6.4: Execution times of 100 test cases for 10k measurements each - transactor clock frequency = 75MHz

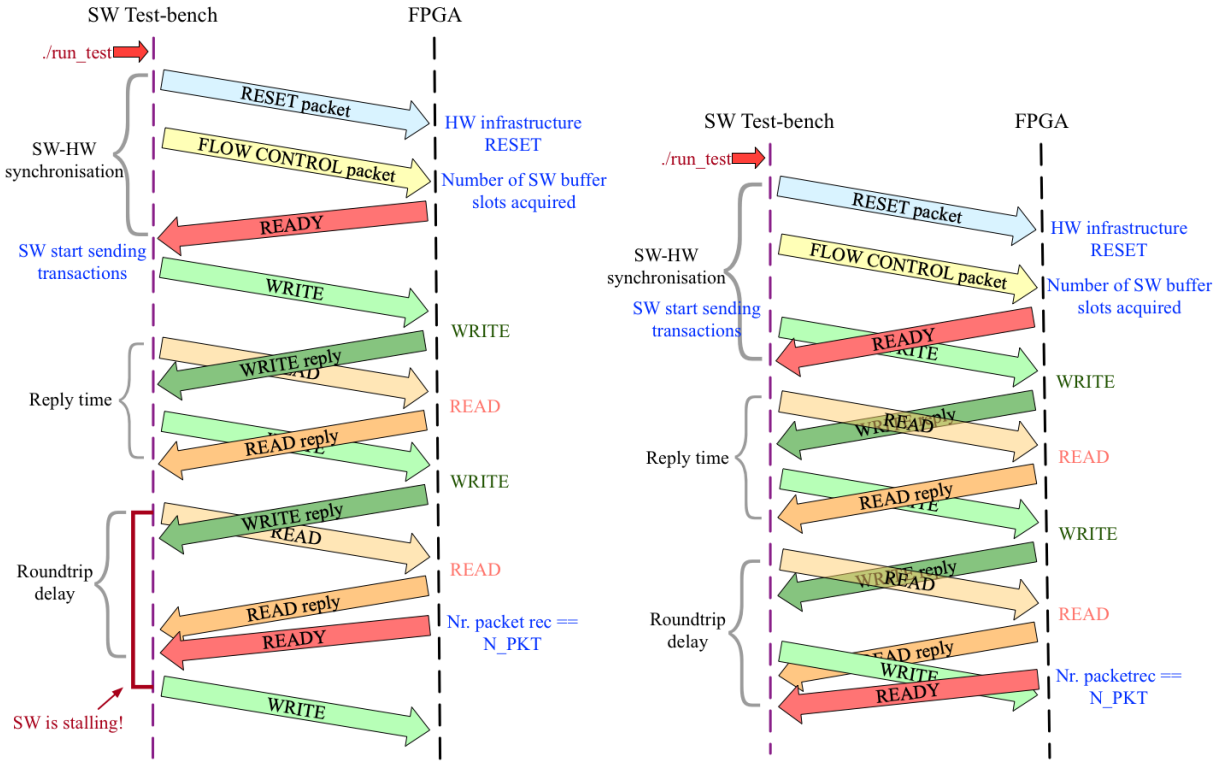
Taking into account the recorded values, having

- T_{max} = 3.53211s
- T_{min} = 3.21701s
- $T_{average}$ = 3.3483501s

the deviation of the average value from the maximum one is around 5.5%, while the one from the minimum value is around 4%.

6.3.1 Bottleneck

To understand which was the bottleneck of the HW-SW communication, the Test-bench has been executed running the Python Profiler. The test case in account was performing 100k measurements with the transactor's clock frequency at 48MHz.
Over a total execution time of 39.952 seconds, 34.387 where used for thread synchronization. This was due to the structure of the Python UVM Test-bench. The SW driver was implemented as a thread, which required synchronization for every read and write instruction. Its structure has been changed to avoid this, allowing to reduce the overhead due to thread synchronization.
Another factor that was slowing down the communication was related to the roundtrip delay introduced by the SW-HW synchronization with the READY message. The software was indeed stalling, waiting to receive the READY message in order to start sending again.



(a) Communication with SW stalling waiting READY (b) Communication with improved performance

Figure 6.5: SW-HW communication speed-up

This problem has been faced taking into account the following factors:

- Roundtrip delay ($t_{RTdelay}$): time between the READY message and the previous instruction sent by the software
- Reply time ($t_{HWreply}$): time between the instruction sent by the software and the response from the hardware

$$n = \frac{(t_{RTdelay})}{(t_{HWreply})} \tag{6.1}$$

where n is equal to the number of packets that can be additionally sent by the SW before checking the READY message queue.
With these improvements it was possible to reduce the overall time for the 100k measurements test case from 39.952 seconds to 33.009725 seconds.
However a higher performance improvement was expected. Let's consider the absolute time spent by a Sce-Mi frame to propagate over the entire hardware architecture. It can be divided in the following contributions:

- N_{RX}: number of cycles needed from the Receive Avalon Streaming interface on the TSE IP to the clock domain crossing of the Driver's FIFO
- N_{xtor}: number of cycles spent on the transactor domain, from the time in which the Driver reads a message from its XTOR_IN_FIFO, to the time in which the Monitor writes the message into its XTOR_OUT_FIFO
- N_{TX}: number of cycles needed from the clock domain crossing of the Monitor's FIFO to the Transmit Avalon Streaming interface on the TSE IP
- N_{SGMII}: number of cycles needed to transmit the Ethernet frame over the serial line at 1.25Gbps

Looking at the simulation waveforms, $N_{RX} = 79$, $N_{xtor} = 5$, $N_{TX} = 72$. About the serial transmission, having a Ethernet frame equal to 78 bytes as in our case, $N_{SGMII} = 78x8 = 624$. Having the following clock periods

- $T_{clk\_IF} = 8ns$: clock period of the interface clock running at 125MHz
- $T_{clk\_xtor} = 20.83ns$: clock period of the transactor's clock running at 48MHz
- $T_{clk\_SGMII} = 0.8ns$: clock period of the serial Ethernet clock running at 1.25GHz

the overall time for a Sce-Mi message can be computed as:

$$T_{HW} = (2 * N_{SGMII} * T_{clk\_SGMII}) + (N_{RX} * T_{clk\_IF}) + (N_{xtor} * T_{clk\_xtor}) + (N_{TX} * T_{clk\_IF}) \tag{6.2}$$

Replacing the above numbers:

$$T_{HW} = (2 * 624 * 0.8ns) + (79 * 8ns) + (5 * 20.83ns) + (72 * 8ns) = 2.31us \tag{6.3}$$

However, looking at the timings of the packet transmission, it has been noticed that the time between an instruction (WRITE or READ) from the software and the reply from the hardware is around 20-30us. This value is more than 10 times higher than T_{HW}.
The time spent by a packet inside the TSE IP is unknown and should be added to T_{HW}, but it should most likely be in the order of nanoseconds as for the other time contributions in the hardware domain. From the obtained numbers, the main assumption is that there still is some bottleneck on the communication.

6.4 Further improvements

As highlighted before, even after the latest improvements, a bottleneck in the communication is still present. It prevents to reach a higher speed and performance. To have a better understanding of where the bottleneck of the communication is, it would be needed to place some probes all over the hardware architecture and compute the exact times at which packets are transmitted. This kind of computation should not be done on RTL simulation, but using a logic analyzer (SignalTap) over the execution of the test case in co-emulation.
On top of this, other improvements can be applied. The first one would be to extend the depth of the transactor's FIFOs. This would allow to send the READY message back to the software a lower number of times. In the current implementation, the READY message is sent from the FPGA after every 4 instructions.
Let's consider a test case of N measurements. The total number of transactions generated by the SW Test-bench, considering also the start-up sequence, is equal to

$$n_{\text{transactions}} = 19 + (2 * N) \tag{6.4}$$

The number of READY messages generated is then equal to

$$n_{\text{READY}} = \frac{(n_{\text{transactions}})}{4} \tag{6.5}$$

In a test case of 100k measurements, 50.004 READY messages are generated.
Increasing the number of counted packets from 4 to 20, for example, would reduce the number of generated READY messages of 1/5.
Another important improvement is related to the packet structure. In the current implementation, the frame has been composed in the same way suggested by the standard: 4 bytes for the TRANSACTOR_ID field and 8 bytes for the TIMESTAMP field.
Having 32 available bits for the TRANSACTOR_ID means that 4294967296 transactors could be supported by the infrastructure. However, this seems to be an unfeasible case. A field of 1 byte would actually be more than enough to support the verification of any complex IP.
Also the TIMESTAMP field is unused. Over 8 bytes, only the last 3 are used as an identification field for the messages dedicated to the transactors.
Since the Ethernet communication is serial, having a lower number of bits to be transmitted is important if a speed-up is needed.
Further improvements in the communication speed could be achieved by replacing the Ethernet connection with a USB or PCIe one. However, implementing the communication protocol in pure hardware would not be so easy in this case. The adoption of a Nios processor would then be needed to take care of the communication protocol. Adopting a Nios would also reduce the Flow control effort on the hardware infrastructure itself.

Chapter 7

Conclusions & Perspectives

The obtained speed-up factors are approximately between 750x and 2000x, which represents a very good result. In the analyzed papers(Chapter 3), the maximum reached speed-up factor was related to the implementation developed by Hassoun, Kudlugi, Pryor and Selvidge [HKPS05]. It was equal to 320x. In our Test-bench acceleration, the minimum obtained speed-up factor is equal to 751x, more than two times higher.
What has been obtained from the developed Test-bench acceleration methodology was actually higher than what was expected when this research project began. It could have been expected to obtain a performance improvement higher than 320x, but still close.
Having, instead, performance improvements around 700-900x, Test-bench acceleration has been performed to run a test case of 100M measurements. Such scenario would be practically unfeasible in RTL simulation. For 1M measurements in RTL simulation, 4d 3h 8m 46s were required. Considering a linear increase, a test case of 100M measurements would have taken more than 400 days, so more than one year.
It has been tried to run 100M measurements with Test-bench acceleration. The execution has been killed after 37347530 measurements, within 1d 3h 30m 42s. The reason why the test case has been killed is unknown, but it could be due to the fact that the csv file containing the results had reached a maximum dimension. It has been tried to elaborate the results on Matlab, but the tool reported an error message stating the size exceeded the limits.
Even referring to 30M measurements, RTL simulation would have required more than 120 days (more than 3 months), while Test-bench acceleration provided the results in 1 day. This is a clear example of the fact that co-emulation would represent a solution to develop test cases that would be impossible to reproduce in RTL simulation.
Test-bench acceleration represents an optimal solution to save time in the verification process. Of course the bring up phase of the architecture required an initial demanding effort. However, having designed it in a standard and general way, no modifications will be needed on it to be used for the verification of new IPs. What would be needed is the design of the required transactors and their connection to the DUT and the infrastructure itself.
The Test-bench acceleration methodology provided in this project represents an initial version on top of which other improvements can be applied. Some of them have been discussed in section 6.4, others may be highlighted by its adoption for the verification of other IPs.
Co-emulation cannot, of course, entirely replace RTL simulation. However, according to

the verified DUT and the test case that must be executed, Test-bench acceleration may or may not represent a valid option.
This project's case study was, indeed, the perfect fit for Test-bench acceleration. Having just one control interface, only two transactors were designed. At the same time, its test cases required several hours to reach completion, which became seconds or minutes in co-emulation.
Looking at the obtained timing results for this project's test case and at its repartition provided by the Python Profiler, almost half of the overall time was spent on the *send* method of the socket. Let's then compare the obtained timing with Figure 3.3. Taking into account the test case for 1M of measurement and its speed-up factor equal to 941x, the co-emulation time is only 0.106% of the RTL simulation time. Half of them is used by the API. The remaining is mostly used in the Test-bench software domain.

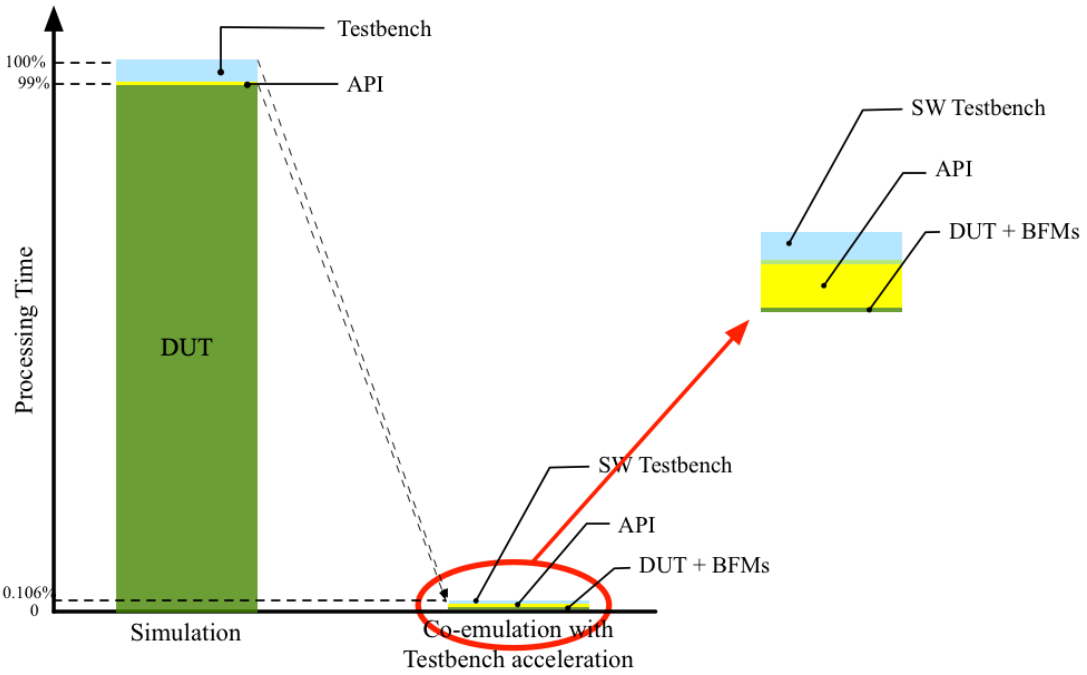


Figure 7.1: Processing times in RTL simulation and Test-bench acceleration for 1M measurements

The designed infrastructure is parametric, scalable and ready to use for the verification of other IPs. Being able to easily perform Test-bench acceleration for all the IPs would not only reduce the verification time, but also allow to think about test case scenarios which would have been infeasible in simulation.

Bibliography

[20119] "Medium-dependent interface". *Wikipedia*, 2019.

[Acc11a] Accellera. *"Universal Verification Methodology (UVM) 1.1 Class Reference"*, June 2011.

[Acc11b] Accellera. *"Universal Verification Methodology (UVM) 1.1 User Guide"*, May 2011.

[Acc16] Accelera. *"Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual"*, November 2016.

[Ald] Aldec. "Riviera-PRO Advanced Verification Platform".

[Alt03] Altera. *"FLEX 10K"*, 4.2 edition, January 2003.

[AS16] Mohamed AbdElSalam and Ashraf Salem. "SoC verification platforms using HW emulation and co-modeling Testbench technologies". *IEEE*, February 2016.

[AV19] Pranav Ashar and Vinod Viswanath. "Closing the Verification Gap with Static Sign-off". *IEEE*, April 2019.

[Ayn] John Aynsley. "A practical guide to OVM Part 1". Technical report, Mentor Graphics.

[BB03] Arnold Berger and Michael Barr. "Introduction to On-Chip Debug". *embedded.com*, February 2003.

[Bis19] Priyabrata Biswas. "Introduction to FPGA and its Architecture". *towardsdatasience.com*, November 2019.

[Dou15] Doulos. *"UVM Adopter Class"*, 2.3 edition, 2015.

[ea12] U. Farooq et al. *"Tree-Based Heterogeneous FPGA Architectures"*. Springer, 2012.

[Eth20] "Ethernet flow control". *Wikipedia*, 2020.

[Fai08] Gorry Fairhurst. *"The User Datagram Protocol (UDP)"*, November 2008.

[Fos10] Harry Foster. "Redefining Verification Performance (Part 2)". *Mentor*, August 2010.

[Fra98] Jon Frain. "1000BASE-X Physical Coding Sublayer (PCS) and Physical Medium Attachment (PMA) Tutorial". June 1998.

[FVe] "Formal verification". *Tech Design Forum*.

[Got06] Bob Gottlieb. *"Advances in Electronic Testing: Challenges and Methodologies"*. Springer, 2006.

[Har17] Ed Harmoush. "Traditional ARP". *practicalnetworking.net*, January 2017.

[HKPS05] S. Hassoun, M. Kudlugi, D. Pryor, and C. Selvidge. "A transaction-based unified architecture for simulation and emulation". *IEEE*, February 2005.

[HWa20] "Hardware acceleration". *Wikipedia*, 2020.

[Int18] Intel. *"Intel Arria 10 Device Overview"*, June 2018.

[Int19a] Intel. *"Avalon Interface Specifications"*, 2019.

[Int19b] Intel. *"Intel Arria 10 Device Datasheet"*, June 2019.

[Int19c] Intel. *"Triple-Speed Ethernet Intel FPGA IP User Guide"*, 2019.

[Joh02] Howard Johnson. "Random and Deterministic Jitter". *sigcon.com*, 2002.

[Mar13] Marvell. *"88E1111 Product Brief"*, October 2013.

[Mur] Robert Murphy. *"USB 101: An Introduction to Universal Serial Bus 2.0"*. Cypress.

[Neu05] Ralf Neuhaus. "A Beginner's Guide to Ethernet 802.3". 2005.

[PCI] "Down to the TLP: How PCI express devices talk (Part I)".

[PGM11] Chandrasekhar Poorna, Varun Gupta, and Raj Mathur. "Transaction-Based Acceleration - Strong Ammunition In Any Verification Arsenal". *deepchip.com*, January 2011.

[Qui11] Liam Quin. *"Extensible Markup Language (XML)"*, 2011.

[Riz14] Lauro Rizzati. "Hardware Emulation: A Weapon of Mass Verification". *Electronic Design*, October 2014.

[TJY14] Bill Jason Tomas, Yingtao Jiang, and Mei Yang. "SoC Scan-Chain verification utilizing FPGA-based emulation platform and SCE-MI interface". *IEEE*, November 2014.

[Tom] Bill Jason P. Tomas. "sce-mi for soc verification". *aldec.com*.

[vdSS17] Hans van der Schoot and John Stickley. "A brief history of hardware-assisted verification: from the ICE age to today's UVM transactors". 2017.

[vdSY15] Hans van der Schoot and Ahmed Yehia. "UVM and Emulation: How to Get Your Ultimate Testbench Acceleration Speed-up". *DVCON*, 2015.

[Wal12] Colin Walls. "In-Circuit Emulators". *ScienceDirect*, 2012.

[WBW01] Remigiusz Wišniewski, Arkadiusz Bukowiec, and Marek Węgrzyn. "Benefits of hardware accelerated simulation". June 2001.

[WDBP11] Wlodzimierz Wrona, Pawel Duc, Lukasz Barcik, and Wojciech Pietrasina. "An example of DISPLAY-CTRL IP Component verification in SCE-MI based emulation platform". *IEEE*, May 2011.

[Yas11] Muhammad Yasir. "Introduction to FPGA Technology". *FPGARELATED.com*, May 2011.