Tampere University

Jarkko Passi

VERIFICATION OF A HETEROGENEOUS MULTI-PROCESSOR SOC

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Prof. Timo Hämäläinen
Antti Rautakoura, M.Sc.
August 2023

ABSTRACT

Jarkko Passi: Verification of a heterogeneous multi-processor SoC
Master of Science Thesis
Tampere University
Master's Programme in Electrical Engineering
August 2023

System-on-chip (SoC) designs are getting more and more complex due to the constantly evolving semiconductor business. A single SoC can consist of a great number of sub-blocks such as CPUs, AI accelerators, memories, and interconnects. A heterogeneous structure enables the use of different kinds of processing units on the same chip. Resulting in greatly improved performance and power efficiency compared to homogeneous designs.

Verification is done to guarantee that a chip design is functional and ready for fabrication. More complex designs can be seen in verification as more time-consumed and as a need for more resources. The increasing complexity of the system greatly increases the number of configuration combinations which makes it a challenge for verification to cover all possible scenarios.

This thesis presents the verification process of Ballast SoC. Ballast is a heterogenous multiprocessor SoC developed by SoC Hub. Firstly, the thesis studied SoCs and verification in general. In addition, related work was explored. Secondly, the thesis outlines the strategy and implementation for the Ballast verification process. The strategy section shows how the planning was done and what methods were used. The implementation section outlines the practical implementation of the verification. Finally, the results are presented.

The results presented in the thesis prove that the tapeout of Ballast SoC was reached with a high level of confidence. Later the Ballast samples arrived and the wake-up of the chip was started. Ballast was proven to be functional and only one major issue was found which affected only one of the nine subsystems.

Keywords: SoC, MPSoC, verification, RISC-V

TIIVISTELMÄ

Jarkko Passi: Heterogeenisen moniprosessori-järjestelmäpiirin verifiointi
Diplomityö
Tampereen yliopisto
Sähkötekniikan DI-ohjelma
Elokuu 2023

Järjestelmäpiirit ovat käymässä entistä monimutkaisemmiksi jatkuvasti kehittyvän puolijohdeteollisuuden vuoksi. Yksi järjestelmäpiiri voi koostua suuresta määrästä erilaisia alilohkoja, kuten prosessoreista, tekoälykiihdyttimistä, muisteista ja väyläliitännöistä. Heterogeeninen rakenne mahdollistaa erilaisten prosessori-yksikköjen käytön samassa piirissä, mikä johtaa huomattavasti parempaan suorituskykyyn ja pienempään virrakulutukseen verrattuna homogeeniseen rakenteeseen.

Verifiointi suoritetaan, jotta varmistutaan järjestelmäpiirin toiminnasta ja, että voidaan siirtyä piirin valmistukseen. Monimutkaisemmat järjestelmäpiirit lisäävät verifiointiin tarvittavaa aikaa ja resurssien tarvetta. Järjestelmän kasvaessa erilaisten konfiguraatioiden määrä kasvaa, mikä asettaa haasteita verifioinnille, että saadaan katettua kaikki mahdolliset skenaariot.

Tämä diplomityö esittää Ballast järjestelmäpiirin verifiointiprosessin. Ballast on heterogeeninen moniprosessori-järjestelmäpiiri, joka on kehitetty SoC Hub projektissa, Tampereen yliopistossa. Ensimmäiseksi, työssä käydää läpi yleiset asiat liittyen järjestelmäpiireihin ja verifiointiin. Lisäksi tutustuaan verifiointiin samankaltaisissa projekteissa. Toiseksi, työ esittää strategian ja toteutuksen Ballastin verifiointiprosessille. Startegia osio esittää, miten suunnittelu tehtiin ja mitä menetelmiä käytettiin. Toteutusosio kuvaa miten verifiointi tehtiin käytännössä.

Työn tulokset osoittavat, että Ballast-järjestelmäpiirin verifointi suoritettiin onnistuneesti ja voitiin siirtyä valmistukseen hyvällä luottamuksella. Myöhemmin piirit saapuivat valmistuksesta ja Ballastia alettiin testaamaan. Ballast osoittautui toimivaksi ja ainoastaan yksi merkittävä ongelma löytyi, mikä vaikutti vain yhteen yhdeksästä alijärjestelmästä.

Avainsanat: järjestelmäpiiri, verifiointi, RISC-V

PREFACE

This thesis was written as part of SoC Hub project hosted by Tampere University.

Thanks to my thesis examiners Prof. Timo Hämäläinen and M.Sc Antti Rautakoura for guidance during the writing process. SoC Hub was a great environment to work in. For that, I would like to thank the entire Ballast development team for making this environment possible.

Tampere, 13th August 2023

Jarkko Passi

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

AI	Artificial Intelligence
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
C2C	Chip-tp-Chip
CI	Contineus Integration
CPU	Central Processing Unit
DMA	Direct Memory Access
DSP	Digital Signal Processor
DUT	Design Under Test
FLL	Frequency-Locked Loop
FPGA	Field Programmable Gate Array
HPC	High Performance Computing
HW	Hardware
IO	Input-Output
IP	Intellectual Property
JTAG	Joint Test Action Group
MAC	Media Access Control
MPC	Medium Performance Computing
PLL	Phase-Locked Loop
RGMII	Reduced Gigabit Media-Independent Interface
ROM	Read-Only Memory
RTL	Register Transfer Layer
SDIO	Secure Digital Input Output
SOC	System-On-Chip
SPI	Serial Peripheral Interface
SW	Software
TB	Testbench

UVM	Universal Verification Methdology
VIP	Verification Intellectual Property

1. INTRODUCTION

The constant need for more powerful and efficient electronics drives the development of more complex SoC designs. SoCs enable us to integrate a vast amount of functionality on a single chip. Nowadays SoCs power everything from wearables to high-performance computing systems. A key element of modern SoCs is a heterogeneous design integrating different types of processing units on the same chip to increase performance and energy efficiency.

The increasing complexity creates new challenges for the whole SoC design flow including design, verification, and validation. This has driven the industry to use standardized methodologies and complex tools to achieve functional SoCs. Verification is commonly the most time-consuming part of the SoC flow and only increases as the SoC complexity increases. Heterogeneous architecture leads to tailored verification approaches as the different blocks need to function together which becomes a big challenge in the verification process.

The goal of this thesis is to achieve the required confidence in the design so that it is feasible to move to tapeout and fabrication of the SoC. The first part of the thesis focuses on the relevant background by giving an introduction to SoCs and verification. Chapter 2 gives a general overview of SoCs and more detailed information about heterogeneous SoCs. Chapter 3 studies verification in general and what needs to be taken into consideration when verifying an SoC. In addition, the general verification flow and different approaches and methodologies are explored in this chapter. Chapter 4 explores related work and how verification is implemented in other projects.

The second part of the thesis focuses entirely on Ballast architecture, Verification strategy, verification implementation, and results. Chapter 5 gives the first introduction to Ballast SoC and goes through the structure in detail describing each subsystem. Chapter 6 presents the Ballast verification strategy including the used methods, plans, and areas that were focused. Chapter 7 documents the practical implementation of the verification, divided into processor, communication, and processing subsystems. In addition, top level and FPGA implementations are presented. Chapter 8 shows the results of the verification process to prove that the goal was achieved. Finally, chapter 9 includes the conclusions of the thesis.

2. SYSTEM-ON-A-CHIP

This chapter provides relevant background regarding SoCs, their structure, challenges, and things to consider.

2.1 SoC overview

System-on-a-chip is a complex digital device. It is an integrated circuit that implements an entire electronic system. What makes them complex is that the whole system is implemented on a single chip. That can include for example processing units, busses and interconnects, I/O, and memories. [1] The SoC that this thesis revolves around is a multi-processor SoC (MPSoC) containing multiple processing units.

The SoC structure contains multiple layers. System-level, also called top level, includes multiple subsystems and usually a bus interconnect to handle communication between the blocks. The next layer is the subsystem level which consists of smaller blocks which are called intellectual properties (IP). The layered structure continues similarly. Figure 2.1 contains a general block diagram of a multi-processor SoC.

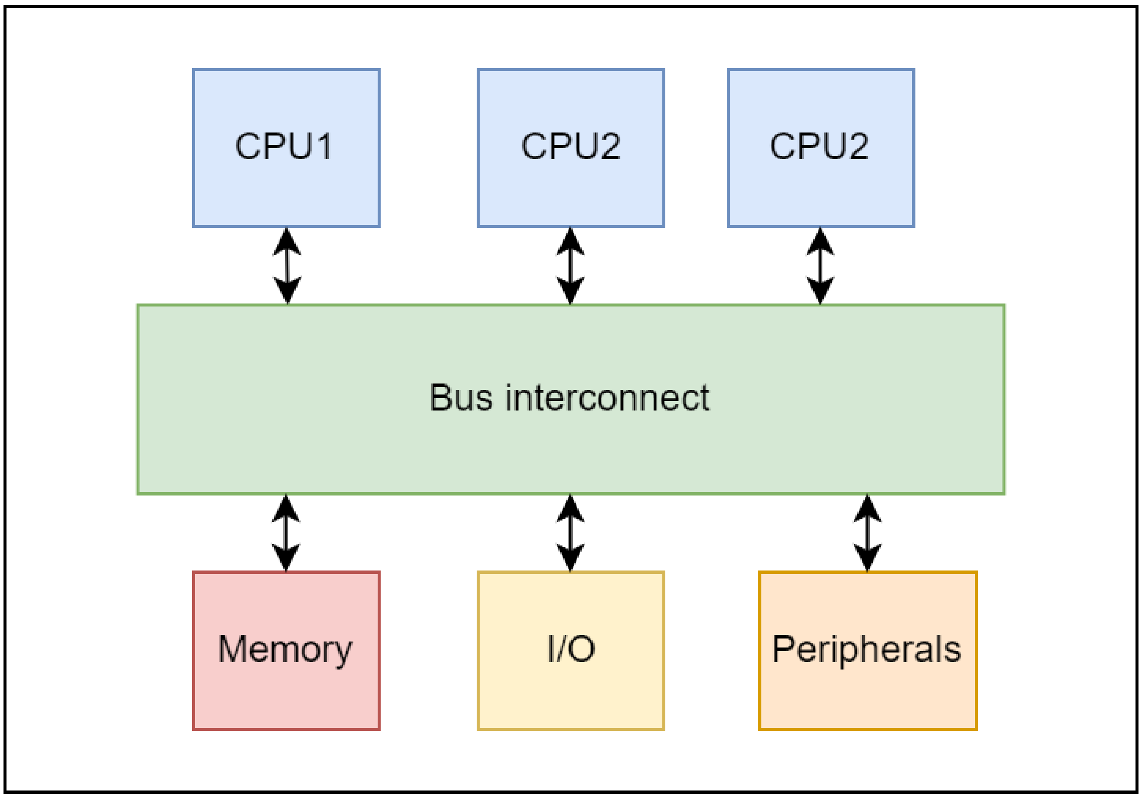


Figure 2.1. *General SoC block diagram*

2.2 Heterogeneous multi-processor SoCs

Modern SoCs are often heterogeneous, which means they have multiple processing subsystems with different architectures. These can include for example CPU subsystems, AI subsystems, and DSP subsystems. The motivation is to enable parallel processing, increase performance, lower power consumption, and reduce die area. Performance and low power consumption are achieved by using specialized processing units for specific tasks. For example, a CPU can do AI processing but does it with significantly lower efficiency than an AI accelerator would that is designed specifically for the task. The structure of such heterogeneous SoC is described later in section 5.

Heterogeneous SoC trades some flexibility for other benefits such as mentioned above. Heterogeneous SoCs are tailored to specific use cases to increase efficiency but in the process specialize the usage for a limited amount of applications. In reality, this is not an issue because heterogeneous SoCs are designed to serve a specific purpose. Such specific use cases can be for example an SoC for a mobile phone or for a baseband station.

On the other hand, the heterogeneous approach increases flexibility by making it possible to develop a wide range of different applications. The opposite is a homogeneous system that uses multiple identical processing subsystems, for example, general-purpose CPUs. Homogeneous chips offer great flexibility while reducing efficiency, making them less ideal for embedded applications such as mobile phones requiring high performance with low energy consumption. To meet the demanding requirements of a modern SoC such as high performance and low power consumption, the heterogeneous design approach is the feasible way. [12]

3. VERIFICATION

This chapter provides background information about SoC verification, different verification approaches, and methodologies. The chapter explains the content of the verification flow and the challenges included.

3.1 Verification overview

Design verification is about making sure that the hardware fulfills the requirements and functions according to the specification. However, this is not enough. In addition to that the verification should be thorough enough, to prove that the design works in unexpected scenarios and it behaves as expected in all circumstances. Bugs will be found and fixed during the process. Figure 3.1 illustrates how the verification process sits in the whole SoC project timeline. Verification activities are started at the same time with design to achieve fluent testing from the ground up.

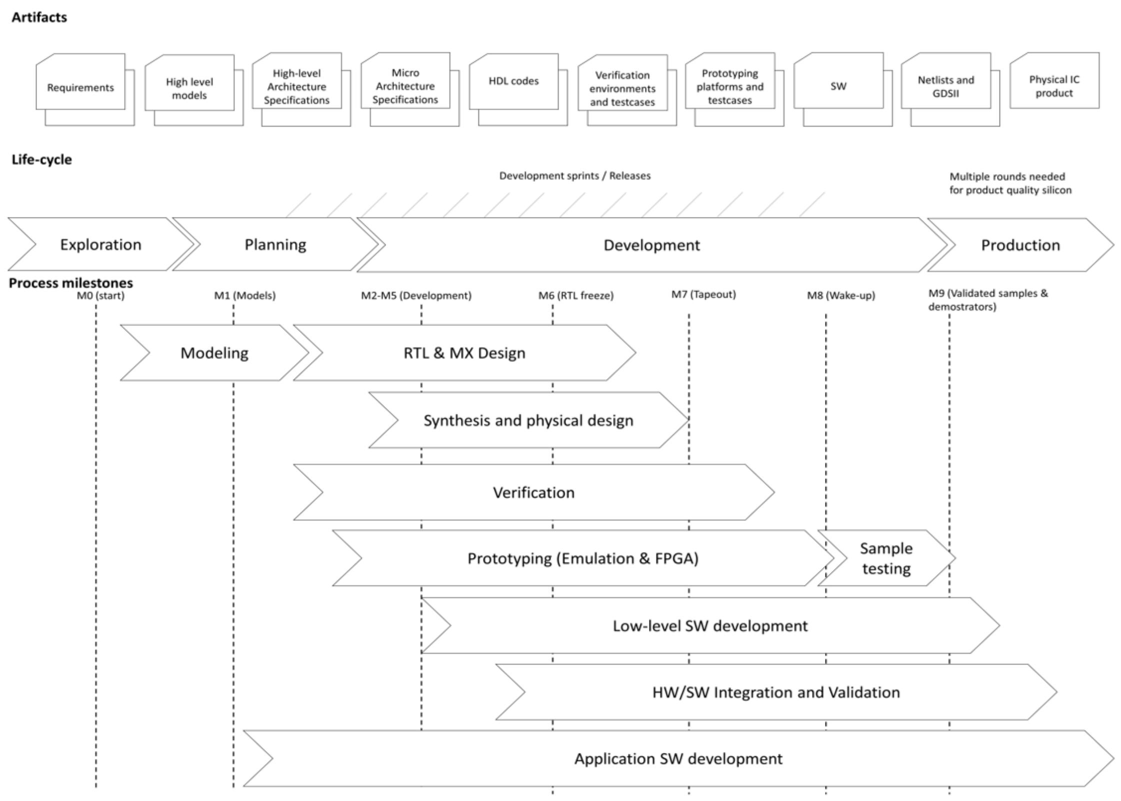

Artifacts
Requirements
High level models
High-level Architecture Specifications
Micro Architecture Specifications
HDL codes
Verification environments and testcases
Prototyping platforms and testcases
SW
Netlists and GDSII
Physical IC product
Life-cycle
Development sprints / Releases
Multiple rounds needed for product quality silicon
Exploration
Planning
Development
Production
Process milestones
M0 (start)
M1 (Models)
M2-M5 (Development)
M6 (RTL freeze)
M7 (Tapeout)
M8 (Wake-up)
M9 (Validated samples & demostrators)
Modeling
RTL & MX Design
Synthesis and physical design
Verification
Prototyping (Emulation & FPGA)
Sample testing
Low-level SW development
HW/SW Integration and Validation
Application SW development

Figure 3.1. *SoC flow diagram [23]*

3.2 Verifying a system-on-a-chip

Verifying an SoC is a challenging task due to the complexity and size of the design. Complexity naturally increases as the size of the design expands. When an SoC contains several subsystems with heterogeneous processing blocks that require concurrent functionality, the verification effort needed increases substantially. In the end, a lot of resources are needed to complete the work within the tight project deadlines.

The challenge can be seen in the time it takes to close the verification. As seen in figure 3.1, verification takes a notable space in the diagram. In addition to that, prototyping and a lot of the SW development and HW/SW integration also contribute to the verification. Thus in reality verification has even more ground than what can be quickly observed from the diagram.

Because of the layered structure of the SoC, also the verification is performed in multiple layers. An SoC consists of subsystems, subsystems include smaller IPs, and so on. Verification is done on each hierarchy level. First, the IPs are verified before integrating them into a subsystem. Subsystems are then verified before integrating them into the system level also called as top level. Test case re-use is important because it can save a lot of time and it is often possible to re-use test cases from lower levels to higher levels. The figure 3.2 shows an example of a subsystem structure in Ballast SoC.

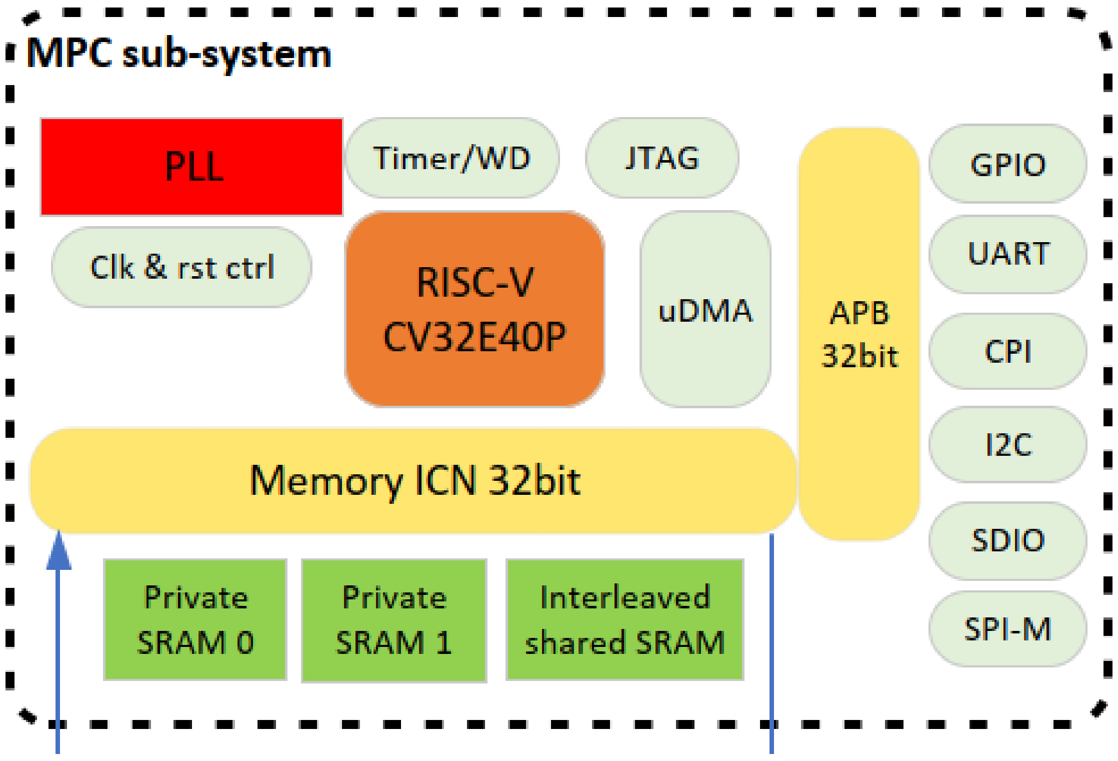


Figure 3.2. *Example block diagram of a subsystem in Ballast [24]*

Top level verification's primary focus is integrating the subsystems and verifying the connectivity between them. Top level verification can be done in multiple ways depending on the SoC architecture. If the SoC has CPU subsystems, at least part of the top level verification is done with HW-SW co-simulation by running SW on a capable subsystem. In other cases for example verification IPs used to verify subsystems can be re-used to verify the top level. Verification IP is an external component usually residing in the testbench and can interact with the design under test (DUT) while verifying its functionality. The whole process is described in more detail in Ballast verification chapters 6 and 7.

3.3 Functional verification

The goal of functional verification is to verify, that the design is implemented according to specification. Functional verification is done by reviewing the specification and producing test cases to prove the specification is met. The process contains iteration back to RTL design to get to the point where the design matches the specification. As shown in diagram 3.1, design, and verification go hand in hand for most of the design cycle. It is fairly easy if to prove that some functionality is not correct if a discrepancy is found. However, it is difficult to tell that the specification is met in the end, for example, if the specification was not understood correctly. Luckily coverage metrics help to guarantee that every part of the design is verified. [2] Coverage is explained in more detail in a later section.

Functional verification can be divided into three different approaches: black-box, white-box, and grey-box verification. These approaches describe the transparency of the DUT. In black-box verification, the internal architecture of the design is not visible to the verification engineer. In this approach, the engineer relies entirely on the interfaces of the design. Stimulus is applied to the inputs of the design and the output is observed to determine if the functionality is correct. The benefit of the black-box approach is that the internal implementation doesn't matter. The same black-box testbench can be utilized if the implementation changes. However, it lacks the detailed low-level information needed in nowadays complex designs. White-box approach is the opposite. It relies entirely on the internal implementation of the design and has full visibility. Assertions are an ideal way to do white-box verification. Bugs can be caught at the lowest level of implementation leading straight to the root cause. The issue here is that often when the internal implementation changes, the white-box testbench also needs to be changed accordingly. [2]

Grey-box approach is a compromise between the alternatives above. Combining the lack of detail in the black-box approach and the lack of portability in the white-box approach. It retains the approach of only working with the interfaces of the design but targets implementation-specific parts of the design. A common use case for the grey-box approach is to patch coverage holes during the verification process. Either black-box or gray-box verification needs to be used if the verification is executed in parallel with the RTL implementation. Since white-box verification is dependent on the exact final implementation. [2]

3.4 Methodologies

3.4.1 UVM

Universal verification methodology (UVM) is an open-source verification framework that focuses on re-usability. It's based on classes that enable re-usability and makes it highly configurable. Being class-based, it inherits the pros that object-oriented programming provides. [16] It is especially useful when verifying a design that has interfaces that use a certain protocol. If one creates a UVM agent to drive such an interface, one can re-use the same agent in the current and future projects using the same interface protocols.

UVM provides the UVM class library which holds the pre-made classes. These classes are called UVM components. [16] An example of a UVM testbench is always built using the same set of basic components. The user molds the components to fit the architecture of the DUT.

The basic UVM hierarchy consists of multiple components. The top level is the testbench which instantiates UVM test class, DUT, and the virtual interfaces connecting these together. Under the UVM test, there is a UVM environment, which instantiates a UVM scoreboard, responsible for correctness checking and one or more UVM agents. UVM agent is a UVM-based verification IP (VIP). The agent instantiates the components responsible for the interface handling which is protocol specific. These components are a UVM sequencer, UVM driver, and UVM monitor. UVM sequencer is only responsible for feeding the driver. UVM driver takes the higher level transaction items and converts them to signal level, feeding the interface connected to DUT. UVM monitor is also connected to the same interface. As the name suggests, the UVM monitor monitors the interface by having access to the data within the interface. It converts the signal level data back to transaction items and can feed those forward to the scoreboard for example. [16] Figure 3.3 illustrates the hierarchy of a traditional UVM testbench.

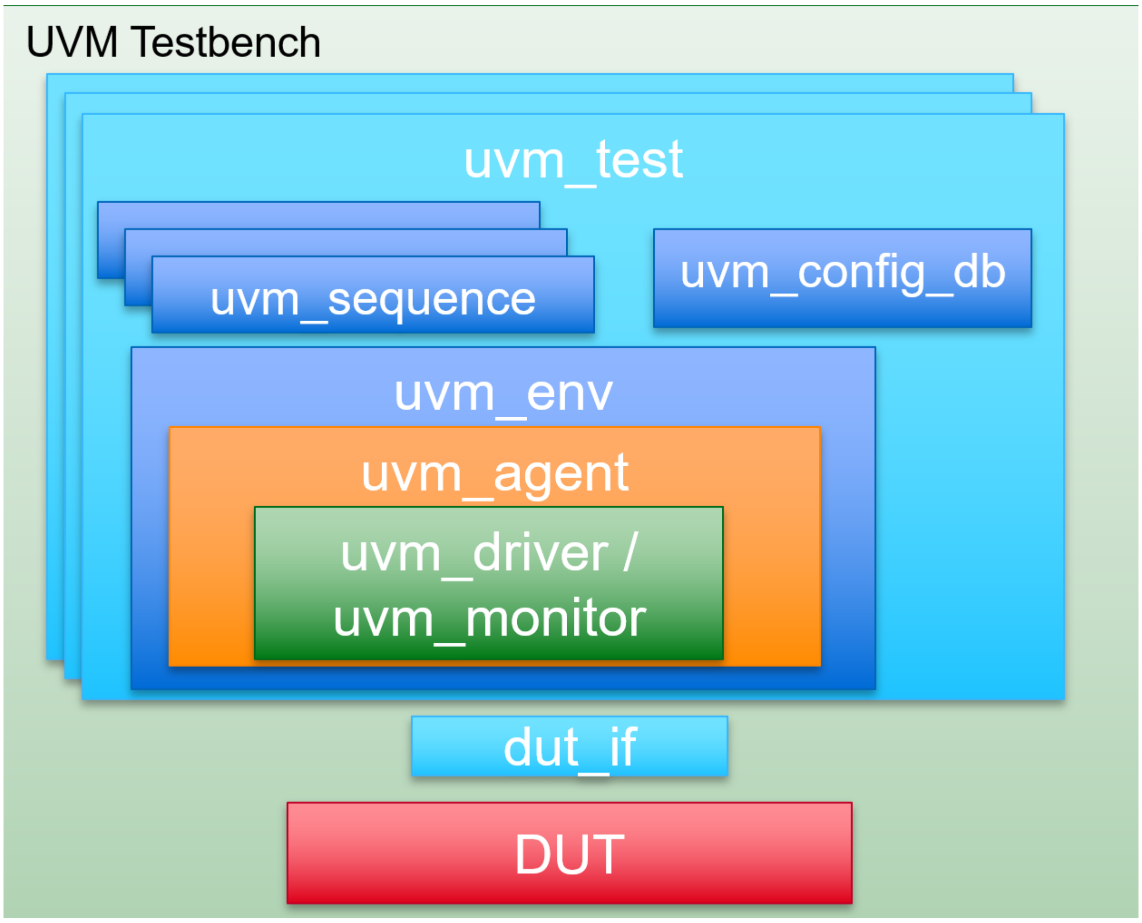


Figure 3.3. *An example of a traditional UVM testbench environment [19]*

Because UVM is open-source, it opens the possibility to re-use work from open-source projects. For example, open-source UVM agents can be found and reused. It can save huge amounts of time but also brings in the possible downsides of the open-source world. Often providing insufficient documentation and the use of these components can become more difficult and take more time than expected. Caution needs to be taken because verifying blocks with faulty VIPs can result in a faulty design.

3.4.2 FPGA prototyping

FPGA is a silicon chip containing millions of re-programmable digital blocks. This architecture allows almost any digital design to be implemented on FPGA. It provides a reusable platform for digital design applications. There are several drawbacks and pros when comparing FPGA to ASIC. The most important difference is that through reconfigurability, FPGA provides excellent flexibility. The most notable drawback is the performance. FPGA can't achieve as fast clock frequencies as ASIC which decreases the performance. For example, the FPGA clock frequency can be 1-quarter of the clock frequency on ASIC but there is variance between applications. [26]

In ASIC SoC projects, FPGAs can be used for verification and software development. Those can also happen at the same time through HW-SW co-simulation described in the next section. FPGA setup can take a long time because the design meant for ASIC often requires changes to implement it on FPGA. After setting up, FPGA prototyping is a great tool for speeding up certain verification tasks and software development. In addition, it enables the use of external physical components and tools to be used with the design on FPGA. For example, external debugger, communicating via peripherals such as UART to PC, and to monitor interfaces with tools such as oscilloscopes.

3.4.3 HW-SW co-simulation

In the hardware-software (HW-SW) co-simulation method, hardware, and software are verified concurrently. For this method, the design must include a processor subsystem capable of running software. The general flow of this method contains writing software code, compiling it for the target platform, loading it into the hardware memory, and running the simulation which starts the program execution from the memory. Co-simulation makes it possible for software engineers to run and debug their code on the simulated hardware. It offers visibility to the code execution process that other methods are unavailable to deliver. For example, FPGA prototyping of the same system provides the same software running capability but sacrifices visibility for speed. Speed is a common issue with co-simulation as it can be very slow to run. This of course is not suitable when developing software as you could be waiting for a day to be able to test a small code modification. However, it is also useful from a hardware point of view as the hardware gets verified as well in the process. This method is not optimal for IP level verification but for top level verification, it is needed to cover the integration. [28] [13]

3.4.4 Assertions

Assertions are low-level correctness checks integrated into the design during the design and verification processes. They are usually implemented when a block is designed but can also be added during verification and become useful at any point in the verification process. Basically, assertions set rules that even other blocks accessing the block in question need to follow. They are useful for checking specific things for example verifying an interface protocol. Assertions are based on checking transitions or states of individual signals. It can for example reveal if a signal is not changing its state according to specification. Assertion coverage can be checked during coverage analysis to spot critical assertions not being hit. During simulations, assertions can reveal illegal operations and can stop the simulation accordingly and/or report the violation to the user.

3.5 Verification flow

The verification flow includes strategy planning, verification plan development, executing the plan, and analyzing results. The flow is not straightforward and there is iteration involved e.g. adding new tests to the plan after analyzing the results. These activities also overlap and someone can be still developing the plan when the execution is already started and vice versa. The verification strategy is the starting point. It defines the used verification requirements, methods, goals, and debug methods. Based on the strategy, the verification plan is created which defines the test cases and methods to be used for each test case.

Verification is time-consuming and as the designs grow more complex, the time to verify it increases. Verification can take even 70% of the total IC design cycle [29]. Poor planning will definitely result in more time spent overall.

The verification plan defines all the tests that are to be completed. It also should determine how the test is executed, for example, RTL simulation or FPGA. The plan is used to track the progress of the test cases. Meaning indication if the test is e.g. not started, failing, or passing. Also, a detailed description of the tests should be included. Test cases are defined based on requirements and use cases coming from the higher level. Progress can be tracked with coverage methods described later.

3.6 Bugs and debugging

A lot of bugs are found during verification. Finding bugs should get rarer and rarer when closing the tapeout. The most obvious ones are caught in the beginning and more complex ones can be found as late as in gate level simulation (GLS) closer to tapeout. The bug itself might not be more complex but if it is found during use case tests or GLS, it can be more difficult to find due to a more complex environment. Unfortunately, any bug can be critical for the SoC and that's why the areas that can most likely cause a critical failure are focused most during verification.

Debugging is a critical part of the process. When a fault is observed, it is critical to find the root cause as quickly as possible. Transparency is an important aspect of founding the issue, visibility of a black-boxed design is just not sufficient unless the cause is obvious from the higher level which rarely is the case. In each verification method, the debugging methods might differ. However, the general principle is the same.

There isn't only one way to debug. Each person approaches the problem differently. The general approach is to look at what failed, how it failed, and then why it failed. For example in simulation, the method is to analyze the simulation log and waveform produced by the simulator. The printed simulation log gives higher-level information that can point in the correct direction. It requires that the test case and testbench environment provide sufficient information in the form of prints. When the general direction of the issue is discovered, it's time to move to the waveform to analyze the problem in more detail. Verification engineers work together with design engineers to discover the root cause, especially in more complex cases. It requires the verification engineer to provide sufficient information to the design team so that they can reproduce the same error.

3.7 Coverage

For measuring the progress of verification, verification coverage is an important tool. Verification coverage uses coverage data that can be extracted from the simulator tools. The data is commonly exported into an automatically generated report. The report shows the coverage percentage, which indicates how thoroughly the design has been verified. Engineers analyze this and use it to find areas where the coverage is lacking. Based on the information, more tests can be created to target specific areas. The coverage goal should always be 100% meaning that every tiny bit of the design is covered. However, sometimes achieving 100% is not practically possible. In this case, careful thinking is required to determine when the coverage result is good enough. There are different types of coverage metrics used to track different areas.

Code coverage often called implementation coverage is simply a measure of RTL code lines executed during verification tests. It is automatically collected which makes it a simple tool to use. Code coverage quickly exposes which areas of the designs are not exercised. It can be used to spot such areas which might be caused by faulty test implementation or a missed block during the verification planning. [17]

Functional coverage is a more specific metric and it can be used to track covered features and configurations. For example, the engineer is aiming to verify a block while covering all the different configuration modes. Here functional coverage is useful to know when everything required is covered. Compared to code coverage, functional coverage gives more detailed information and also isn't automatically collected. The engineer needs to specify in detail what is monitored in the testbench environment. This metric is most important on the block level because it's much more efficient to get the needed coverage on the standalone environment versus a more complex top level setup. [17]

Test-based coverage in its simplicity is tracking the test cases from the verification plan. Once all test cases are covered, test-based coverage is 100%. [17] Test-based coverage on its own is not sufficient as it does not provide detailed information on the test case quality. However, it is an important metric among others to track the verification progress.

3.7.1 Git CI

Git continuous integration pipeline (CI) [6] automates building and testing the design. Based on configuration the pipeline runs the complete build flow and all the test cases for thorough regression testing. Every time a new commit is pushed, the pipeline checks that everything is still working.

Automatic testing provides many benefits. The most significant benefit is that the design stays sane in the main branch and the possibility for human error is decreased. For example, a developer pushes changes that possibly compromise the design and doesn't run the test cases to confirm that. Another developer finishes their work and realizes that the design is no longer functional. In the end, a lot of time can be wasted looking for the root cause. With the CI pipeline in place, time can be saved when every commit is verified by the regression test run. The usual flow is that a development branch is merged with the main branch. In that case, the pipeline makes sure that everything is passing before the merge is accepted, thus keeping the main branch safe.

4. RELATED WORK

This section explores verification work done in other projects. The main point is to get an idea about how verification is implemented in related projects.

OpenHW group is a global organization involving its members and individual contributors. OpenHW specializes in open-source RISC-V cores and related areas including for example software. [8] Their projects include multiple RISC-V cores targeting platforms such as embedded, FPGA, and application class with full Linux support. [7]

CORE-V-VERIF is OpenHW group's verification project for verifying the RISC-V cores. It is advertised as an industrial-level verification environment. They use a thorough set of verification methods including simulation, FPGA prototyping, emulation, and formal. The verification environment for all cores is based on UVM which goes hand in hand with their claims about industry-level verification and they also mention one of the benefits of UVM to be industry standard. [8] Their verification planning is standard based on a verification strategy document and a verification plan. The verification plan approach is explained in verification plan 101 [9].

OpenTitan is an open-source silicon Root of Trust (RoT) project. The goal is to improve silicon RoT design and implementation transparency and trustworthiness. It is a collaborative project to produce capable and high-quality IP. [20] OpenTitan offers thorough documentation of the verification process.

OpenTitan pursues to achieve industry-strength verification. The goal is to achieve the quality required for a full-production silicon chip tapeout. For this purpose, the project uses UVM as the main verification platform. [21]

As the main verification documentation, OpenTitan has a testplan to track test case development and functional coverage. DV document is used for higher-level documentation such as strategy, goal, testbench structure, and used VIPs. They state the environment structure is created in such a way that it enables high reusability. OpenTitan defines key focus areas for verification that are common across different DUTs. Those range from sanity tests to stress tests. To achieve coverage closure, OpenTitan uses coverage collection to progress toward 100% coverage for all applicable verification metrics. [21]

5. BALLAST SOC

This chapter focuses on Ballast SoC structure and architectural overview. The following sections contain top level structure and descriptions of the included subsystems.

5.1 Overview

Ballast is an MPSoC. Multiple processor subsystems and processing units provide a competent platform for numerous applications. These applications include for example deep-learning-based machine vision object detection and IoT applications.

Ballast is developed in the cross-organization project SoC-hub, hosted by Tampere University.

5.2 Top level structure

Ballast's top level structure includes 9 different subsystems. They are the System control subsystem, Medium-performance computing subsystem (MPC), High-performance computing subsystem (HPC), Top level peripheral subsystem (TLP), Chip-to-chip interface subsystem (C2C), NVDLA-based AI subsystem, Ethernet subsystem, AamuDSP co-processor subsystem, and Interconnect subsystem. The Ballast structure is illustrated in figure 5.1.

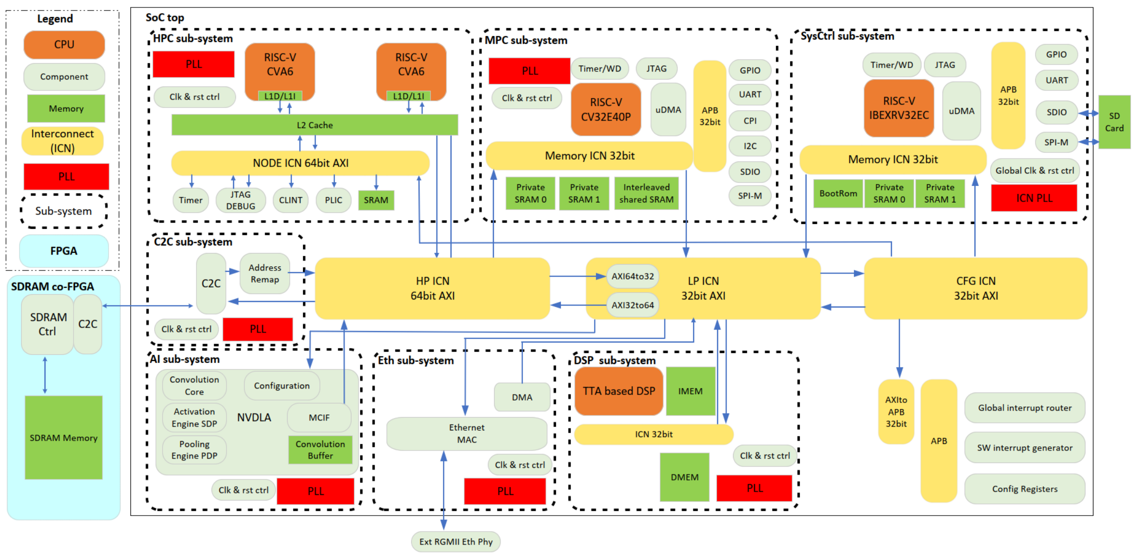


Figure 5.1. *Ballast block diagram [24]*

Interconnect subsystem makes subsystem to subsystem communication possible using AXI protocol. Off-chip communication is mainly possible through the C2C interface and ethernet. The top level instantiates the interconnect subsystem and all the subsystem wrappers and connects them together. The ballast top level has interfaces for off-chip communications and control signals.

The structure consists of template-based subsystem wrappers. Each subsystem resides in its own wrapper. Subsystem wrapper instantiates the subsystem top level, CDC components for clock domain crossing, possible converters, subsystem clock control block, and a phase-locked loop (PLL) block capable of generating high-speed clock frequencies. The wrapper also has the connections to connect the components together. The interface of the subsystem consists of the CDC component interfaces and possible control signals e.g. clock controls. The use of a template-based structure makes top level integration cleaner and less time-consuming when each subsystem wrapper instance is similar. PLL provides the capability of generating a high-speed clock. Having configurable clocks creates multiple clock domains within the system which is why clock domain crossing components are needed when transitioning from one clock domain to another. The subsystem wrapper structure is illustrated in figure 5.2.

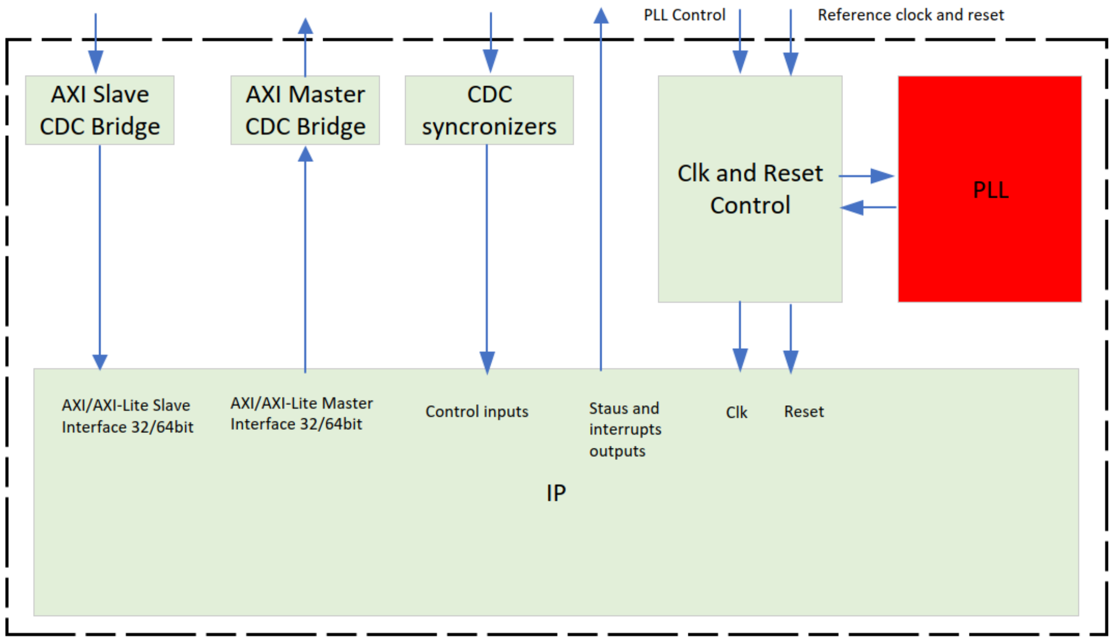

PLL Control
Reference clock and reset
AXI Slave CDC Bridge
AXI Master CDC Bridge
CDC syncronizers
Clk and Reset Control
PLL
AXI/AXI-Lite Slave Interface 32/64bit
AXI/AXI-Lite Master Interface 32/64bit
Control inputs
Staus and interrupts outputs
Clk
Reset
IP

Figure 5.2. *Subsystem wrapper overview [24]*

5.3 Subsystems

This section contains the descriptions of the subsystems in detail.

System control subsystem

The system control subsystem is based on the Pulpissimo microcontroller architecture by Pulp-platform. The heart of this subsystem is a single-core IBEX CPU[15]. IBEX core includes a 2-stage pipeline and support for the following extensions: base integer instruction set(I), and compressed instructions(C). It also includes configuration options for multiplication instruction set(M) and a reduced number of registers(E). [22]

System control subsystem's primary use case is to boot the SoC. Because of the limited use cases, Pulpissimo was stripped from excess peripherals to reduce the area and complexity of the subsystem. Boot options include SDIO and SPI interfaces to load a software image off an SD card, and JTAG to load the image through an external debugger

MPC

MPC is also based on the Pulpissimo microcontroller architecture by Pulp-platform. However, this subsystem is using a single RI5CY core. RI5CY core includes a 4-stage pipeline and support for the following extensions: base integer instruction set(I), compressed instructions(C), multiplication instruction set(M), and configuration option for single-precision floating-point instruction set. [22]

MPC is based on Pulpissimo with minimal changes. FLL clock generation is removed and a PLL is used instead for high-speed clock generation. All the interfaces remain as original. Sysctrl handles the boot procedure which includes loading a SW image to MPCs SRAM thus bootROM was removed.

HPC

High-performance computing subsystem HPC is based on the Ariane[14] cores. It utilizes two cores, a low-performance and a high-performance one. Ariane is based on RISC-V architecture and includes a 6-stage pipeline and implements I, M, A, and C extensions.

In addition to the cores, the subsystem includes an L2 cache, standard RISC-V peripherals, and a boot RAM. The L2 cache is 256kB in size and implements an 8-way functionality with 256b write-back. The peripherals include a timer, JTAG debug module, core local interruptor (CLINT), and platform-level interrupt controller (PLIC). The boot RAM is a small 32kB SRAM to be used to run simple software. As the main memory, HPC is designed to use external memory through C2C which is described next.

C2C

Chip-to-chip subsystem provides a data interface for off-chip communication. It has an AXI interface and it is used as a memory-mapped region for easy access. C2C converts AXI into a packet-based protocol that is capable of high-performance data transfer. Doing so decreases the number of signals needed significantly from the AXI protocol. There is no dependency between clock domains across the interface.

C2C includes multiple configuration options. Those include AXI data width configuration, physical interface width configuration, and configurable internal buffers. It supports the full range of AXI burst length, incremental and fixed bursts, and supports simultaneous requests.

Top level peripherals

Top level peripherals subsystem provides access to certain functionality for multiple subsystems. This functionality includes configuration registers and top level interrupts.

The configuration registers are EMA control registers for physical memory configuration, C2C address re-map registers, and pad configuration registers for Ethernet, C2C, and Sysctrl pads. Top level interrupt functionality contains top level interrupt routing and configurable SW interrupts to trigger interrupts between subsystems to allow simple communication in an efficient way.

NVDLA (Nvidia deep learning accelerator)

AI subsystem provides the capability of AI processing for different kinds of applications e.g. machine vision through MPC's camera interface. The subsystem houses the NVDLA which is an open-source AI accelerator, developed by NVIDIA. The main features of the accelerator are that it is scalable, configurable and it is designed to simplify integration [4].

Ethernet

The Ethernet subsystem provides ethernet connectivity for the Ballast SoC. It includes tri-mode ethernet MAC, based on IEEE802.3ab specification, and a DMA to transfer data between the subsystem AXI interface and the MAC. Tri-mode MAC supports 10Mbps, 100Mbps, and 1Gbps transfer speeds and offers full-duplex in 1Gbps mode.

AamuDSP subsystem

AamuDSP subsystem is a co-processor designed for Ballast. It houses a custom VLIW DSP core which is capable on many types of processing e.g. demosaic-ing, denoising, color mapping / white balancing, gamut mapping, tone map- ping and audio processing. On the top level, it communicates through AXI with the rest of the chip. [11]

ICN subsystem

Interconnect subsystem enables high-speed communication within the SoC. It's based on high-speed AXI protocol and built from open-source components provided by Pulp-platform. The key components are AXI crossbars, AXI converters, and clock-domain-crossing components.

ICN subsystem hierarchy has a top level wrapper that instantiates three AXI crossbars. High-performance crossbar, low-performance crossbar, and configuration crossbar. Having a lot of ports on single crossbar increases die area and consumes more power. The multi-crossbar structure is used to optimize it and also makes it possible to have different data-width AXI interfaces within the SoC. The high-performance crossbar uses a 64-bit data width while the other two are 32-bit. Two AXI converters make it possible to connect low-performance and high-performance crossbars together. One AXI upsizer and one downsizer are used. [5]

6. BALLAST VERIFICATION STRATEGY

This chapter goes through the verification strategy used in Ballast SoC verification.

6.1 Verification overview

Schedule and resources were taken into account when planning the verification. Many of the components and subsystems are open-source and some were already been in use in other projects. This means the majority of the IPs were already verified thus reducing the amount of verification work in this project. This enabled us to focus on the critical functionality instead of verifying everything. Verification progress is tracked by test plan coverage and achieving high coverage on other areas is not the goal. The difficulty of this approach is to define what should be verified. If high coverage is not the goal, it means some parts of the design will be left at least partly unverified in this project. It is unknown how thoroughly the re-used components have been verified previously. However, going for 100% coverage in such a complex system would require a lot of resources and time. The critical functionality is determined with the designers, the people responsible for the area in question, and the verification team. Based on this collaboration, the verification plans are constructed which are covered in the upcoming sections.

During the verification planning, a few critical areas were recognized. These key areas include top level, interconnect, and boot. Each of these areas is critical for the SoC to be functional. The strategy for each area is described in the upcoming sections.

6.2 Subsystems

The verification strategy for subsystems varies from subsystem to subsystem but the principle is the same. In general, the subsystems that were acquired from open-source as a whole came with a testbench and possibly test cases to do our own verification. In addition, such subsystems are expected to be verified to a certain degree. Despite that, all subsystems are verified in the project to some extent. For subsystems that are designed in the project or need a testbench, a suitable approach is taken into creating one. A general subsystem testbench using open-source UVM components is created to have a reusable testbench environment.

Subsystem verification effort is concentrated on critical areas such as areas that were modified after branching an open-source design. As an example, Sysctrl and MPC are based on the open-source Pulpissimo architecture but modifications were made to suit them better for our needs. During verification planning, those modifications are taken into account, and implementation is done accordingly.

6.3 Top level

The main focus in top level verification is to verify that these subsystems can access all parts of the system as specified and that the subsystems can work together. In Ballast, there are no limitations in access between subsystems and all subsystems with the capability to initiate write and read operations should be able to access all parts of the design. This includes subsystems with for example a CPU or a DMA. Accessing every possible part of the address map is not feasible within the project schedule, the strategy focuses on accessing all the memories in different subsystems and address boundaries of register locations. The top level environment is based on a SystemVerilog testbench which provides the needed components to interface the system and to initialize it. Top testbench structure is illustrated in figure 6.1.

When planning top level verification, the general rule is that at this point standalone functionality of subsystems should be verified. This enables us to focus only on subsystem integration and use cases on the top level. Top level verification relies completely on HW-SW co-simulation, which means that the subsystems capable of running software are used to exercise the design. In Ballast, those subsystems are Sysctrl, MPC, and HPC.

In addition, the top level adds the subsystem wrappers. The wrappers are not included in standalone subsystem verification. This introduces some risk to the top level which is taken into account during top level strategy planning. The connectivity and functionality of the wrapper components needs to be covered by the top level verification. Second thing to take into account in top level verification is that for some of the subsystems the AXI interfaces were not covered in subsystem verification and they will also rely completely on top level verification

Interconnect subsystem can be seen as a part of the top level. In theory, the top level integration tests and use cases would also cover the interconnect functionality. The issue relying only on top level verification in this case is that exercising the interconnect in top level can be fairly difficult because of the black-boxed nature of the top level. Also the stimulus applied to the interconnect is limited by the features of the processor subsystems. To solve this issue, a separate testbench is created for interconnect to verify it thoroughly. That way top level verification doesn't need to worry about interconnect functionality and can focus on integration verification alone. This is covered in the verification implementation section.

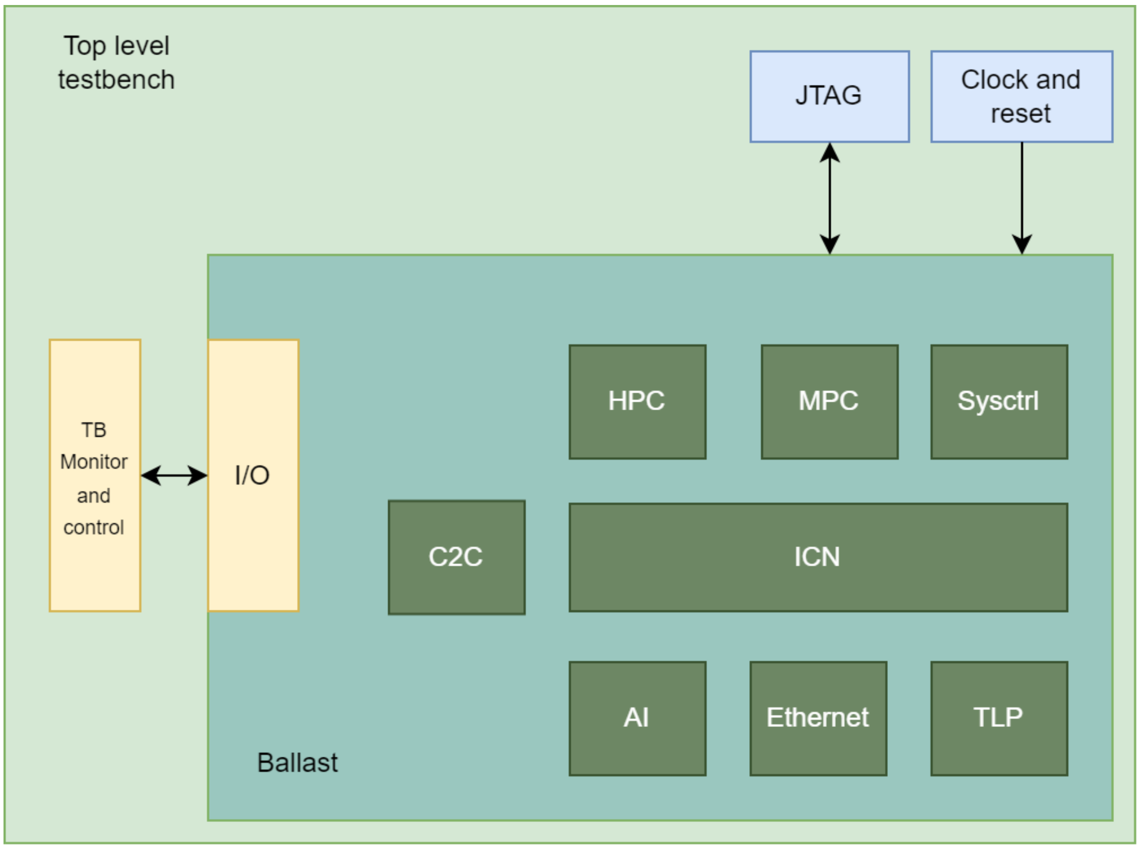


Figure 6.1. *Simplified image of top level testbench*

6.4 Boot

The boot is considered to be one of the most critical areas in verification. The goal of the boot verification is to guarantee the functionality of every possible boot option. In addition to guarantee that one failing option doesn't compromise others. Which is needed to have a resilient boot process. [18]

The methods for verifying the boot process are RTL simulation, gate level simulation, and FPGA prototyping. All the available methods were used to verify the functionality thoroughly. Actual implementation using these methods is described in chapter 7.

The autonomous boot process relies on the software executed from the bootROM. Being read-only-memory, the software binary is set during fabrication and can't be altered afterward. Thus it has to be verified to a point where we are sure it doesn't contain any bugs. Autonomous boot is one of the core features planned for the SoC. The verification steps go according to the following list:

1. RTL simulation. The verification process starts by using RTL simulation to get maximum visibility to the software execution. Most bugs needs to be get rid of during this step to make prototyping more fluid which is the next step. Testbench contains an SD card VIP with SDIO interface to emulate SD card behavior.

2. FPGA prototyping. With the use of an FPGA, the testing becomes much faster but loses a lot of the visibility that RTL simulation offers. Most important gain here is the moving to use the actual physical interfaces that are in use on the ASIC. Most importantly SD card contains software loaded by the bootROM and external debugger which provides access to the software execution.
3. Gate level simulation. GLS with annonated delays provides the most accurate platform simulating the ASIC. However simulating the boot in GLS takes a lot of time, thus it's used as a final step to check timing violations in the boot process. The downside being the loss of physical interfaces gained in prototyping,

SDIO boot is the primary autonomous boot method and is verified most thoroughly. The direct alternative is SPI boot which achieves the same results. SPI boot verification suffers from the lack of a simulation model emulating an SD card with an SPI interface. This leaves FPGA prototyping as the only possible verification method for it.

The third autonomous boot option is the external boot which can be entered by driving an external pin which affects the boot location address in hardware. The external boot went through all the above verification steps. This boot mode relies completely on hardware implementation. This means it is accessible even if bootROM is corrupted in the fabrication.

In addition to autonomous boot options, there is the JTAG interface to provide debugging capability while also offering the option to boot the SoC. JTAG boot is thoroughly covered in all of the boot verification steps. Gaining a high level of confidence in the process. In the simulation environment, there is the possibility to emulate the JTAG access. During prototyping an external debugger is used. [18]

6.5 Verification plans

Verification plans were created to plan the verification process beforehand and to track the verification progress. Verification plans were created for each subsystem and one for top level. The basic structure of the plans include connectivity tests, integration tests, use cases, and performance tests. An example verification plan structure is shown in the following table 6.1. The table has columns that give information about a test case name, description, and status. Status can be for example not started, failing, or passing. The FPGA column indicates if a test case is planned to be re-used in FPGA prototyping phase. The target milestone describes a deadline for each test case, indicating on which milestone the test case should be implemented and passing.

Connectivity tests are the simplest possible test cases to verify connectivity between IP blocks and connectivity to available memories, in practice sanity checks for connections. On the subsystem level that includes connectivity to possible peripherals and memories

Name	**Description**	**Status**	**Fpga yes/no**	**Target milestone**
Connectivity tests		Passing	No	HW1
Register tests	Access all registers	Passing	No	HW1
Memory tests	Access all memories	Passing	No	HW1
Integration tests		Passing	Yes	HW2
Use cases		Passing	No	HW4
Performance tests		Passing	Yes	HW5

Table 6.1. *Verification plan example*

inside the subsystem. On top level it includes connections between subsystems, for example accessing registers and memories through the top level interconnect. The purpose of these tests is to guarantee that the most basic functionality is working before more complicated test cases. By making sure the basic operations work first, we create a reliable platform for further test cases. It saves time by having simpler tests to debug in a case where some connections are not functional

Integration test cases are the next step after the connectivity tests are passing. Integration test cases are used to test top level functionality. Verifying that blocks that are supposed to work together, do really work. On subsystem level, an example could be a CPU subsystem using it's peripheral to verify peripheral connection out of top level. On top level an example could be any subsystem to access off-chip through C2C interface or managing external interrupts using Top level peripherals subsystem.

Use cases are again one step towards more complexity and shall be tested after all the related integration tests are passing. Use cases provide information about system functionality on a larger scale. It is up to the people planning the verification to decide, which tests are considered as integration tests and which are use case tests. In practice, the test can be defined as a use case if it's significantly more complex than other integration test cases. An example of a use case test could be utilizing Sysctrl to initialize AI subsystem to perform processing which would then output the processed data through C2C interface off-chip.

In addition to the test types above, there are performance tests. The purpose is not the test the functionality anymore, but to evaluate the performance. After the performance is measured, for example, how many transactions C2C can pass through in a certain amount of time which gives information if the required bandwidth is achieved, usually measured in Gbits/s. The results can be evaluated based on the requirements and specifications. The tests will guarantee that the requirements are met.

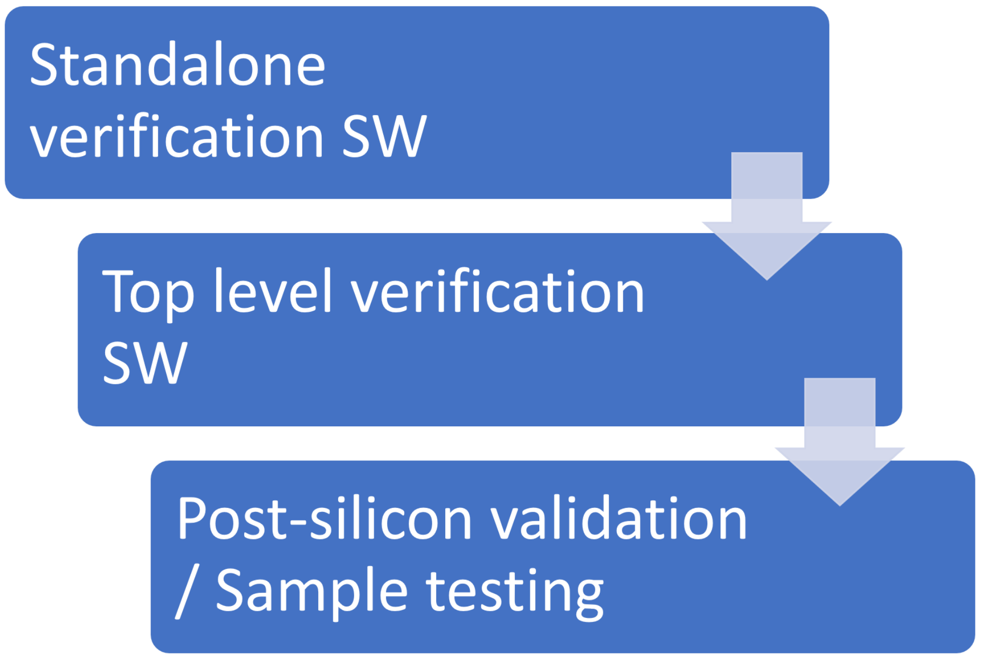

Standalone verification SW
Top level verification SW
Post-silicon validation / Sample testing

Figure 6.2. *Verification re-use*

6.6 Verification re-use

Ballast SoC includes three RISC-V processor subsystems. They are verified using hw/sw co-simulation with RTL simulation. While running software in RTL simulation is slow, a significant amount of time can be saved by re-using the software test cases from standalone level to top level.

In addition to re-using the test bottom-up, some of the tests can also be used between the processor subsystems with none to minimal modifications. Sysctrl and MPC are based on the same platform and most of the peripherals are identical. This enables to re-use the same peripheral test cases between these systems. In addition to re-using subsytem level test cases between the subsystems, also some of top level test cases can be re-used between them. For example, top level connectivity tests where the processor subsystems access memories and registers of the whole system. The tests only need minimal changes to addressing and possible data width changes to run similar tests on each processor subsystem. Re-use illustration in figure 6.2.

Another upside of the software-based test cases is to be able to re-use them again in sample testing. The test cases can be run on the ASIC to verify the functionality after fabrication. This strategy saves a lot of time during the sample testing activity.

6.7 Methods used

This section defines the methods used to verify the SoC. The methods are behavioral RTL simulation, GLS, and FPGA prototyping.

6.7.1 RTL simulation

RTL simulation is the main method for the Ballast SoC verification. With RTL simulation, a detailed analysis of the internal implementation is made possible. This allows us to verify the functionality and debug the design with full transparency.

RTL simulation is used for the entire design cycle from start of the design until the tapeout. It is used to verify the functionality of the design and it is the main way of debugging in fault cases.The simulator provides a GUI to check the status of each signal in the design.

Downsides of the RTL simulation is that it doesn’t take account the physical delays in the design which is why GLS is used and that is described in the next section. RTL simulation can also be very slow as the design complexity increases which makes it not optimal for example software development [3]. However in this project it is the main platform for testing bare metal software.

In many cases RTL simulation relies on simulation models that are not designed to be synthesised. These include most memories and mixed-signal designs. These models are used to model the functionality and in later stages of the cycle, they are replaced by the components that will be synthesised.

6.7.2 Gate level simulation

Gate level simulation relies on a synthesized netlist provided by the physical design team. It includes all the final synthesized components used in the design and RTL simulation models are replaced.

Gate level simulation is run with annotated delays that include information about flip-flop set-up and hold time requirements. It provides information if the timing is met or produces timing violations that need to be fixed before forwarding the design for fabrication.

GLS is incredibly slow compared to RTL simulation and it is not suitable for testing the full functionality of the system. Common areas to cover in GLS are memory functionality and other critical areas. On Ballast, the GLS is used to verify only the critical functionality for example the boot process.

6.7.3 FPGA prototyping

FPGA prototyping is planned to be used in critical areas where there is significant benefit from doing it. In this project, these areas include the boot process and interfacing external hardware.

Sysctrl is planned to be prototyped mainly to cover the boot process. This is described earlier in the boot section 6.4.

MPC is planned to be prototyped to cover the camera interface it provides. It is done by connecting an external camera to the interface. Then, the software is run on MPC to utilize the camera interface to control an external camera module.

C2C is planned to be verified on FPGA to cover the physical interface. For example two FPGAs with both having a C2C block to communicate across the interface between FPGAs.

7. BALLAST VERIFICATION IMPLEMENTATION

This chapter presents how the verification implementation was done.

7.1 Subsystem level

Overview of how subsystems were verified.

7.1.1 Processor subsystems

Ballast includes Sysctrl, MPC, and HPC as the processor subsystems. The verification process for each processor system is similar. They use pure SV (SystemVerilog) testbenches with the functionality to load software test cases into the design. By running different software, different parts of the design can be covered. A generic structure of a processor subsystem testbench can be seen in figure 7.1.

Sysctrl and MPC use identical testbench structures. The testbench generates a clock for the design and de-asserts the reset. JTAG is used to control the core execution and it also loads the software binary into the SRAM memory from which the core executes the software. The JTAG flow consists of halting the core, writing the boot address where the core starts the software execution, loading the software into the design, and resuming the core. After the JTAG interfacing is done, the testbench waits for CPU execution to end which can be seen via the JTAG interface. After the end of execution is seen, the simulation is stopped. HPC testbench differs in the software loading process. Instead of JTAG, the memory is preloaded before the simulation starts. The benefit of this approach is that it is faster than loading the binary using JTAG. The downside is that this approach can't be used with real ASIC and doesn't cover JTAG functionality which can be used with ASIC.

As described above, the verification is implemented through software. Software test cases are implemented with C programming language. The implementation of the test cases is defined by the verification plan. For each feature in the verification plan, a software test case is created and that's how the verification was completed on the subsystem level.

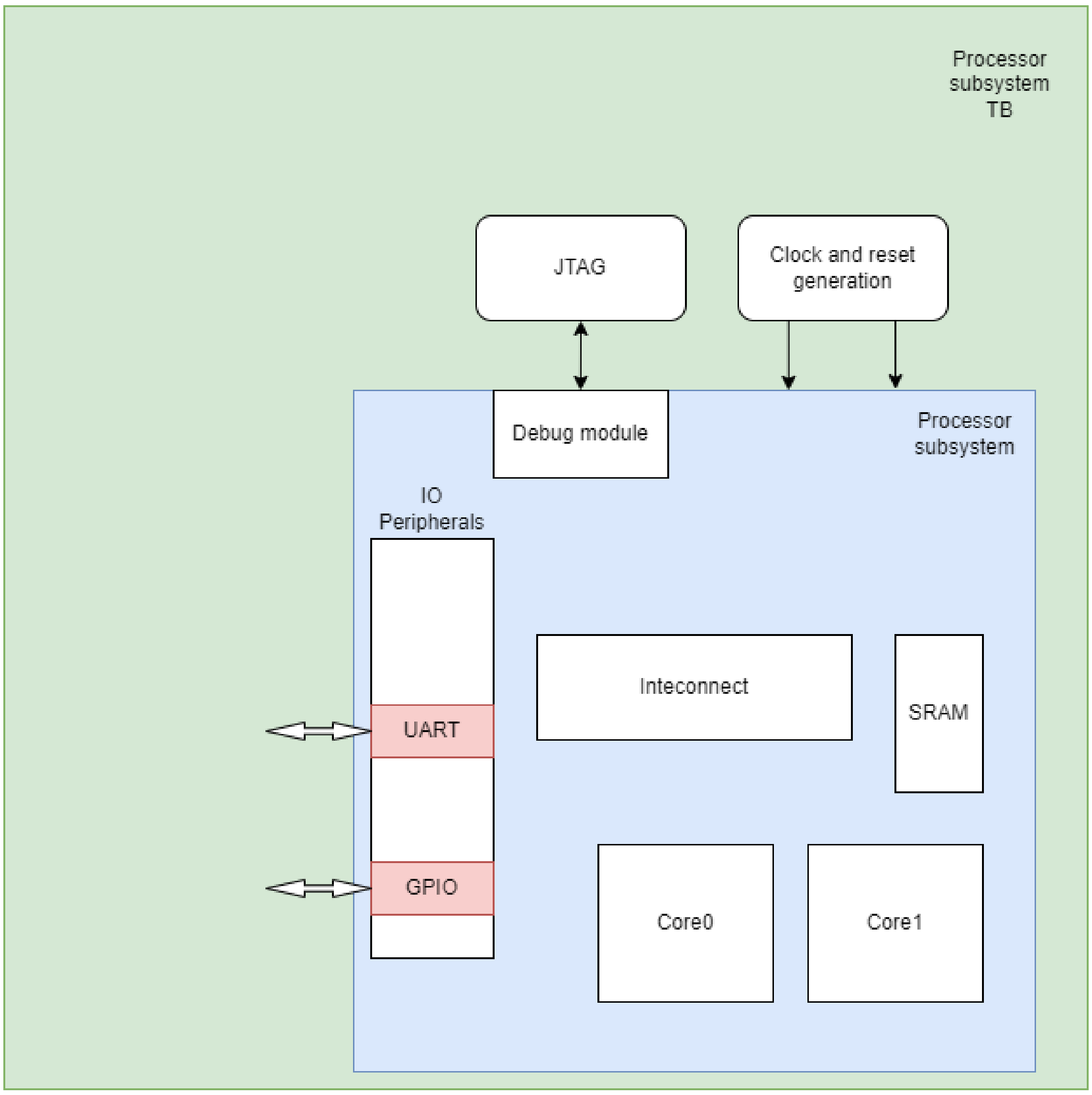


Figure 7.1. *Processor subsystem testbench block diagram*

7.1.2 Communication subsystems

Ballast includes three subsystems with the main purpose of communicating outside the SoC or between subsystems. These are ICN, C2C, and Ethernet. Verification of these subsystems is implemented by thoroughly covering different AXI transaction types that they support. For that purpose, an open-source UVM environment was used. TVIP-AXI includes AXI master and slave UVM agents with very detailed configurability [10]. Figure 7.2 illustrates the structure of this UVM testbench.

The implementation relies on random constrained verification, which randomizes the AXI transactions fed by the master agent. By randomization, different types of transactions can be verified with minimal configuration which leads to thorough coverage. On the other side of the design is the AXI slave agent which is responsible for capturing the transaction and saving the data of write transactions to an internal memory of the agent.

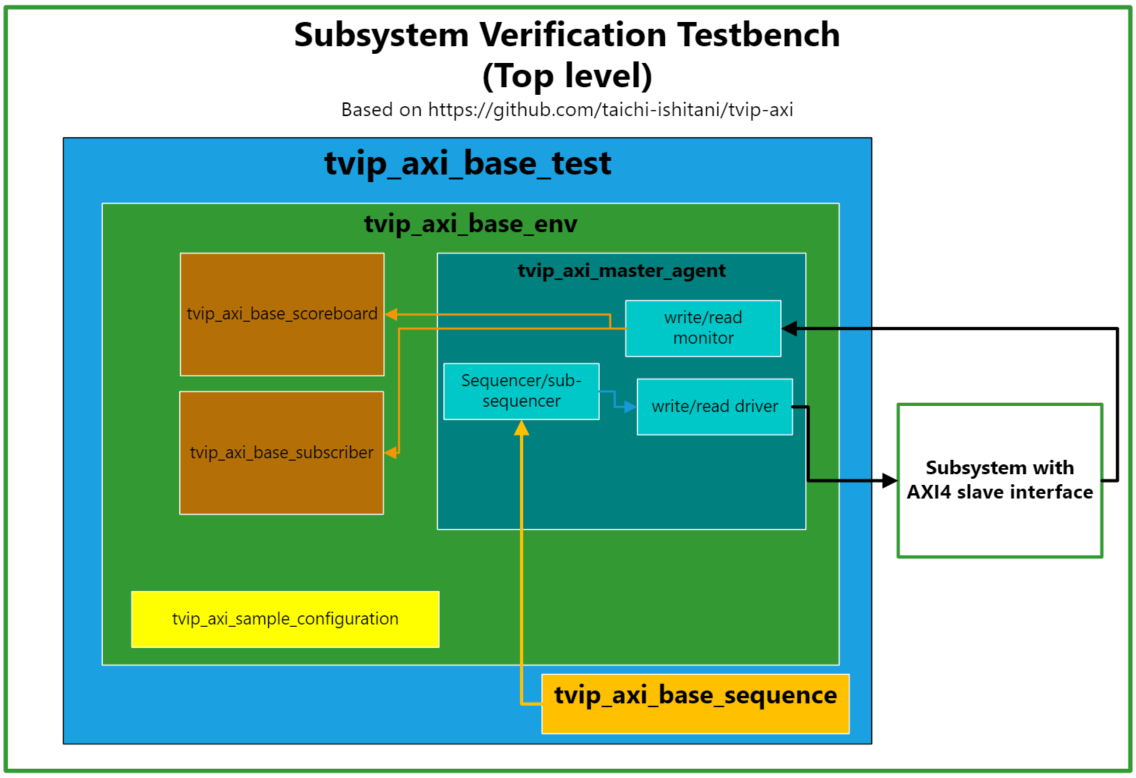


Figure 7.2. *Subsystem testbench block diagram [27]*

The memory provides the ability to read the data afterward with read transactions to provide bidirectional data flow. Both agents include a monitor which is responsible for capturing the transaction from the interface.

The transactions from both agents are passed to a scoreboard which is responsible for correctness checking. For a write transaction, the transaction is first captured when leaving the master agent and then captured again arriving to the slave agent. The scoreboard receives both transaction items and compares the payloads. For read transactions, the functionality is a bit more complicated since the read transaction from the master agent doesn't contain any data. Instead, the read transaction is used to indicate the slave agent to send data to the master. To verify the read transaction's correctness, a memory structure was created in the scoreboard. When a write transaction is received by the scoreboard, it saves the payload to the memory structure. Afterward, when a read transaction is captured with the read data, the payload is compared against the memory structure to verify that the same data written before is now read correctly.

C2C only required one master agent and one slave agent. Interconnect environment required one agent per AXI port, which led to several master and slave agents. Because UVM scales very well, adding multiple agents was a quite simple process. However using multiple agents at the same time made the situation more difficult, when multiple agents were sending transactions at the same time and in addition containing different sized ports made comparing one transaction to another impossible. The issue was solved by adding more memory structures to the scoreboard. The data of the transactions was saved to the memory structures thus negating the need to compare transactions to each other, at the end of the simulation the memory structures were compared to verify that all the data matches.

The ethernet subsystem testbench is based on the same UVM environment but the structure is different. The subsystem communicates with the SoC using AXI interfaces, both a master and a slave. On the other side ethernet subsystem has a reduced gigabit media-independent interface (RGMII) interface. RGMII interface is looped back by connecting RGMII tx to RGMII rx. With the loopback, it is possible to verify both transmit and receive functionality just from the AXI interfaces. In the testbench environment data is driven through the AXI master interface and after loopback, the data returns through the AXI slave interface. Data integrity is verified by comparing transmitted data against received.

7.1.3 Processing subsystems

Ballast includes two subsystems whose main purpose is processing data. These subsystems are AI and DSP. The main part of AI is the NVDLA core from Nvidia. It is verified on IP level already which means only top level integration verification is performed. Verification for DPS was performed on the subsystem level and integration verification on top level. DSP is based on TTA core which has been verified before this project which reduces the amount of verification work.

AI subsystem top level verification utilizes Sysctrl and C2C. In addition to the basic integration tests mentioned in general top level verification, AI functionality is verified on top level with a few different use cases. In these use cases Sysctrl is used to initialize the SoC by enabling AI, ICN, and C2C. After enabling clocks and resets, Sysctrl configures AI to perform actions needed for the use case. NVDLA then reads the input data from a memory behind the C2C interface and starts processing it. After the processing is finished which is indicated by a status register or a system level interrupt, the results are checked against reference data.

DSP uses a VHDL testbench for standalone verification in two different configurations. In the first configuration, a program can be loaded into the DSP. After processing, results are read and verified. The second configuration includes 2 DPSs in the same testbench. It is used to verify DSP's AXI read and write functionality. The DSP verification environment includes a test program generator which is used to verify the DSP core functionality such as different operations. On top level the test case used in the first testbench configuration is replicated. Only this time MPC is used to load a program to DSP and it also reads the results to verify the functionality. In addition, basic connectivity tests are executed on top level.

7.2 Top level

As described earlier, by this point all the subsystems should be verified and fully functional, thus top level verification focuses on integration verification by testing the connections and control between subsystems. The verification process is similar to processor subsystems' standalone verification. Top level testbench has the functionality to use each processor subsystem separately or all together. At least one processor subsystem needs to be active during the simulation. That is because non-processor subsystems rely on initialization to be done by one of the processor subsystems. In addition, Sysctrl is always active when the SoC is powered on and reset is lifted and that is needed because it contains the registers to control clocks and resets for all other subsystems. Control from Sysctrl can be done either by software running on Sysctrl or configuring the registers through JTAG. This approach is used for integration tests but also for use case tests. Use cases especially try to use Sysctrl booting as it is the main way to boot the SoC.

Top level verification also covers the TLP subsystem. The subsystem was designed and developed specifically for Ballast. Thus all verification is also completed within the project. TLP is only verified on top level because of the relatively simple functionality that gets covered by running integration test cases on top level. This includes testing all the registers, C2C configuration, and top level interrupt routing. In addition, a formal sanity test case is to check registers and routing.

7.3 FPGA implementation

FPGA prototyping was implemented in a few phases. The reason for this was to be able to cover as much as possible in the time available. Having the whole SoC design on the FPGA at once wasn't possible due to the time-consuming nature of the work. The design wasn't designed to be FPGA friendly, which led to having to re-design some parts for those to work on the FPGA itself. Prototyping was started by having standalone subsystems on the FPGA. The first ones were Sysctrl and MPC. In later phases multiple subsystems were implemented on the FPGA at the same time, making cross-communication possible.

In total three different FPGA boards were used for the prototyping. The first one was PYNQ-Z1 the smallest of the three. The work was started on it due to the availability. The limitation of the PYNQ board was that it had area limitations and a limited amount of IO, thus it could only fit one subsystem at once. Later the project acquired two other boards, ZCU104 and VCU118 from Xilinx. ZCU104 makes it possible to fit larger standalone designs and VCU118 potentially fits the entire system.

8. BALLAST VERIFICATION RESULTS

This chapter presents the results of the Ballast verification process.

8.1 Coverage results

Coverage results collected during the project are test plan coverage and code coverage summaries. The test plan is coverage collected from verification plans of each subsystem and top level, presenting how many total test cases were planned and how many are passing, failing, or not implemented. Code coverage is automatically collected by simulation tools and can be extracted into HTML report format. Snippets from those HTML reports containing coverage summaries are presented in upcoming sections.

As explained in the verification strategy section, only code coverage is collected because functional coverage groups were not used during the verification process. Thus the summaries only include code coverage. Coverage summaries presented below have the following code coverage types enabled:

- Statements: statements covered
- Branches: branch options covered. For example if-else structure
- FEC: focused expression coverage
- Toggles: signal transition coverage from 0->1 and 1->0
- FSM: state machine coverage. Includes states executed and state transitions

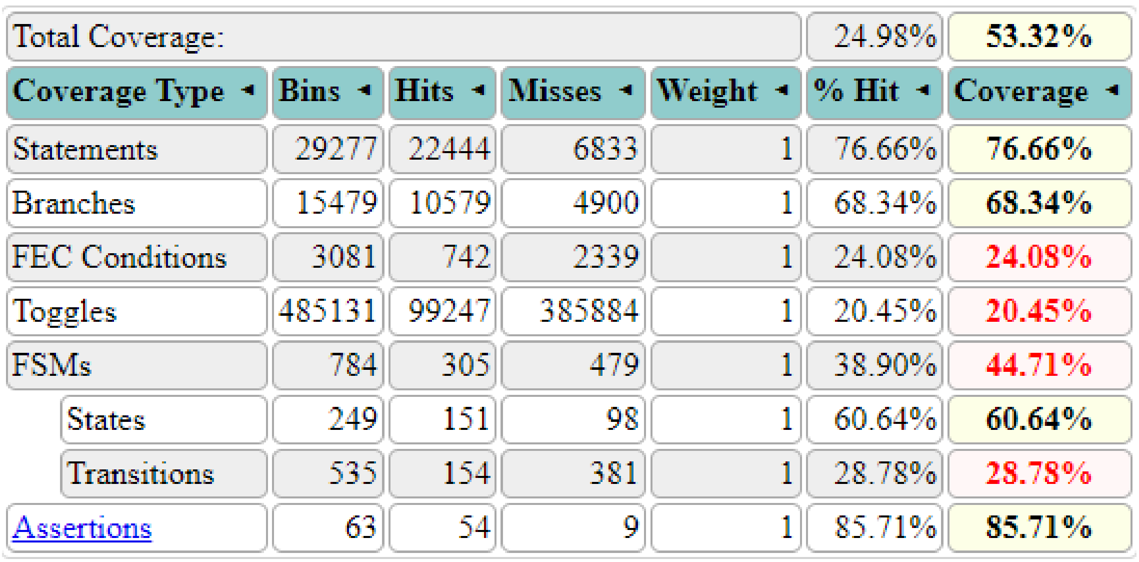
In a typical industrial project, the code coverage can be used as a direct measure of the current verification status. It is common to aim for 100% code coverage, especially on IP level. Designs are very large and at the start, the coverage data includes areas that might not be interesting from the coverage perspective. Those can be for example blocks that cannot be exercised during simulation. Coverage exclusion is used to get rid of such areas to make coverage statistics more accurate. As mentioned in the strategy section, coverage was not used as a measurement of the verification process in this project. Thus coverage exclusion was not done either which makes the coverage scores less accurate.

8.1.1 Subsystem code coverage

Coverage results for each subsystem are extracted from the simulation tools. Coverage report summaries are presented in figures 8.1, 8.2, 8.3, 8.5, 8.6, and 8.4. A coverage summary includes different coverage types as rows. The bins column indicates the number of targets for each coverage type. Hits and misses columns indicate how many of those targets were hit and missed. The weight column decides how much each coverage type affects the total coverage. Finally, the hit and coverage columns show the percentages of how large a portion of the coverage targets were hit.

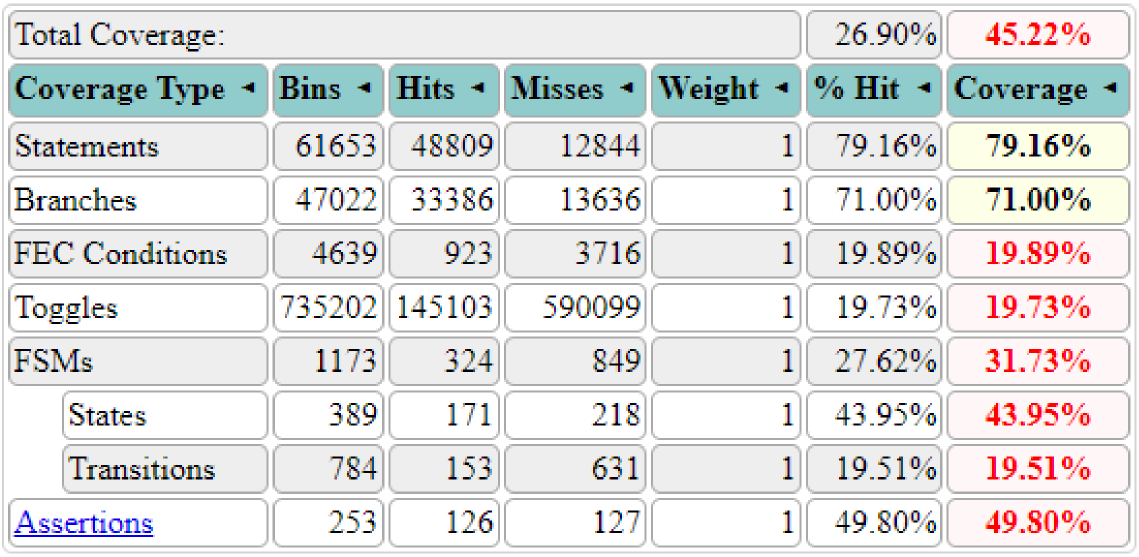
Coverage reports for different subsystems are not directly comparable. The reason is that the results are unique for each subsystem. However, one indication of the design size can be seen in the number of bins in the bins column of the coverage summaries. Bigger design results in more bins. Bins directly affect the coverage percentage.

As an example, we can try to compare Ethernet and C2C coverage-wise. The total coverage percentage for Ethernet is 79,66% and for C2C it is 42.79%. Both subsystems were thoroughly tested. The main reason for ethernet having a much higher coverage score is that the design is relatively simple and small. As mentioned above, the size can be seen in the number of bins. The main reasons why the coverage percentage is relatively low or high compared to other subsystems are lack of coverage exclusion, size of the design, and thoroughness of the verification for a specific subsystem. All these factors affect the final percentage.



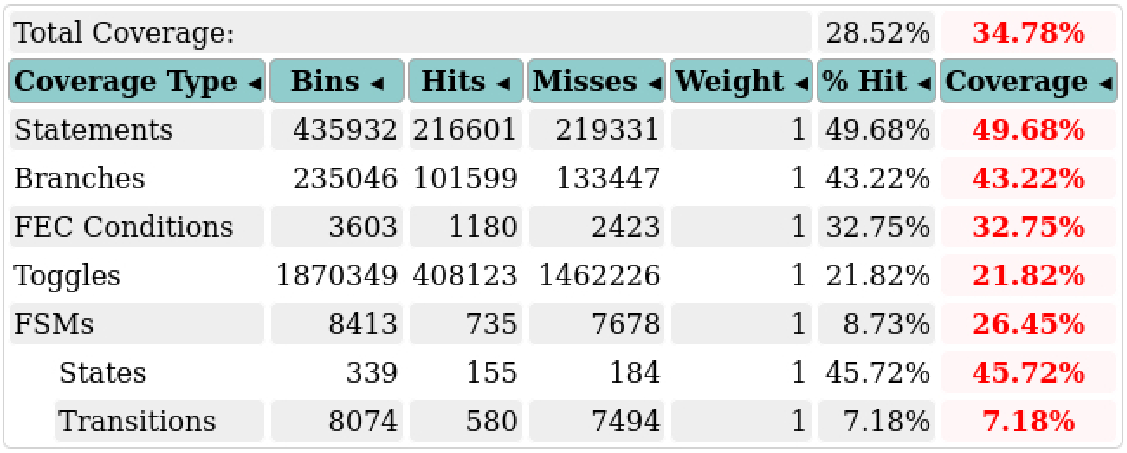
Total Coverage:					24.98%	53.32%
Coverage Type ◂	Bins ◂	Hits ◂	Misses ◂	Weight ◂	% Hit ◂	Coverage ◂
Statements	29277	22444	6833	1	76.66%	76.66%
Branches	15479	10579	4900	1	68.34%	68.34%
FEC Conditions	3081	742	2339	1	24.08%	24.08%
Toggles	485131	99247	385884	1	20.45%	20.45%
FSMs	784	305	479	1	38.90%	44.71%
States	249	151	98	1	60.64%	60.64%
Transitions	535	154	381	1	28.78%	28.78%
Assertions	63	54	9	1	85.71%	85.71%

Figure 8.1. *Sysctrl coverage report summary*



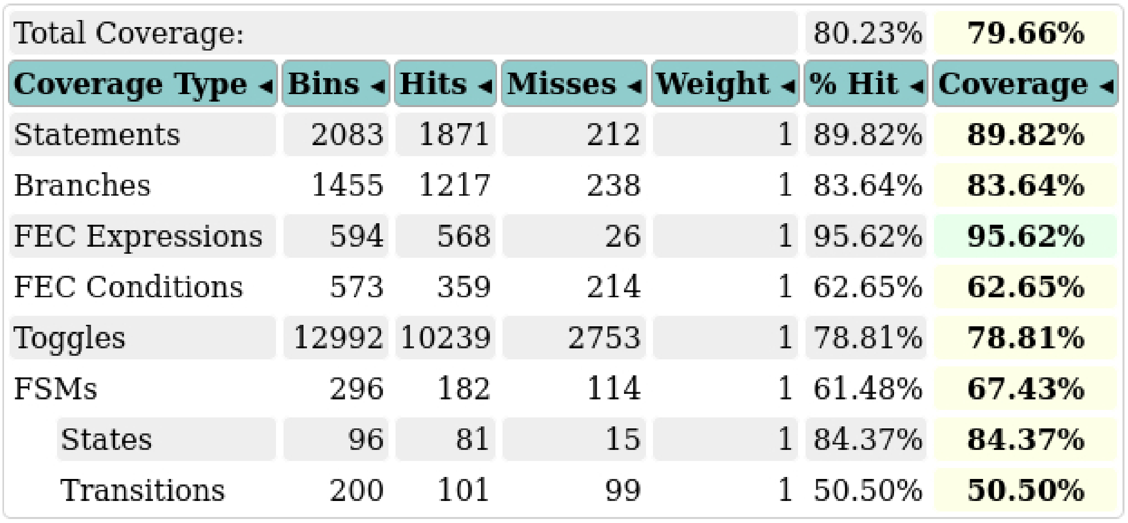
Total Coverage:					26.90%	45.22%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	61653	48809	12844	1	79.16%	**79.16%**
Branches	47022	33386	13636	1	71.00%	**71.00%**
FEC Conditions	4639	923	3716	1	19.89%	**19.89%**
Toggles	735202	145103	590099	1	19.73%	**19.73%**
FSMs	1173	324	849	1	27.62%	**31.73%**
States	389	171	218	1	43.95%	**43.95%**
Transitions	784	153	631	1	19.51%	**19.51%**
Assertions	253	126	127	1	49.80%	**49.80%**

Figure 8.2. *MPC coverage report summary*



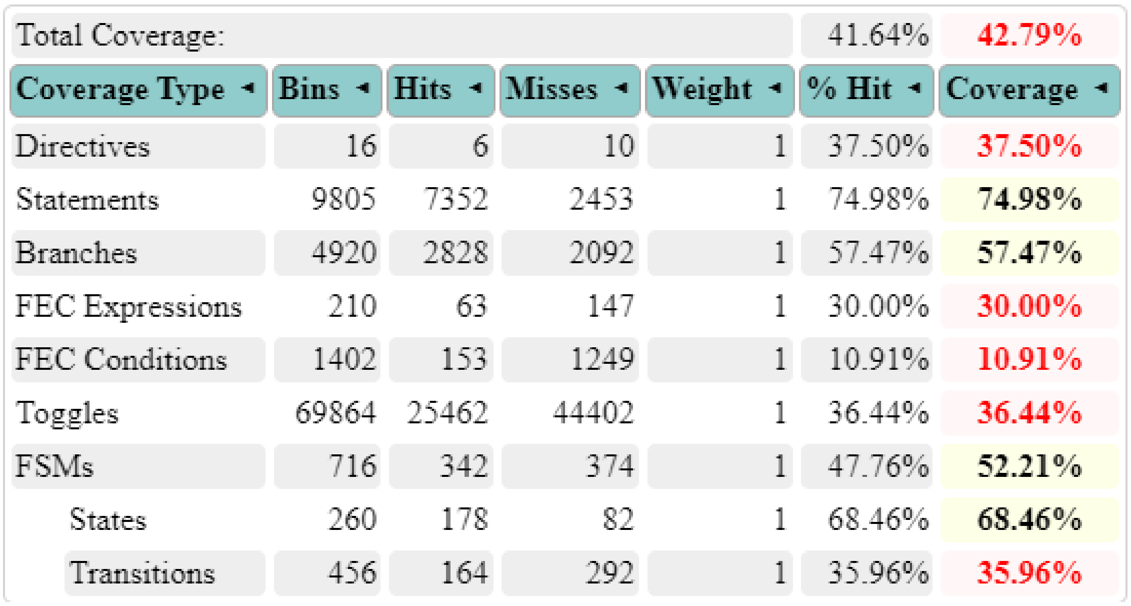
Total Coverage:					28.52%	**34.78%**
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	435932	216601	219331	1	49.68%	**49.68%**
Branches	235046	101599	133447	1	43.22%	**43.22%**
FEC Conditions	3603	1180	2423	1	32.75%	**32.75%**
Toggles	1870349	408123	1462226	1	21.82%	**21.82%**
FSMs	8413	735	7678	1	8.73%	**26.45%**
States	339	155	184	1	45.72%	**45.72%**
Transitions	8074	580	7494	1	7.18%	**7.18%**

Figure 8.3. *HPC coverage report summary*



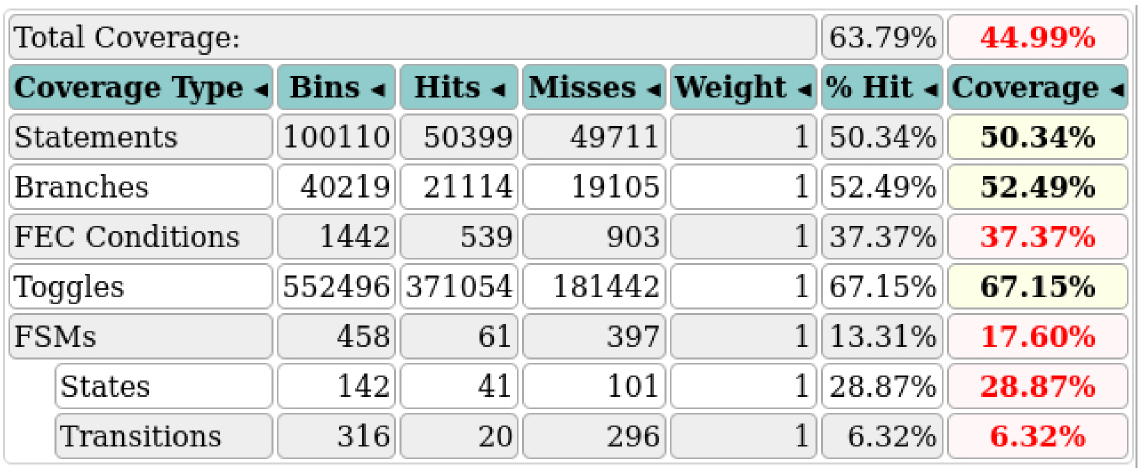
Total Coverage:					80.23%	**79.66%**
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	2083	1871	212	1	89.82%	**89.82%**
Branches	1455	1217	238	1	83.64%	**83.64%**
FEC Expressions	594	568	26	1	95.62%	**95.62%**
FEC Conditions	573	359	214	1	62.65%	**62.65%**
Toggles	12992	10239	2753	1	78.81%	**78.81%**
FSMs	296	182	114	1	61.48%	**67.43%**
States	96	81	15	1	84.37%	**84.37%**
Transitions	200	101	99	1	50.50%	**50.50%**

Figure 8.4. *Ethernet coverage report summary*



Total Coverage:					41.64%	42.79%
Coverage Type ◂	Bins ◂	Hits ◂	Misses ◂	Weight ◂	% Hit ◂	Coverage ◂
Directives	16	6	10	1	37.50%	37.50%
Statements	9805	7352	2453	1	74.98%	74.98%
Branches	4920	2828	2092	1	57.47%	57.47%
FEC Expressions	210	63	147	1	30.00%	30.00%
FEC Conditions	1402	153	1249	1	10.91%	10.91%
Toggles	69864	25462	44402	1	36.44%	36.44%
FSMs	716	342	374	1	47.76%	52.21%
States	260	178	82	1	68.46%	68.46%
Transitions	456	164	292	1	35.96%	35.96%

Figure 8.5. *C2C coverage report summary*



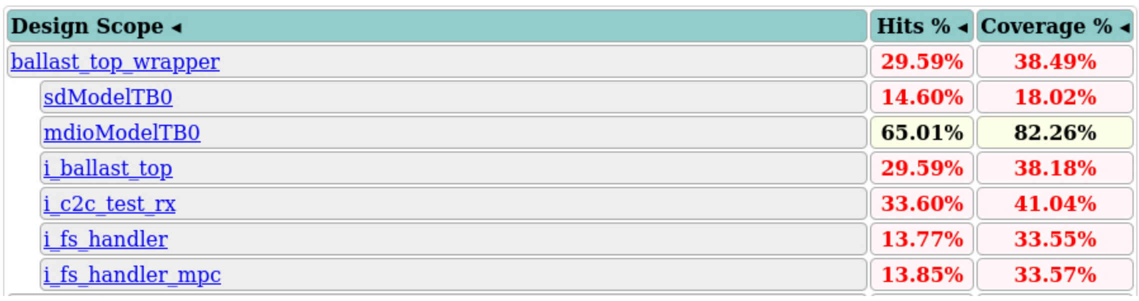
Total Coverage:					63.79%	44.99%
Coverage Type ◂	Bins ◂	Hits ◂	Misses ◂	Weight ◂	% Hit ◂	Coverage ◂
Statements	100110	50399	49711	1	50.34%	50.34%
Branches	40219	21114	19105	1	52.49%	52.49%
FEC Conditions	1442	539	903	1	37.37%	37.37%
Toggles	552496	371054	181442	1	67.15%	67.15%
FSMs	458	61	397	1	13.31%	17.60%
States	142	41	101	1	28.87%	28.87%
Transitions	316	20	296	1	6.32%	6.32%

Figure 8.6. *ICN coverage report summary*

8.1.2 Top level code coverage

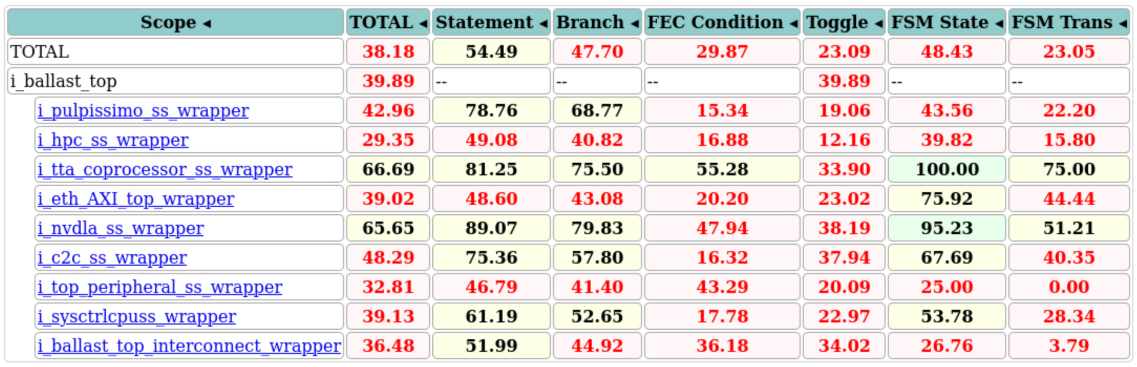
Coverage results of the top level include all Ballast top level test cases. Figure 8.7 has hierarchical coverage results for the Ballast top wrapper. Figure 8.7 has hierarchical coverage results and figure 8.9 coverage summary for Ballast top level.

Hierarchical coverage shows the coverage score for each block under the current level. It can be useful to easily spot areas that are lacking compared to others. If a low score area is seen, it should be investigated if there is a gap in verification or if that area should be excluded from the results using the coverage exclusion mentioned earlier.



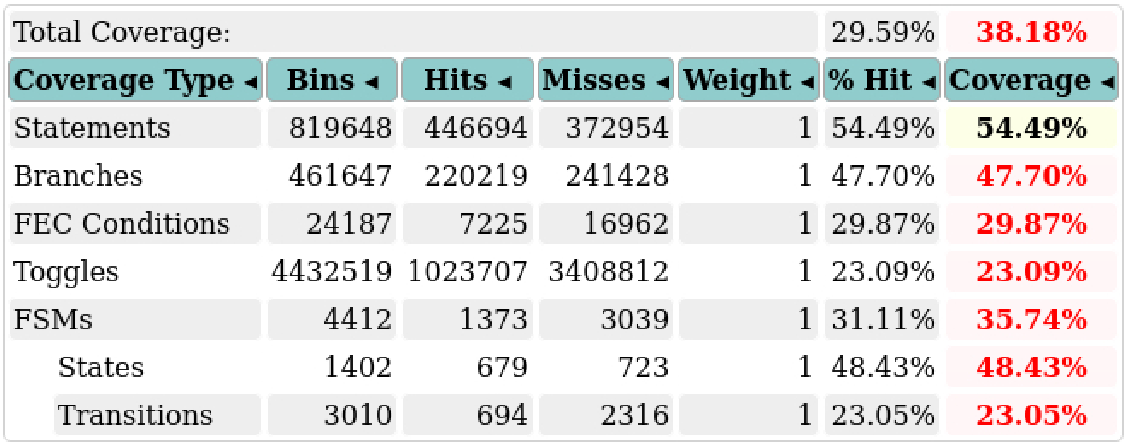
Design Scope ◂	Hits % ◂	Coverage % ◂
ballast_top_wrapper	29.59%	38.49%
sdModelTB0	14.60%	18.02%
mdioModelTB0	65.01%	82.26%
i_ballast_top	29.59%	38.18%
i_c2c_test_rx	33.60%	41.04%
i_fs_handler	13.77%	33.55%
i_fs_handler_mpc	13.85%	33.57%

Figure 8.7. *Ballast top wrapper hierarchical coverage*



Scope ◂	TOTAL ◂	Statement ◂	Branch ◂	FEC Condition ◂	Toggle ◂	FSM State ◂	FSM Trans ◂
TOTAL	38.18	54.49	47.70	29.87	23.09	48.43	23.05
i_ballast_top	39.89	--	--	--	39.89	--	--
i_pulpissimo_ss_wrapper	42.96	78.76	68.77	15.34	19.06	43.56	22.20
i_hpc_ss_wrapper	29.35	49.08	40.82	16.88	12.16	39.82	15.80
i_tta_coprocessor_ss_wrapper	66.69	81.25	75.50	55.28	33.90	100.00	75.00
i_eth_AXI_top_wrapper	39.02	48.60	43.08	20.20	23.02	75.92	44.44
i_nvdla_ss_wrapper	65.65	89.07	79.83	47.94	38.19	95.23	51.21
i_c2c_ss_wrapper	48.29	75.36	57.80	16.32	37.94	67.69	40.35
i_top_peripheral_ss_wrapper	32.81	46.79	41.40	43.29	20.09	25.00	0.00
i_sysctrlcpuss_wrapper	39.13	61.19	52.65	17.78	22.97	53.78	28.34
i_ballast_top_interconnect_wrapper	36.48	51.99	44.92	36.18	34.02	26.76	3.79

Figure 8.8. *Ballast top level hierarchical coverage*



Total Coverage:					29.59%	38.18%
Coverage Type ◂	**Bins ◂**	**Hits ◂**	**Misses ◂**	**Weight ◂**	**% Hit ◂**	**Coverage ◂**
Statements	819648	446694	372954	1	54.49%	**54.49%**
Branches	461647	220219	241428	1	47.70%	**47.70%**
FEC Conditions	24187	7225	16962	1	29.87%	**29.87%**
Toggles	4432519	1023707	3408812	1	23.09%	**23.09%**
FSMs	4412	1373	3039	1	31.11%	**35.74%**
States	1402	679	723	1	48.43%	**48.43%**
Transitions	3010	694	2316	1	23.05%	**23.05%**

Figure 8.9. *Ballast top level coverage summary*

8.1.3 Test plan coverage

Verification plan coverage is collected from subsystem verification plans. Data contains the number of total test cases and how many of those are passing. From these values, the pass rate can be calculated. The total number of tests minus passing test cases leaves us with the number of test cases that are not passing. These tests are either failing or planned but not implemented due to various reasons. Subsystem standalone test statistics can be seen in table 8.1. Top level statistics are shown in table 8.2.

Subsystem	*Total planned*	*Passing*	*Pass rate*
Sysctrl	31	30	96,8 %
MPC	38	36	94,7 %
HPC	98	81	82,7 %
C2C	7	7	100,0 %
ETH	14	12	85,7 %
DSP	12	12	100,0 %
TLP	7	7	100,0 %
ICN	3	3	100,0 %
AI	N/A		
Total:	210	188	89,5 %

Table 8.1. *Subsystem verification plan coverage*

	Total planned	*Passing*	*Pass rate*
Top level	47	31	66,6 %

Table 8.2. *Top level verification plan coverage*

8.2 Recap of different methods

A short recap of what was gained from different verification methods.

8.2.1 RTL simulation

RTL verification was the main tool for verification and debugging. Most of the verification work was done in RTL simulation. That is because verification with RTL simulation was possible to start right at the beginning of the design process, enabling us to catch bugs from the beginning. RTL simulation remained invaluable all the way to the tapeout.

8.2.2 FPGA

FPGA prototyping was invaluable for boot verification. It enabled verifying features that were not possible within the used simulation environment. One of the most notable ones is the SPI boot that was lacking a simulation model. In addition to that FPGA prototyping also revealed faults in the RTL interfacing an SD-card in SDIO boot mode. Without noticing this, the SDIO boot might not be functional. This issue was not found in the simulation because the used simulation model was faulty as well.

8.2.3 GLS

As planned, GLS was used as the last verification step before tapeout. It provided invaluable information about the timing and with GLS we are able to simulate the design using the actual synthesized blocks. No major issues were found in the areas that were verified with GLS but still increased confidence in the design.

8.3 Bugs

As expected a lot of bugs were found during Ballast verification. That is because the verification was already ongoing in parallel with design and the lack of maturity in the design caused more bugs to be found.

Some bugs were tracked and some were fixed instantly. To get a clear picture of the found bugs, tracking is very important. At the beginning of the project, an open-source project management system Redmine[25] was used for bug tracking. It was not seen as valuable and the use of it was later dropped. One reason is that detailed bug tracking is not as important in an academic project as it is for an industrial project. As the organization is much larger scale, bug tracking is more important at a higher level. The reason is that bug tracking can be used as an important measurement at the higher level of the organization structure.

Bugs fixed close to tapeout can be observed from Git commit history, for example analyzing the final weeks before tapeout. Some examples of the findings are listed in the table 8.3. The commit message column has the actual commit message used to commit the change to git. The bug column gives a short description of the bug. The method column shows what method what used to discover the bug.

Table 8.3. *Bugs found during verification*

Commit message	**Bug**	**Method**
Fixed for higher number of pads	Missing pad configuration for added pads	RTL simulation
Fixed x assign reachable issue in dbg lint module	Value X is assigned in Sysctrl RTL	Backend tools
Fixed reg-IF and SD card model to use block addressing like a real SD card	SD-card was accessed using byte addresses instead of block addresses	FPGA

8.4 Sample testing

When the physical chips arrive from fabrication, the sample testing is started. Before that, a wake-up plan is created to make the chip bring-up as smooth as possible. The basic idea is to replicate the behavior observed during verification. Re-use is important here as most of the verification test cases can be directly re-used in sample testing. Before the actual wake-up the SoC and board it is mounted to are tested for electrical issues. The first steps of Ballast wake-up are:

1. Testing for electrical faults
2. Connecting an external debugger to sanity check Sysctrl
3. Running integration tests such as register and memory access on Sysctrl to check accessibility to the rest of the chip
4. Checking the rest of the subsystems that can be communicated with by the external debugger and running more integration tests
5. Running use cases

During sample testing, possible bugs are documented for further analysis. Sometimes a documented bug can be later declared as for example electrical fault, fabrication fault, or a bug missed by verification. Of course, in these cases, it is important to find the root cause as soon as possible. During Ballast wake-up, it is important because many were to be re-used in the second SoC-Hub chip, Tackle. Notable issues found during sample testing, their cause, and the solution are listed in the following table 8.4.

Table 8.4. *Bugs found in sample testing*

Bug	**Cause**	**Impact**	**Reason**
DSP memory integration	Wrong file in GDS-II tapeout	High. Subsystem not usable	GLS was not run
Direct SDIO boot	Unknown	Low. Boot mode executing directly from SD-card not functional	Unknown

9. CONCLUSION

This thesis focused on documenting the entire verification process of the Ballast SoC. The goal was to introduce the relevant background and present all steps of the verification process that were completed during Ballast SoC verification. The thesis introduces background information about SoCs in general and relevant topics about pre-silicon verification, including the verification flow and multiple methods that can be used in verification, and what kind of results can be extracted from the verification process. The next chapter gives insight into related work and how verification is completed in similar projects. Ballast architecture is presented in the next chapter including information about all subsystems. Finally, the last chapters presented Ballast verification strategy, implementation, and results.

Ballast verification activities were ongoing for the entire design cycle. The chip itself was considered huge relative to other previous academic projects which naturally results in a lot of verification work. Overall the schedule to complete the verification was tight and required careful planning. Many of the used components were re-used from open-source projects and were considered to be verified to some degree without having a full guarantee of the functionality. Careful planning was done to identify critical areas and those became points of focus for the verification.

Samples arrived for sample testing during the writing of this thesis. Sample testing was conducted and the results were good. Only one critical bug was found and it affected only one of the 9 subsystems. After sample testing, the work kept going in terms of software development and further testing of the chip.

REFERENCES

[1] Wayne Wolf Ahmed Jerraya. *Multiprocessor Systems-on-Chips*. The Morgan Kaufmann series in systems on silicon. Saint Louis: Elsevier Science, 2004. ISBN: 978-0-08-051227-3.

[2] Janick Bergeron. *Writing Testbenches using System Verilog*. Boston, MA: Springer US, 2006. ISBN: 978-0-387-29221-2 978-0-387-31275-0. DOI: 10.1007/0-387-31275-7. URL: http://link.springer.com/10.1007/0-387-31275-7 (visited on 11/30/2022).

[3] Wen Chen, Sandip Ray, Jayanta Bhadra, Magdy Abadir, and Li-C Wang. “Challenges and Trends in Modern SoC Design Verification”. In: *IEEE Design & Test* 34.5 (Oct. 2017). Conference Name: IEEE Design & Test, pp. 7–22. ISSN: 2168-2364. DOI: 10.1109/MDAT.2017.2735383.

[4] NVDLA doc. *NVDLA*. URL: http://nvdla.org/primer.html (visited on 10/11/2022).

[5] Aleksei Gimbitskii. “Interconnect design for the edge computing system-on-chip”. MA thesis. Tampere university, 2022. URL: https://urn.fi/URN:NBN:fi:tuni-202206035477.

[6] Git. *What is CI/CD?* 2023. URL: https://about.gitlab.com/topics/ci-cd/ (visited on 01/21/2023).

[7] OpenHW Group. *CORE-V cores*. 2023. URL: https://github.com/openhwgroup/core-v-cores (visited on 02/24/2023).

[8] OpenHW Group. *CORE-V Verification strategy*. 2023. URL: https://docs.openhwgroup.org/projects/core-v-verif/en/latest/ (visited on 02/10/2023).

[9] OpenHW Group. *OpenHW verification planning*. 2023. URL: https://github.com/openhwgroup/core-v-verif/blob/master/docs/VerifPlans/VerificationPlanning101.md (visited on 02/24/2023).

[10] Taichi Ishitani. *TVIP-AXI*. 2022. URL: https://github.com/taichi-ishitani/tvip-axi (visited on 12/14/2022).

[11] Multanen J. Jääskeläinen P Hepola K. *AamuDSP Ballast TTA*. 2021. URL: https://gitlab.tuni.fi/soc-hub/ballast/ballast_tta/-/blob/master/doc/manual/manual.pdf (visited on 10/11/2022).

[12] A. Jerraya, H. Tenhunen, and W. Wolf. “Guest Editors’ Introduction: Multiprocessor Systems-on-Chips”. In: *Computer* 38.7 (July 2005). Conference Name: Computer, pp. 36–40. ISSN: 1558-0814. DOI: 10.1109/MC.2005.231.

[13] Youn-Long Steve Lin, ed. *Essential Issues in SOC Design: Designing Complex Systems-on-Chip*. Dordrecht: Springer Netherlands, 2006. ISBN: 978-1-4020-5351-

1 978-1-4020-5352-8. DOI: 10.1007/1-4020-5352-5. URL: http://link.springer.com/10.1007/1-4020-5352-5 (visited on 11/30/2022).

[14] lowRISC. *Ariane RISC-V CPU*. 2019. URL: https://github.com/lowRISC/ariane (visited on 08/08/2023).

[15] lowRISC. *Ibex: An embedded 32 bit RISC-V CPU core*. 2020. URL: https://ibex-core.readthedocs.io/en/latest/ (visited on 08/08/2023).

[16] Ashok B. Mehta. *ASIC/SoC Functional Design Verification A Comprehensive Guide to Technologies and Methodologies*. 1st ed. 2018. Cham: Springer International Publishing, 2018. xxxi+328. ISBN: 978-3-319-59418-7. DOI: 10.1007/978-3-319-59418-7.

[17] *Metric Driven Design Verification*. URL: https://link-springer-com.libproxy.tuni.fi/book/10.1007/978-0-387-38152-7 (visited on 11/30/2022).

[18] Antti Nurmi, Antti Rautakoura, Henri Lunnikivi, and Timo Hämäläinen. "A Resilient System Design to Boot a RISC-V MPSoC". In: *25th Euromicro conference on Digital System Design*. In press. 2022.

[19] Arto Oinonen. *COMP.CE.420 System-on-Chip Verification*. Course material. June 2023.

[20] OpenTitan. *OpenTitan introduction*. 2023. URL: https://opentitan.org/documentation/index.html (visited on 06/09/2023).

[21] OpenTitan. *OpenTitan verification methodology*. 2023. URL: https://opentitan.org/book/doc/contributing/dv/methodology/index.html#documentation (visited on 06/09/2023).

[22] PULP platform. *Pulpissimo GitHub*. 2022. URL: https://github.com/pulp-platform/pulpissimo (visited on 11/18/2022).

[23] Antti Rautakoura. *Lecture slides from System Design COMP.CE.400*. Feb. 2022.

[24] Antti Rautakoura, Timo Hämäläinen, Ari Kulmala, Mehdi Duman, and Mohamed Ibrahim. "Ballast: Implementation of a Large MP-SoC on 22nm ASIC Technology". In: *25th Euromicro conference on Digital System Design*. In press. 2022.

[25] Redmine. *Redmine wiki*. 2023. URL: https://www.redmine.org/projects/redmine/wiki (visited on 08/11/2023).

[26] Andrea Guerrieri René Beuchat Florian Depraz. *Fundamentals of System-on-Chip Design on Arm Cortex-M Microcontrollers*. Arm Education Media, 2021. ISBN: 978-1-911531-35-7.

[27] SoC-Hub. *SoC-Hub internal documents*.

[28] *System-on-a-Chip Verification*. Boston: Kluwer Academic Publishers, 2002. ISBN: 978-0-7923-7279-0. DOI: 10.1007/b116428. URL: http://link.springer.com/10.1007/b116428 (visited on 11/30/2022).

[29] Krishnan K Yadu and Ramesh Bhakthavatchalu. "Block Level SoC Verification Using Systemverilog". In: *2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA)*. 2019 3rd International conference

on Electronics, Communication and Aerospace Technology (ICECA). June 2019, pp. 878–887. DOI: 10.1109/ICECA.2019.8821909.