NTNU
Norwegian University of
Science and Technology

Verification of a Large Heterogeneous Many-core Computer

Eivind Gamst
Edward Mitacc

Embedded Computing Systems
Submission date: June 2016
Supervisor: Donn Morrison, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Summary

The historical trend of steady increase in processor performance with each technology generation has slowed down during the last years due to power limitations. As transistor sizes reduce, the power density on a chip does not remain constant anymore, which is known as the end of Dennard scaling. This increase of power demand limits the number of transistors that can be used simultaneously without exceeding the power budget. This phenomenon, known as the Dark Silicon effect, can be mitigated by building heterogeneous systems containing processing elements of different performance and power characteristics. The SHMAC project at the NTNU aims to provide a platform for investigating heterogeneous systems at all abstraction levels.

Current verification strategies for verifying the hardware parts of the SHMAC platform include block-level and top-level testbenches, and bare-metal testing on FPGA. Both of them run directed tests that exercise specific features of the design, which are manually handcrafted by each SHMAC developer. This approach not only represents a tedious task for the designer, but also does not ensure to reach all corner cases within the design. In addition, these verification strategies lack of coverage metrics that measure verification progress and quality, and do not provide mechanisms to effectively identify and track bugs in the design.

This project proposes a new verification framework and methodology for the SHMAC platform using the Universal Verification Methodology (UVM). This new methodology is aimed to overcome the limitations of the existing verification strategies previously presented, and also is intended to provide highly reusable verification environments. The latter plays an important role in reducing the effort and time spent on creating new tests as the design complexity of the SHMAC platform increases and new extensions are implemented.

Preface

This thesis is submitted to the Norwegian University of Science and Technology as a partial fulfillment of the requirements for a masters degree. The thesis is written as a part of the SHMAC research project by EECS, NTNU. The project researches challenges in heterogeneous computing systems, for improved energy efficiency.

The work was performed at the Department of Electronics and Telecommunications, Faculty of Information Technology, Mathematics and Electrical Engineering, NTNU, Trondheim, with Donn Morrison as supervisor.

We would like to thank Donn for useful feedback and a positive attitude. Our weekly meeting with Donn has helped as a motivation to keep up a consistent amount of work. It's much thanks to these meetings that this thesis is at the level it is now.

Eivind want to give a special thanks to Donn for allowing him to work remotely and have all the meetings through an online video service.

Table of Contents

List of Tables

List of Figures

Acronyms

VIP	=	Verification IP
IP	=	Intellectual Property
SHMAC	=	Single-ISA Heterogeneous Many-core Computer
FPGA	=	Field-Programmable Gate Array
UVM	=	Universal Verification Methodology
EECS	=	Energy Efficient Computing Systems
SoC	=	System on Chip
DUV	=	Device Under Verification
DUT	=	Device Under Test
HDL	=	Hardware Description Language
HVL	=	Hardware Verification Language
RTL	=	Register Transfer Level
VC	=	Verification Component
TLM	=	Transaction Level Modeling

Chapter 1

Introduction

1.1 Motivation

Computing system performance has grown at an impressive rate since the mid 1980s. Until the early 2000s, this increase in performance has followed Moore's law [35] with a growth of roughly 50% annually, as depicted in Figure 1.1. This was in part made possible because improvements in production technology adhered to the principle called Dennard scaling [15]. In a Dennard scaling process, transistor sizes are reduced while keeping the electric fields constant. This translates into a lower power consumption per transistor, which again was used to add more transistors to the same die area. The result is more transistors on a fixed chip area at a constant power consumption.

Dennard scaling keeps the electric fields constant by reducing the transistors supply voltage and threshold voltage. However, sub-threshold leakage currents increase exponentially when the threshold voltage is reduced, which has led to the end of Dennard scaling when production size decreased past the 130 nm level [30]. This means it is no longer possible to power more transistors without either increasing the power budget or leaving parts of the chip unpowered. The latter is associated with the concept of Dark Silicon Effect [24], which implies that only parts of a chip can be active simultaneously while others have to be powered off in order to stay within the power budget.

The Dark Silicon Effect is motivating the emergence of new fields of study, where different approaches for transistor utilization are explored. One such field is the area of heterogeneous processor design, where multicore processors are composed of cores that have different performance and power characteristics. The task at hand is to select the processing elements that will maximize performance for the current application under a fixed power budget (i.e. maximizing energy efficiency), while the remaining processing ele-

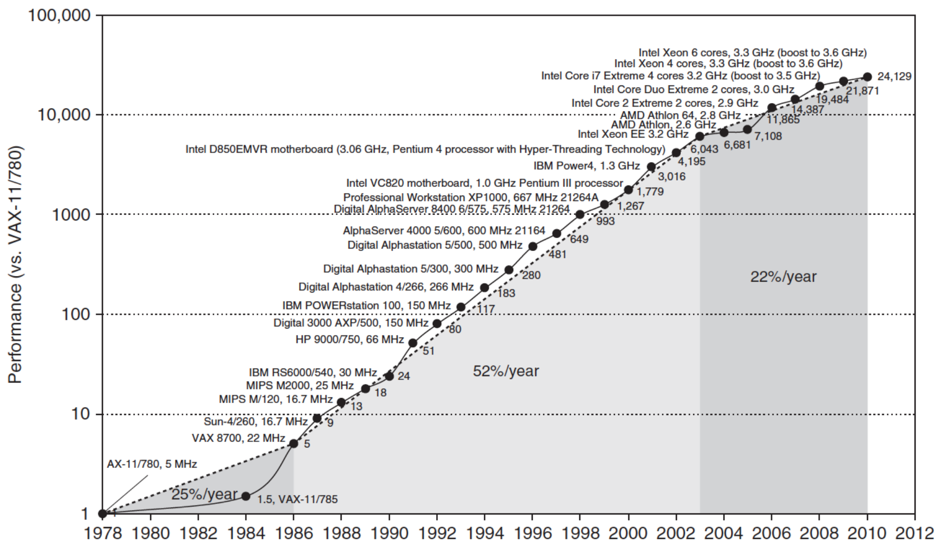

Figure 1.1: Historical growth in processor performance [27]

ments are powered off. ARM's big.LITTLE technology is an example of a heterogeneous computing architecture employed in the consumer market. It combines relatively slow, low-power processor cores (LITTLE) with relatively more powerful and power-hungry ones (big). Each processing task is dynamically allocated to a big or LITTLE core depending on the instantaneous performance requirement of that task, as shown in Figure 1.2, where A15 and A7 clusters are the big and LITTLE cores respectively.

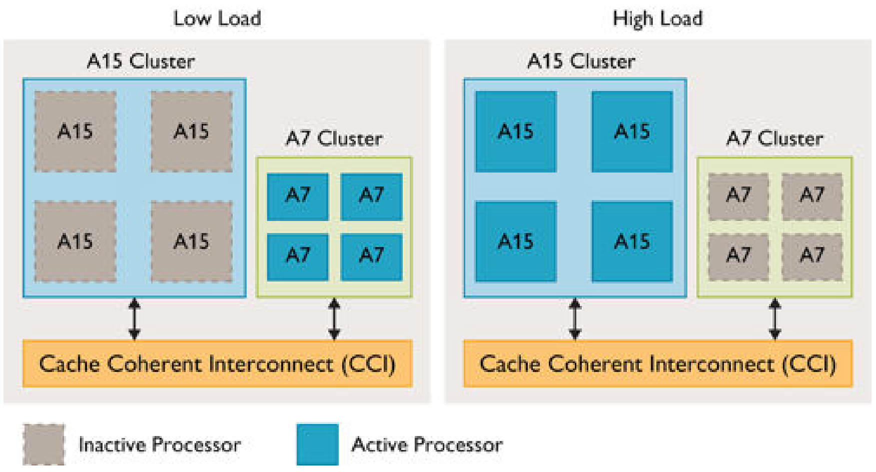

Figure 1.2: ARM's big.LITTLE technology [40]

The EECS group at NTNU contributes to heterogeneous processors research with the Single-ISA Heterogeneous MAny-core Computer (SHMAC) project. SHMAC is an infrastructure for investigating heterogeneous systems at all abstraction levels. The key idea is to create a flexible framework in which different heterogeneous processors can be created from a collection of processing elements and accelerators. The SHMAC platform has a tile-based architecture, where processing elements are laid out in a rectangular grid with connections to their nearest neighbor. To facilitate software design space exploration, the programming model is kept constant across SHMAC-instances.

Functional verification is necessary to ensure the correctness of the SHMAC architecture. Current verification strategies for validating the functionality of the hardware parts of the SHMAC platform include block-level and top-level testbenches, and bare-metal testing on FPGA. The first ones run a set of tests on the SHMAC RTL, while the latter executes tests on a FPGA platform with a SHMAC bitfile. Both of them currently employ only predefined test cases, also referred as directed tests [29], which are hand-coded by each SHMAC developer. The main drawback of this approach is that each verification scenario has to be considered independently and human interaction is required to set up the complete verification environment, generating the necessary traffic and checking results for each test case. This represents a time-consuming and tedious task for the developer, and also often leads to verification incompleteness, since it requires detailed enumeration of all possible scenarios and corner cases.

Other deficiencies of the actual verification strategy for the SHMAC platform were identified by conducting a survey among SHMAC developers (see **Appendix G** for results). Some of them agreed that the existing Verilog testbenches and the framework for running bare-metal tests are not properly documented. Also, one of the main limitations they found was the absence of debugging mechanisms to effectively identify and analyze errors in the designs. This is particularly the weakest point of bare-metal tests since it is not possible to get access to the values of the internal nodes unless additional hardware is added. It was also confirmed that the limited error diagnosis capability is one of the major obstacles software developers of the SHMAC face, since it makes it difficult to distinguish between hardware and software bugs.

The current verification strategy has several limitations as described in the previous lines, and in order to be able to bring the current platform up to leading industrial quality standards as well as enable future upgrades and modifications to the platform to be quickly integrated and verified, a new verification strategy is needed. In addition to create an optimized verification environment, it is also important to provide guidance on how the SHMAC developers should implement the best verification practices. This includes the elaboration of templates for efficient design specifications documents and verification plans, as well as guidelines for implementing good design for verification practices.

This project is aimed at providing a new verification framework for the SHMAC platform which maximizes quality and reusability while reducing the effort and time spent on verification. This will be possible by employing the Universal Verification Methodology (UVM), which is a standardized methodology for verifying complex IP and SoC in the semiconductor industry. As UVM places an emphasis on reuse of the verification infrastructure, it will make it possible to save significant time in the verification process when new extensions or modifications are implemented on the SHMAC platform. The quality of verification will be ensured by providing guidelines for good verification practices to the SHMAC developers, as explained before.

1.2 Requirements for the project

Requirements are essential for systematically evaluating the success of the verification framework to be implemented. The following table presents the main requirements that the new verification framework must satisfy.

Table 1.1: Requirements for the new verification framework of the SHMAC platform

Requirement ID	Description
REQ-Q1	The verification strategy must clearly define which metrics and level of those metrics that is needed in order to decide when verification is complete. If different environments require different metrics to be collected, that must be clearly stated. Example metrics: raw line coverage for block level testbenches must be higher than 95%. Explained (i.e. justified) line coverage for block level testbenches must be 100%.
REQ-Q2	The verification strategy must define the different types of environments that should be implemented and high level features of those environments such as stimuli generation strategy and checking strategy.
REQ-Q3	The verification strategy must enable the user to do as efficient debugging as possible.
REQ-Q4	The verification strategy must enable verification environments that are robust against future changes to the design and that can easily be reused.
REQ-Q5	It must be easy to determine if the design has a HW bug.
REQ-Q6	Assuming a certain quality level has been achieved, the verification strategy must describe how the same level can be maintained over time even if modifications to the RTL is ongoing.
REQ-Q7	The verification strategy must give guidance on how the team should prioritize in order to get from the current state to the new and improved way of doing verification.
REQ-Q8	The full system contains both hardware and software. The verification strategy should also describe how hardware and software can be verified together.

REQ-Q9	The verification strategy should also describe typical design for verification guidelines that designers should follow when writing RTL.
REQ-Q10	The verification strategy must list required documentation needed before a verification engineer can efficiently start on investigating how to e.g. verify a block. A outline of the content of the necessary documents must be specified.
REQ-Q11	The verification strategy must describe what document that must be produced when architecting a new verification environment and the content of those documents.
REQ-Q12	The verification strategy must describe how bugs will be managed and tracked.

1.3 Report structure

The organization of this report into individual chapters is briefly described for the readers convenience:

Chapter 2: Background gives an overview of the SHMAC architecture, and explains key concepts regarding functional verification, UVM and SystemC.

Chapter 3: Related work describes the existing verification strategy for the SHMAC platform, as well as similar works regarding verification of complex processors and SoCs.

Chapter 4: SHMAC verification methodology includes an analysis of the limitations of the current verification strategy for the SHMAC, and presents a new verification methodology aimed at overcoming these deficiencies.

Chapter 5: SHMAC verification framework gives a description of the new UVM-based verification framework for the SHMAC, and explains how the verification environments were implemented.

Chapter 6: Discussion/evaluation presents an analysis of the contribution of this work to the continuous development of the SHMAC.

Chapter 7: Conclusion and future work provides concluding remarks and propositions for future work on this project.

Chapter 2

Background

Parts of this chapter have been fully or partially reused from the project preceding this thesis, this is information of equal relevance to this dissertation [20].

2.1 SHMAC

The Single-ISA (Instruction Set Architecture) Heterogeneous MAny-core Computer is a research project by the Energy Efficient Computing Systems group at NTNU that tries to figure out how to make more energy efficient computers through a heterogeneous architecture [21]. The motivation for this comes as a result of the Dark Silicone effect.

The Dark silicon effect is a result of the fact that transistors are still shrinkable but the power consumption is not [24]. An increased amount of transistors per area and a fixed power per chip constraint, has resulted in that a lot of transistors has to be turned off during operation. Because of this a lot of alternative architectures has emerged. They rely on large-scale parallelism, heterogeneous cores, and accelerators to achieve performance and energy efficiency [9]. The challenge is to figure out which amount of smaller and bigger cores, with or without internal accelerators, and separate external accelerators that gives the largest performance increase on the smallest amount of power.

The architecture is a mesh of different tiles coupled together. The tiles have a common instruction set and architecture model, such that there is software portability across different SHMAC instances. There is a common router in all the SHMAC tiles that makes this possible. It is a uniform architecture, all processing tiles see the same memory map, but tile registers are per-tile, other memory locations are global. Every tile can be loaded with any type of logical unit. The tile communicates only with its neighbours, and the

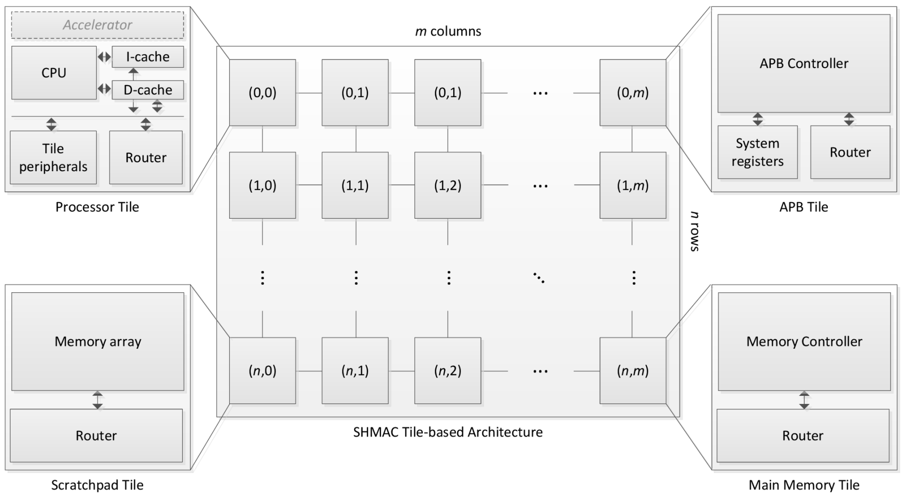

Figure 2.1: Illustration of the high level architecture of the SHMAC processor [21]

mesh must be rectangular, but not necessarily square. Every SHMAC architecture must contain one and only one APB tile, and one and only one main memory tile [22]. The APB tile takes care of all the I/O in and out of the SHMAC mesh, and the memory tile, depending on the target FPGA it is emulated on, is either ZBT-RAM or DDR-RAM. In addition to these we also have processor tiles, scratchpad tiles, accelerator tiles and dummy tiles. The processor tiles are based around a modified RISCV sodor Z-scale core, but also includes a tile register block, a tile memory and glue logic to bind it all together. Additional internal accelerators are also possible to include on the processor tile. The cores can be implemented with a number of different energy/performance characteristics, and can be optimized with vector processing, Out-Of-Order (OOO) operation, Branch prediction, and etc. Cache sizes are also variable. The scratchpad tile is a RAM tile that provides extra memory to connected tiles. Accelerator tiles are external accelerators designed for a specific purpose, an example being an SHA1 core based on the SHA1 algorithm [22]. Dummy tiles are empty tiles, that only includes the router. These can be used in cases where it is impossible/impractical to have a functional tile, and is just used to create the rectangular shape of SHMAC. SHMAC tiles are summarized in table 2.1.

2.2 Verification

Since Moore's law was introduced in 1960 the amount of transistors on a chip has increased exponentially, and with it the complexity of the chip. With the increased complexity of today's processor designs, the time spent on verification has grown from 30-40% spent of the total effort in 1996 to 50-80% spent today [13]. This is illustrated in Figure

Table 2.1: SHMAC tiles

Prefix	Tile type	Amount allowed in the grid
V	I/O	1
C	CPU	0-9
R	Scratchpad	0-8
Z	ZBT RAM	1, if synthesizing for the RealView PB11MPCore
D	DDR RAM	1, if synthesizing for the Versatile Express
S	SHA1	0-8
.	Empty	0-9

2.2. As processor designs keep getting more and more complex, verification becomes a bigger and bigger part of the total effort spent on each design, and also the biggest delay in the time to market process.

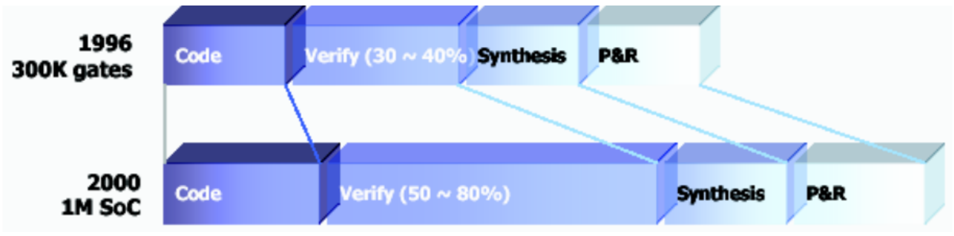

Figure 2.2: Illustration of the increasing verification gap [13]

The process of verifying a design have evolved from simple directed testing to extensive verification libraries and methodologies. Directed tests are fast and easy to write for small designs but a thorough verification requires more detailed systems. UVM which is used for verification in this thesis, is a combination of different methodologies mostly based on SystemVerilog [38] and *e* [38].

Verification is a process used to make sure that a design does what it is specified to do. A test to an extent that it gives confidence in the correctness of the design. It is an observation of design response based on input stimuli that ideally reaches all possible states in a design. Most designs today are too large and too complex that to completely verify them, reach all possible states and to try all possible combinations, would take such a large amount of time, that it is practically impossible. That is why verification is normally done through simulation of either a behavioral model or a specific state/value of the design at a given time.

2.2.1 Directed test

Directed tests are simple tests, written from scratch, that directly targets expected signal transitions and requires little overhead. It is an efficient way of writing basic tests for small designs, but becomes tedious and completely impossible to do for large designs. Tests derived from a black box test specification has been shown by experience to be insufficient to reach structural coverage targets [6]. Certain bugs in a system can be very hard to imagine for a human mind, and added to this is the lack of ability for directed tests to be reused to a larger degree. This calls for better and more efficient methods for verification.

2.2.2 Static Verification

Static verification, also known as formal verification/analysis, is a mathematical proof of a device under verification (DUV) [28]. A property to be studied is represented as a mathematical model, where a calculus is used to perform computations on the model [10].

It is normal to describe a circuit as a functional unit and compute its outputs based on its inputs and internal states, while avoiding timing constraints and the circuits electrical behavior. The calculus must prove or disprove the existence of a relation between the abstract models of the circuit specification and its implementation. Formal verification can therefore only ascertain the logic correctness of a circuit. It is independent of technological choices and cannot verify the design at a physical level. Model checking is the most common form of formal verification. It represents the DUV as a finite state machine where all sets of specified properties are mathematically proven for all input combinations, and across all execution paths.

Formal verification is heavily reliant on logic as a tool. That includes first-order predicates, higher order predicates and temporal logic. First-order predicates takes only individual constants/variables as arguments, memory elements and interconnection wires is described as time functions, gates is described as logical connectives. Higher order predicates have additional quantifiers and stronger semantics compared to the first-order predicates. Inputs and outputs are modeled as parameters, interconnection of components results from the conjunction of the predicates of the components, and wires are modeled as quantified variables in each predicate. What gives the logic a higher order is that functions and predicates can be arguments and results of other functions and predicates. Temporal logic is defined as a system which contains both linear and branching operators [7]. It is used to reason and represent propositions that are fitting in regards to time.

2.2.3 Dynamic Verification

Dynamic verification is verification done at the same time as test code is executed on the DUV. A model of the DUV has to be executed with applied input stimuli, compared to a specified behaviour, and flagged for differences [28]. Dynamic verification consists of three elements, stimulus generation, coverage measurements and analysis, and response checking.

Stimulus generation

Stimulus generation is to create and apply input patterns that fully exercise the DUV, with legal and useful stimulus.

- Legal stimuli, a particular data/temporal pattern that is not prohibited by the specification.
- Useful stimuli, a pattern that improves verification coverage, exercises corner cases and find bugs.

There are three common stimulus generation techniques:

1. Manual directed tests, a handwritten program that stimulates the DUV and checks its response.
2. Verification environments, application specific program that implements stimulus generation, response checking and coverage measurements in the same program.
3. Random test generators, tests of random functionality or behavior on the DUV to test for hidden holes in the verification coverage.

Coverage measurements and analysis

Coverage measurement is formally defined as how thoroughly a design has been exercised during verification [28]. Its primary coverage is code coverage, including integral coverage, and functional coverage, usually designed for and applied to both the specification and the implementation. The coverage measurement can be divided into two tasks:

1. Identification of DUV features
2. Coverage model that quantifies the behavioral space it has.

Response checking

Response checking is to apply stimulus to a DUV and then compare its response to its specified behavior [28].

1. Scoreboard, data structure that records expected DUV output and compares it to the observed response from the DUV after it responds to a particular stimulus.
2. Reference model, a program that reflects intended DUV behavior at a chosen abstraction level, providing portals or hooks to observe intended behavior.

3. Assertions, a statement containing a Boolean and/or temporal expression describing a liveness or safety property. A liveness property states something must eventually happen, while a safety property states that something must never happen.

2.2.4 Electronic System-Level Design

Electronic System-Level (ESL) Design is the use of a higher level of abstraction to solve the problem with the increasing design complexity in modern chip design[8]. This is to reduce development cost by creating a virtual system prototype that enables earlier development of software. The result of this is that it eases the communication between different design groups when they try to figure out the best trade off between hardware and software in terms of energy and performance.

2.2.5 Transaction-Level-Modeling

Transaction-Level-Modeling (TLM) is a technique to enable communication between blocks of different abstraction levels, [8]. It makes ESL models practical, as all important information that needs to be transferred in one turn is transferred as one single event or transaction. This opens up the possibility to refine interface blocks or communication blocks independently from the boxes they connect together. And makes for earlier development of software and an earlier and better functional verification of hardware. TLM use cases are architecture and algorithmic modeling, virtual software development platforms, and as reference models for functional verification.

Architecture Modeling

Architecture Modeling is concerned with the partitioning of hardware and software. It is to balance out which type of bus that is needed, how fast it needs to be and what kind of arbitration scheme that is sufficient, when estimating the size and cost compared to the goals of the model.

Algorithmic modeling

Algorithmic modeling is to figure out how to best implement application specific algorithms, whether in hardware and software or just software, and how precise the arithmetic calculations must be, to still have a functional algorithm. TLM is very useful here as it is easier to do refinements in software for an algorithm, than to debug it in RTL. In case parts of it is implemented in hardware the algorithm has to be refined from floating point to fixed point.

Virtual Software Development Platform

By using ESL and TLM software models a platform can be made to enable early development of system software. During which it can be examined if the hardware have the correct features, if the current architecture have enough control and status register, or if it enables the software to meet its timing budget.

Functional Verification

Since TLM enables communication at different levels of abstraction, TLM models can be fully or partly reused in functional verification of the RTL in question. This is very useful as the use of constrained random test stimuli generation requires the development of reference models in order to check the results of the randomized inputs.

Timing

There is three types of timing a TLM model can have, loosely timed, approximately timed, and un-timed [34]. Loosely timed models have a loose dependency between timing and data. Timing information and requested data is provided when a transaction is initiated. The loosely models are independent of time advancement to produce a response, but this also makes it possible to make them really fast. This makes them extremely useful for software development on virtual Platforms. Approximately timed models have a strong dependency between timing and data, and can depend on internal or external events getting invoked and/or time advancement to produce a response. Synchronizing transactions in correct order before processing is necessary to these models, and they also need to trigger simulations switches, which in turn results in worse performance for these models. Un-timed models are relics from TLM 1.0 and have been deprecated in TLM 2.0. It is possible to make untimed models in TLM 2.0 by using the constant SC_ZERO_TIME which represents 0 time, and is the equivalent of a delta-cycle.

Sockets

Sockets are used to connect paths going forward or backward between an initiator and a target [33]. Sockets supports both blocking and non-blocking transport, usually only one of them are used for one connection. They offer a very convenient way to make TLM2.0 connections. Default transaction type is tlm_generic_payload using the TLM base protocol semantics. The initiator socket is made to be used with an initiator to drive a target through the target socket [33]. The initiator must either implement a backward interface or use a simple initiator socket. The target socket is made to be used with a target to receive a transaction from an initiator [33]. The target must either implement a forward interface or

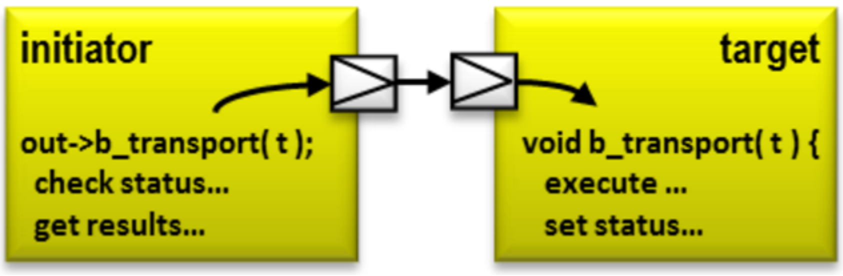

Figure 2.3: TLM blocking Transport [16]

use a simple target socket where only the methods registered with the socket needs to be implemented. This is illustrated in figure 2.3 and figure 2.4.

Blocking and non-blocking transport

There are several interface methods used to pass transactions in TLM2.0, the most important ones being the blocking and the non-blocking interfaces [14]. They are designed to be used together with the generic payload, but can be used separately to model specific protocols [4].

The blocking interface is implemented with one method for targets, b_transport(transaction , timeoffset), and is usually accompanied by the loosely timed coding style [14]. This method can call the SystemC wait function, which responds to time or an event. The time offset parameter will indicate when a transaction is valid in comparison to the current simulation time, when passed from initiator to target. The blocking transport methods pass a non-const reference to the transaction object and a timing annotation [4].

The non-blocking interface is implemented with two interfaces tlm_fw_nonblocking_transport_if and tlm_bw_nonblocking_transport_if, where one is used as a forward path from initiator to target and the other one as a backward path from the target to the initiator [4]. The approximately-timed coding style is what the non-blocking transport interface is intended for. This interface is suited to model detailed interaction sequences between initiator and target during each transaction. The non-blocking transport methods also pass a non-const reference to the transaction object and a timing annotation, but it also passes a phase to indicate the state of the transaction, and returns an enumeration value to indicate if the return from the function also represents a phase transition. Non-blocking transport also supports multiple phases within the lifetime of a transaction. Figure 2.3 shows a blocking interface and figure 2.4 shows a non-blocking interface.

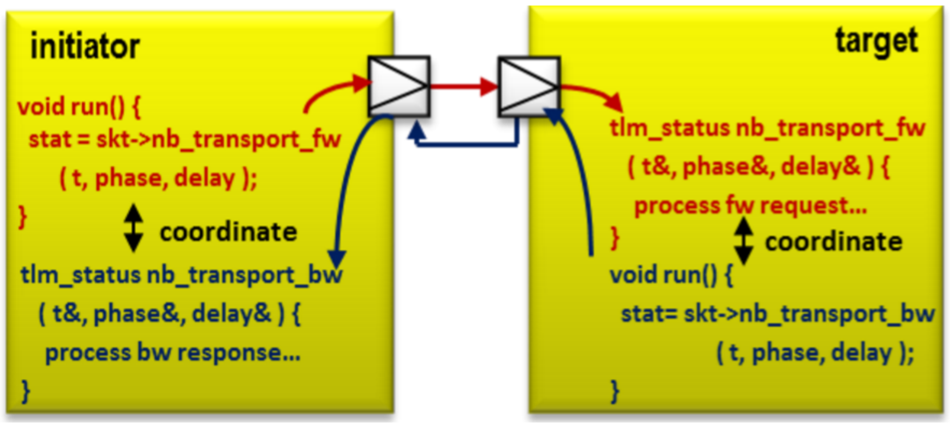

Figure 2.4: TLM non-blocking Transport [16]

TLM Generic payload

TLM2.0 has a standard transaction class, tlm_generic_payload with several attributes; command, address, data, data_length, response_status, byte_enable and byte_enable_length [14] [33]. Command can be either a read, write or an ignore. Address is a base address, a reference point for other memory locations or addresses. Data is a data buffer, organized as an array of bytes. Data_length specifies the number of valid byte-enables in the buffer. In TLM2.0 there is also a base protocol for execution of generic payloads over standard initiator and target sockets. The Generic payload is the default transaction in TLM2.0 for blocking and non-blocking transactions, it represents a generic read or write access to a bus.

2.2.6 SystemVerilog

SystemVerilog is a combination of a hardware description language and a hardware verification language. It is based on Verilog and the advanced verification features found in OpenVera [38]. SystemVerilog have four distinct language subsets; it is an object oriented language for functional verification libraries like UVM or OVM, a design language, an assertion language and a functional coverage language to assertion that a verification environment or a testbench have fully exerted and verified a design [32]. Because of this SystemVerilog spans a large range of domains. Netlists and RTL are covered by the design language subset, general programming is covered by the object oriented features, functional coverage is covered by its respected subset, testbenches are covered by all the subsets except the design subset, temporal properties are covered by the assertion language subset.

Systemverilog was created as a response to the need for a common, open verification language [39]. Verification languages like OpenVera[38] and *e*[38] already existed but they were closed and cost money. A lot of companies did not want to pay for verification tools, so they used a lot of time to create their own. As the design outgrew the verification capabilities of Verilog, a productivity crisis emerged and resulted in the creation of Accellera [1]. Which in turn created SystemVerilog, and also later UVM.

SystemVerilog comes with a large range of different verification features; new data types, classes, constrained random generation, assertions and synchronization.

- New data types: All Verilog data types are included in SystemVerilog but new data types have also been added [39]. Verilog-1995 has two basic 4-state data types: variables and nets. These can be single or multi-bit, signed and unsigned, or floating point numbers. They can also be grouped into fixed size arrays. Systemverilog adds the logic type as an improvement over the old Verilog *reg* data type. The logic data type is a variable but can also be driven by continuous assignments, gates, and modules. It can be used anywhere a net is used except for modeling of bidirectional buses. For improved simulation performance and reduces memory usage, 2-state variables is added. The unsigned bit type, and four signed types, byte, shortint, int and longint. Systemverilog also introduces compact declaration of fixed-size arrays, array initialization using an array literal, dynamic arrays, and associative arrays, all with added array operations and methods. Lastly enumerate, strings, and queue data types are added, streaming operators, type conversion, type defining, structures and unions are included. All arrays and structures can be either packed or unpacked.

- Classes: To ease control and increase reusability in the verification environment, Systemverilog introduces classes. The classes organize functionality and support a single inheritance model. This is what qualifies System Verilog as an object-oriented program.

- Constraint-random: To remove the human fault factor and reach unthinkable states in a design being verfied, Systemverilog includes randomized input variables. These can be constrained by constraints. Constraints are used to exclude illegal values or to test specific parts. They can also be turned off to verify proper error handling of faulty input.

- Communication: Interfaces and modports are used to communicate between the RTL netlist and classes. I/O connections are bundled by the interface, this allows direct access through different levels of the hierarchy, and by that reduce the common issue of spaghetti code. Interfaces can also implement necessary functionality for bus transfer protocols to make bus communication more efficient. This is also the basis for Transaction Level Modeling (TLM).

- Synchronization: *Mailboxes* and *semaphores* are used to synchronize threaded TLM and the RTL signals so that they can be used together. Mailboxes are used for messaging and semaphores are used to control execution order and access to resources.

- Assertions: Methods for checking temporal and functional properties in a design.

The methods can be placed several places in the code for immediate or delayed checking of properties. Assertions are a good way to target specific behavior, by separately describing expected transitions or results for properties one want to verify. A part of SystemVerilog, SystemVerilog Assertions (SVA), is a standardized assertion language that provides a well defined system for assertions.

- Functional Coverage: Coverpoints and covergroups are used to easier track the progress and monitor what exactly has been tested. For random variables this is very important as it shows which functional parts of the design that has been exerted by the input and which parts that must be tested more thoroughly.

2.2.7 Verification Methodologies

The first verification methodologies where based on different languages and created by different vendors to help increase their productivity in verifying designs. They where class based, focused on reusability, and only let the user modify tiny parts of the code. It all started with e Reuse Methodology (eRM), based on e, from Versity Design [38]. Cadence combined it with SystemVerilog and created Unified Reuse Methodology (URM). Synopsis created the Reference Verification Methodology (RVM), based on OpenVera, around the same time as eRM was created. This later became Verification Methodology Manual (VMM), now with OpenVera as a base for SystemVerilog verification. At the same time as URM and VMM was created, Mentor created the Advanced Verification Methodology (AVM) based on a combination of SystemC and SystemVerilog. URM and AVM where later combined into the Open Verification Methodology (OVM), now only with SystemVerilog. This later became combined with VMM to form the Universal Verification Methodology (UVM), which is the standard used today. This is illustrated in figure 2.5.

2.3 UVM

UVM is a common practice for verification which relies on reusing and combining unit-level environments and then running real software on an SoC [19]. It provides a SystemVerilog base class library (BCL) and guidelines which supports the construction and deployment of verification components (VCs) and testbenches that dramatically reduces users coding effort and automatically enforces certain aspects of interoperability [12] [42].

Key concepts of the UVM [12]:

- A simple class hierarchy, rooted in UVM_object,that makes it possible to implement key services.

- Components and data, the two distinct categories for classes in the UVM. Compo-

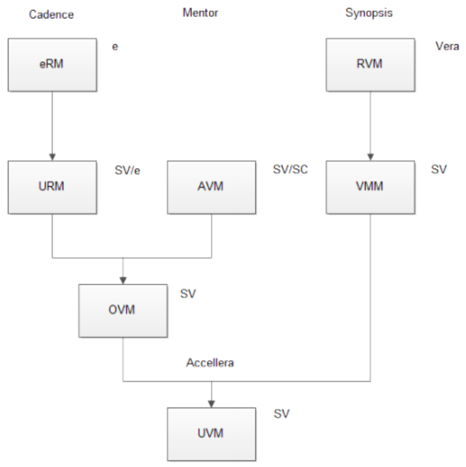

Figure 2.5: Development of methodologies [38]

nents are intended to model permanent, structural parts of a testbench (monitors and drivers). Data are intended to model stimulus, observed transactions, and other data flowing around the testbench.

- An object factory, that automatically creates objects based on either user specification or default settings based on the used classes or derived classes.
- A configuration or resource database, which is a structure that allow configuration values of any data type, including userdefined types, to be stored in a globally accessible table and later retrieved using a string name key.
- Interconnection of components, a SystemVerilog implementation of transaction-level modelling (TLM), which allows VCs to be written to pass data through TLM ports and exports, without regard for the details of other VCs that may be connected to them.
- Sequences for stimulus generation, activity that is coordinated in sequence of random stimuli, both on individual ports and across multiple ports of the DUV. A lot of transactions are built upon the constrain-random functionality enabled by System Verilog.
- Automated code generation using macros, or routine coding tasks automated using macros.

    1. Macros for factory registration, constructs a singleton instance of every object wrapper class, and can use these instances to create instances of user classes on demand.

2. Macros to automate the creation of utility methods, automatically creates utility methods for each derived UVM class created.

A typical UVM structure starts at a top level test, containing an environment. Inside the environment there are subscribers, agents, a scoreboard, and a virtual sequencer. The agents contains monitors, drivers and sequencers. Monitors and drivers connect to the DUT through interfaces, and the environment and agents can be configured by configurations. This is described in figure 2.6.

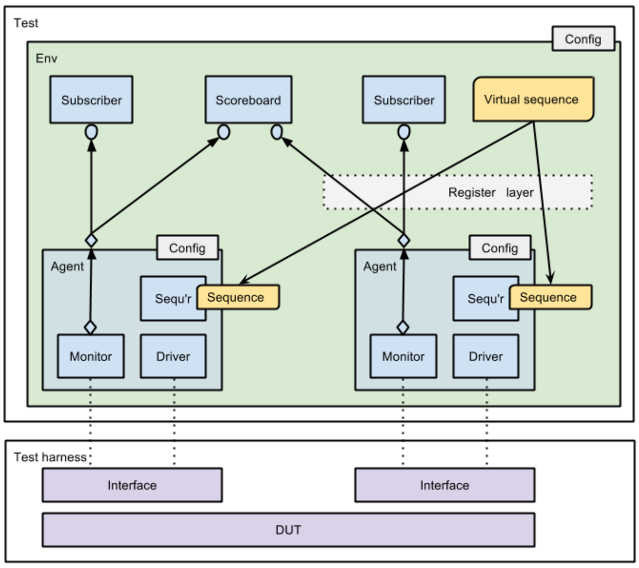

Figure 2.6: The organisation of the verification environment [18]

- Tests are top level components that control generation. This means that they can set the contents of configurations, override components, transactions and sequences by extending them, and start sequencers in specific components [5].
- Environments and agents are simply composed by their contained components.
- A subscriber is just a component with a built in analysis export [5].
- Monitors are components that can observe the communication between the DUT and a testbench, and send transactions to the verification environment [5].
- A sequencer is a component that runs sequences and sends transactions generated by those sequences to another sequencer or driver [5].
- A driver is a component that receives transactions from a sequencers and forwards this to a DUT through its interface [5].

2.4 SystemC Modeling

To make ESl models and TLM possible, SystemC has arisen as a perfect candidate. SystemC is a C++ class library, where the C++ language contains the ability to model software, and SystemC completes this ability by enabling modeling of hardware[8]. Together they are perfect to make ESL models. Many algorithms and applications are already implemented in C++ or in C and by wrapping them in a suitable SystemC wrapper, it can be a good ESl model. SystemC is an IEEE 1666 standard, which means that it is open to the industry and therefore have certain benefits, like access to commercial and freeware based tools, and support. Since SystemC is based on C++ it is fast enough to be used for early software development. Software development is usually the last part being done in a modern hardware development, but by using ESL models in SystemC, that part can be started sooner, which can make a design process more efficient, as software development can be finished faster. SystemC has a simulation kernel which supports parallel/concurrent executing, this is important since hardware runs concurrently. TLM gives SystemC the ability to communicate with other languages like SystemVerilog, and that also enables SystemC modules to be used as reference models in a SystemVerilog test-suit.

2.4.1 Class concept for hardware

SystemC have certain hardware-constructs that are required to model hardware in an environment mainly used for software development. The constructs are all implemented in C++, and enables concepts of time, hardware data types, hierarchy and structure, communications, and concurrency.

Time Model

SystemC uses the class sc_time to obtain current time and to implement delays. The class has a 64bit resolution. It contains an enumerated type that defines natural time units like seconds, nanoseconds and the like. There is also a class called sc_clock for models that require clocks.

Hardware data types

Hardware requires more flexibility in data types width than C++ has native, so SystemC has support for data types with explicit bit width for integral and fixed-point quantities. Both binary and non-binary representations, like tri-state and unknown, is supported.

Hierarchy and structure

To create a hierarchy SystemC uses module entities connected to other modules through channels. Module classes can be instantiated within other modules.

Communications management

The SystemC channel can represent both simple communication like wires or FIFOs and complex communication schemes to map to special hardware. The different channel implementations can be used interchangeably. The library has common software and hardware channels, like FIFOs, signals, mutex and semaphores, built in to it. Port classes are used to connect modules to other modules or channels.

Concurrency model

There is no true concurrency in hardware simulation, but single units execute until simulations of other units are required to continue with correct alignment in time. To determine this in the simulation code, events are used as switches. The simulator swaps between concurrent elements.

2.4.2 Simulation Kernel

The SystemC Simulation Kernel has three phases, elaboration, execution, and post-processing or cleanup. Elaboration is where data structs are initiated, connections are established, and anything else needed to prepare for the next phase, execution. This is all the statements before the sc_start() functions is called. In the execution phase the simulation kernel controls the execution of processes so that they appear to be concurrent. Figure 2.7 shows an overview of the simulation kernel. First the elaboration process, then sc_start() is invoked and all simulation processes, minus a few exceptions, are initialized in an unspecified deterministic order. Then code will be continuously evaluated for events or updates while also advancing the simulation time when no processes needs to be evaluated. When there is no more processes to run, the simulation ends, and a cleanup is invoked.

2.4.3 Threads and Methods

Simulation processes in SystemC are member functions of sc_module classes, registered with the simulation kernel, and the kernel is the only caller of these functions. Theses functions have no arguments and no return value. Other processes, that are not executed by the simulation kernel, are invoked as function calls within the simulation processes

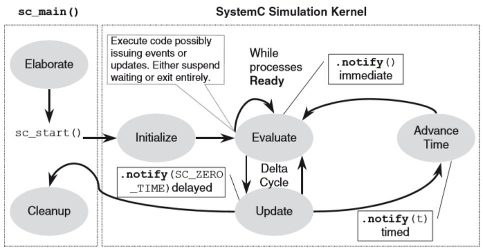
sc_main()
SystemC Simulation Kernel
Elaborate
sc_start()
Cleanup
Execute code possibly issuing events or updates. Either suspend waiting or exit entirely.
While processes Ready
.notify() immediate
Initialize
Evaluate
Advance Time
.notify(SC_ZERO_TIME) delayed
Delta Cycle
Update
.notify(t) timed

Figure 2.7: SystemC simulation kernel [8]

of the sc_module class, like C++ functions or class methods. There are three types of simulation processes SC_METHOD, SC_THREAD, and SC_CTHREAD. SC_METHOD is a timeless function with no arguments, no return value, and can be repeatedly be called by the simulation kernel. It is basically a C++ function. An SC_THREAD can only be invoked once, but can self suspend and allow potential time to pass before continuing, similar to the execution of a software thread. SC_CTHREAD is similar to a SC_THREAD but is required to be clock sensitive.

2.4.4 Events, Sensitivity and Notifications

Events are caused by the event class member function, notify, and invokes SC_METHODs and SC_THREADs that are sensitive to the event. They are implemented as sc_event and sc_event_queue SystemC classes. Sensitivity in SystemC is either static or dynamic. Dynamic sensitivity can change the simulation sensitivity during a process, while static can not. An SC_METHOD or SC_THREAD can switch between dynamic and static sensitivity during simulation.

2.4.5 Channels and interfaces

SystemC processes communicate using channels, events or through module boundaries. Modules connect through ports, and interconnects through channels. Certain channels and interfaces are finished implemented in SystemC. Worth mentioning is sc_mutex, sc_semaphore, sc_fifo, and sc_signal.

2.4.6 Modules and Hierarchy

SystemC separates interface and implementation, C++ header files(.h) are used for entities and C++ implementation files(.cpp) are used for architecture. Design components are represented as modules, which are classes inheriting from the sc_module base class. Modules can contain other modules, channels, processes and ports.

2.4.7 Data types: Logic, Integers, Fixed point

SystemC supports all C++ data types, but also include support for non-binary hardware types as four-state logic (0,1,X,Z), and lets you define new data types. All the new data types have a large operator overload so they can be used almost as easily as C++ data types. Conversion from hardware to hardware or hardware to software data types, and all other necessary methods to use hardware data types, are provided in SystemC.

2.5 UVMC

UVM connect (UVMC) is a UVM open-source library made by Mentor Graphics [23]. It provides TLM1.0 and TLM2.0 connectivity to pass objects between SystemC and SystemVerilog models and components. A UVM Command API is also provided for access and control of the UVM simulation from SystemC.

2.5.1 Enabling IP and VIP reuse

UVMC enables use of SystemC models as reference models in UVM, or reuse of stimulus generation agents in SystemVerilog for verification of SystemC Modules [23]. This in turn enables more VIPs/IPs to be used, and leverage different strengths in each language. Through the UVM Command API, SystemVerilog UVM can be accessed from SystemC.

2.5.2 Key features

UVMC simplifies the connection between the languages by supporting standard UVM, not requiring models or transactions to inherit from a base class, supporting existing TLM models in both languages without modification, and therefore allowing independent models to be reused as they can communicate without being directly referred to each other [23].

2.5.3 Making UVMC Connections

To make cross language TLM connections, UVMC provides connect and connect_hier functions [23]. In SystemVerilog TLM2.0 they are written like
uvmc_tlm#(trans)::connect/connect_hier(port_handle, ”lookup”), and in SystemC TLM2 like uvmc_connect/uvmc_connect_hier(port_ref, ”lookup”). Trans is only used for SystemVerilog to specify the unidirectional TLM transaction type. The port_handle/ref is a handle or reference to the interface, port, export, imp or socket instance to be connected.

The lookup is an optional string used for matching ports together no matter the language used. When the string matches for two ports, those ports are being connected not dependant on if the components are of the same language or a different language.

2.5.4 Transaction Conversion

TLM generic payload is supported by UVMC, but if a different object is transferred a converter is required. UVMC use separate converter classes to pack and unpack transactions, which in turn allows converters to be defined independently from the transactions they operate on. UVMC defines default converter implementations that use the standard methodology for each language.

2.6 Questasim

Questasim is a simulator that has native support for SystemVerilog Testbenches, UPF, UCIS, OVM/UVM, and SystemC [25]. It includes mixed-language capabilities, advanced debugging capabilities, and a single simulation kernel that supports all standard verification languages.

Chapter 3

Related work

In the last couple of years there has been done much work in regards to verification of SoCs and heterogeneous systems. A summary of them will be presented here. Parts of this chapter have been fully or partially reused from the project preceding this thesis, since this is information of equal relevance to the dissertation [20].

3.1 Previous works in regards to verification of complex heterogeneous systems and SoCs

Institute of VLSI Design, Hefi University Of Tech, released a paper in 2004 about ”Reuse Issues in SoC Verification Platform [41]” where they discuss the issues when reusing IP verification components on a SoC. There are two things to consider when verifying a SoC. Interconnections of IPs where focus is on behaviour in comparison with SoC specification, and unexpected interactions between IPs where focus is on chip integration. In the verification of a single IP, bus function models(BFMs) are used as high level models in replacement of real buses. The BFMs can be reused on other IPs with the same bus as it was originally intended for, but it can also be used on other buses with small modifications. In a SoC the BFMs are replaced by real buses. Monitors are used to transform events at pin-level into transactions, which are high-level abstractions that eases reuse and increases verification productivity. Monitors should be independent and not rely on input from other monitors. This to ease the reuse of the monitors. The monitors can be completely reused from IP to SoC verification as they are based on the same buses. But if they are reused, they should keep the internal signals used when testing in the synthesis process. Drivers are used to transform transactions to pin-level signals. If the driver is used as a peripheral module it can not be reused in SoC verification, but if it is connected to external pins, it can. Simulation patterns is a connection between the input signals to the IP and the system service. The system service contains system tasks, verification scripts, tools, and such. As long as the system tools is kept fairly similar from IP to SoC verification, the simulation patterns can be reused as stimuli, but may have to be modified. The strategy is to make the IP verification platform as similar to the SoC verification platform, so that reuse is fairly simple. Figure 3.1 presents the structure for verification of a single IP based on buses.

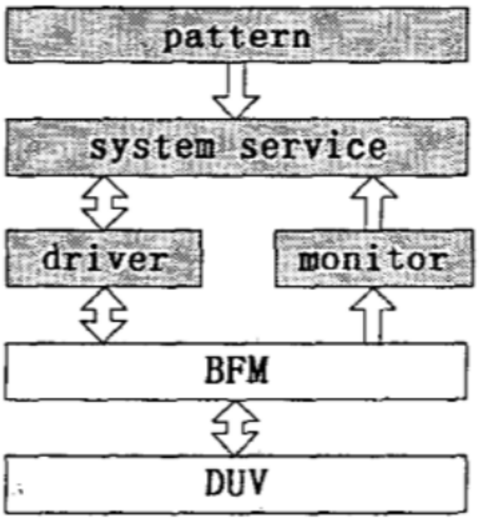

Figure 3.1: Bus-based structure of IP standalone verification platform [41]

The telecommunications laboratory at the National Technical University of Athens released a paper in 2003, ”Verification of a complex SoC; the PRO^3 case-study [3]” where they presents some aspects, in the architecture design, used to support the observability and verifiability of a system. Minimization of the data paths is an important aspect as long data flows are time consuming, hard to observe, and poses a difficulty in identifying a source of error in case of failure. Shared packet buses are another important aspect to consider. In a shared packet all control information related to the SoC, such as destination,source, message type, and command, is integrated in the packet header. Shared interconnect structures eases observability since all data is transferred on the same bus. Lastly an embedded hardware block for monitoring a shared bus enables the ability to control and observe. Also described is a verification procedure where designers are told to write down detailed specification documents from the design specification. A verification is done on the documents to avoid misunderstandings and interface mismatches. Sophisticated test benches are used on the block level, and generic, intuitive test benches are used on the external level. In between these, a hierarchical verification is used on the integration of one block with other blocks, including all the interfaces and interconnects related to the blocks.

3.2 Industry verification approaches

In 2002 Guy Mosensoson from Verisity Design, Inc released a paper on ”Practical Approaches to SoC Verification [36]” where he presents methods to approach SoC verification and the challenges in relation to this. He mentions the importance of reuse of verification IP/components to get fast, minimal effort, automation in the verification of an entire system. Good tools to ease the verification of the separate components is also of great importance. A verification component should be present for each component/unit in an SoC.

The unit verification component should check for internal blockers and provide stand alone coverage metrics, as these are useful for the full system verification. The old test plans for SoC where multiple sets of directed tests. These are very inefficient and inaccurate. To improve this, flexible and easily modifiable generic tests are used as a replacement.

Another thing to consider when verifying a SoC is the HW/SW co verification. This because of unexpected HW/SW sequencing bugs, HW/SW dependencies, and general increase in design time when doing verification of each part in sequence. To do this, real software is run on simulated HW. The simulated HW works as high level references, verification shadows, that the designed HW can be compared against when doing verification. Integration between components is one of the biggest challenges in verification of a SoC, blocks that where assumed verified show up with bugs, conflicts happen when accessing shared resources like memory, arbitration problems and deadlocks on the bus appears, and priority conflicts happens when exception handling is performed. To handle this the verification environment must focus on high level transactions, and be able to verify components in parallel streams, and check their transactions if they are on the same bus. Other things the paper mentions that a verification environment should have are high level abstract descriptions of every component that has to be verified and self checking of these. Also, changes done in the SoC when verifying should be done in the verification environment and should by that avoid changes in the tests.

A paper released in 2003 by Yves Mathys and Andr Chtelain on ”Verification Strategy for Integration 3G Baseband SoC [31]” they mention the importance of verification on different levels when verifying a SoC. They use a top-down, bottom-up approach where they verify in different levels. At the IP level they focus on the functionality of a single IP, and use application stimulus to ensure correct behaviour in comparison with the system specification. At the RTL level SystemC models describing the system are used as reference to compare with. At the SoC level the platform integration for components are verified on all kinds of buses and connections. Worth mentioning are signals connectivity, memory mapping, data paths, DMA, interrupts and inter-process communication, as these are the most used connections and buses.

A practical and efficient SoC verification flow by reusing not only the IPs test bench but also the IPs test case, is presented in the paper ”Practical and Efficient SoC Verification Flow by Reusing IP Testcase and Testbench [43]” written by researchers from Connectivity Solutions. Because of this added reuse SoC verification and debugging becomes less complex and less difficult. This in turn increases verification throughput using less resources. The flow consists of two protocols interfaces, IF_A connected to top chip pads and IF_B, communicating between IPs inside the SoC. Also included in the flow is a System Control Interface, IF_C, that consists of control signals. When either IF_B or IF_A is initiated a complementary verification IP (VIP), is created in the UVM testbench. VIP is a configurable verification component that is encapsulated and follows a consistent architecture for stimulus generation, coverage collection and protocol checking. For IF_C a testcase or testbench is sufficient to generate stimuli for control signals. As long as IP testbenches and testcases are all designed for reuse, they can fully or partly be reused for the SoC testbenches and testcases. Reuse has to be planned for by both IP engineers and SoC engineers, but this collaborative work makes it easier to determine if an error is in the SoC design or the IP design. It also allows IP and SoC engineers to work in their specialized field without having to worry about details outside of their respective fields. There are three different ways to reuse verification component and test case files from IP level to SoC. The first one is in regards to IF_A, where VIP and test files for the IP can be reused, without modification, for the protocol interface connected to the chip pads. Second, the IF_B VIP have to be configured from Active to Passive mode to be reused in the testbench. This is due to the fact that the bus it was normally driving for one IP is now driven by another IP. It does however require a new SoC testcase, as the testcase for the IP is disabled in Passive mode. in the third case IF_C is either controlled by the SoC and it's test case is used to drive the interface. Or it is controlled by another IP and it's VIP and testcase is reused to drive the interface, or it is connected to the top chip pads and the SoC testbench is used to drive and generate stimuli.

3.3 ARM verification

Verification libraries with common test codes, used in base functions to test building blocks in the DUV, are highlighted in "Verification Methodology of Heterogeneous DSP+ARM Multicore processors for Multicore System on Chip [11]" as it is the primary method for test development described in the UVM design test flow. Because test generators will pull a specific test code directly from the library, if needed in an individual test case, copies of a specific test code are prevented to appear in the test directory. In other words, the use of libraries ensures that there is only one copy of a test code. Automatic test generators can generate new test cases from existing test code and by doing this changes or modifications done to the existing test code will propagate to the new cases and update these. It works like a inheritance system where changes done to any test code used by other test cases will be present in all cases. This gives a reduced code effort. Elements from the library are used for test cases performed on the DUV, and consists of manipulators that causes a certain state to exists in a target, and checkers which either checks if data exists in a memory location, if data matches a specific pattern, or a sequence of patterns that can give proof of an event. The combination of manipulators and checkers are used for complex interactions. Test cases are classified in a library structure divided firstly into sub modules and secondly into operation so that similar tests are kept in the same directory. Some classifications are, Smoke which are tests that give a basic indication of a working DUT, Functional which are tests which perform functional coverage, Performance which are tests to show the capabilities of the DUV for specific operations relevant to the DUV, Benchmarks which are test to generate standardized performance metrics, does not give any design coverage. Other things highlighted is that many test at unit level reduces the amount of tests at device level. This increases performance and speed of tests and reduces verification time. Also if several blocks are equal only one needs to be thoroughly verified, as this enhances simulation performance.

An example of an ARM Advanced Microcontroller Bus Architecture (AMBA) Advanced Peripheral Bus (APB), being verified using UVM is described in the journal ”Design and Verification of AMBA APB Protocol” by Shankar, Dipti Girdhar and Neeraj Kr. Shukla at the ITM University, Gurgaon, India [26]. Here we can see the a standard UVM setup being used, with test, environment, scoreboard, sequences, sequencer, driver, monitors and agents. The design is assumed to be correct as it functional and code coverage is correct. Figure 3.2 shows the setup.

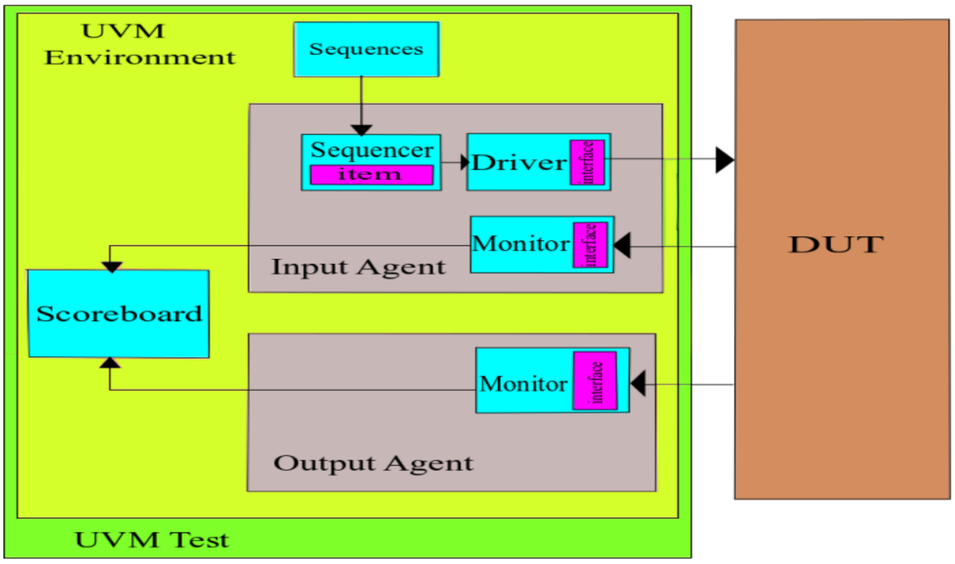

Figure 3.2: AMBA APB UVM setup [26]

3.4 Summary

To summarise all the highlights from the related works section, when it comes to SoC verification, we have that the verification results at unit level should be reusable in the verification of the interconnects and at a higher level, and the tests themselves should be reusable at the unit level. The tests should work like a hierarchy where tests at any level are configurable and modifiable and also partly or fully reusable in other tests. Changes in the tests must automatically propagate to other tests throughout the design that uses something from these tests. This builds upon the common verification libraries which ensures that there is only one copy of a given test code, and that other test codes or test generators which pulls code from the library always are updated with the latest changes to any code. By using a high level description of the circuit at different levels, we have a platform to test software on and a reference design to compare the verified hardware to. This makes it possible to design HW and SW simultaneously, and to test efficiently. Many and thoroughly tests at unit level decreases testing effort at a higher level, also copies of a unit is not verified twice.

Chapter 4

SHMAC verification methodology

4.1 Analysis of the current verification strategies

The current verification strategies for the SHMAC platform present several limitations, which were identified and analyzed on the project preceding this thesis [20]. One of the main shortcomings involves the use of directed tests in both testbenches and bare-metal testing on FPGA. Hand-coding all the possible test cases not only represents a time-consuming and tedious task for the developers, but also usually leads to verification incompleteness. The reason for this is that not all the possible scenarios and corner cases can be easily anticipated by the designer and included as test stimulus.

Other deficiencies of the actual verification strategy were identified by conducting a survey among SHMAC designers, which is shown in **Appendix G**. They were asked about what kind of verification approach they used, what difficulties or challenges they faced and what features they suggest to be included in a new verification framework. Some developers agreed that the documentation of the existing testbenches and the framework for running bare-metal tests is not sufficient and clear. Also, some others claimed that there is a lack of debugging mechanisms to effectively identify and analyze errors in the designs. In addition, it was found out that the limited error diagnosis capability is one of the main obstacles software developers of the SHMAC face, since it makes it difficult to distinguish between hardware and software bugs.

In order to be able to bring the SHMAC platform up to leading industrial quality standards as well as to enable future upgrades and modifications to the platform to be quickly integrated and verified, a new verification methodology is needed.

4.2 Proposed verification methodology

The proposed verification methodology consists of a sequence of steps to be followed when designing and testing any module of the SHMAC platform. These are the following:

1. Elaboration of a circuit description document: the developer should write a document which provides a description of the structure and operation of the module to be implemented.

2. Hardware design using design for verification practices: the developer should take into account design for verification practices when coding the hardware description of the module, in order to make the code as clear as possible as well as to be able to identify any important design feature that needs to be tested.

3. Elaboration of a verification plan document: the developer should write a document which provides a list of the main design features to be tested in the module.

4. UVM-based functional verification: the testbenches developed for the module should be based on the Universal Verification Methodology (UVM) and maximize reusability of previously implemented verification components.

The following subsections explain further these stages of the proposed verification methodology.

4.2.1 Elaboration of a circuit description document

The circuit description document essentially describes the architecture and the functionality of the unit to be implemented. It should be used as a guide when coding the hardware description of the module as well as when implementing the verification plan. Examples of this document can be found in **Appendix A** and **Appendix C**, which correspond to the circuit description documents of the SHA1 core and the SHMAC router, respectively. They are both divided into three sections: overview, external interface and operation. The first provides a general description of the module, the second describes the input and output ports, while the last one gives an explanation of the high-level behaviour of the module.

4.2.2 Hardware design using design for verification practices

Design for verification is a set of rules and guidelines that defines easy tasks designers can perform so verification quality increases [2]. As main rule, register-transfer level (RTL) code must always be maintainable, possibly reusable, and follow standard coding rules. It should also be understandable by both outside designers and the designer who made it. In addition, peer review of behaviour scenarios, corner cases and areas of risk is important as it helps to avoid obvious bugs and ensures correct verification focus.

Additionally, high value assertions must be identified during the design. As the design evolves, a growing number of assertions should be added, and used to check for correct behaviour. The assertions help to eliminate bugs, reduce debug time, capture designers intentions and assumptions, encourage a more thorough thought process about intended behaviour, and ensure that future changes do not corrupt expected behavior. A summary of these and other design for verification guidelines can be found in **Appendix E**.

4.2.3 Elaboration of a verification plan document

The verification plan is a list of all the design features to be exercised and tested on a module, so that it can be used as a guide when implementing the testbench of the module. Examples of this document can be found in **Appendix B** and **Appendix D**, which correspond to the verification plan documents of the SHA1 core and the SHMAC router, respectively. They are both divided into three sections: overview, test plan and UVM-based suggested verification environments. The first gives an overview of the document and describes the parts of it. The second includes a table which specifies all the design features to be tested. This table contains 4 columns: section, description, coverage type and priority. The first indicates the category of the feature, the second describes it, the third one indicates the type of coverage that should be used (cover group, design assertion, or test result if no coverage metric is used) and the last the relative priority of the feature. Finally, the suggested UVM-based verification environments sections includes recommended tests and verification environments that can be run in UVM in order to check the design features listed on the previous section.

4.2.4 UVM-based functional verification

The UVM enables efficient development and reuse of verification environments, and also makes it possible to measure the verification progress (based on the verification goals in the test plan) as it combines automatic test generation, self-checking testbenches and coverage metrics. This does not only ensures a thorough verification of a design, but also reduces effort and time spent creating hundreds of tests (as done in the directed test methodology).

A common approach for implementing UVM-based tests in a complex system as the SHMAC is the bottom-up methodology. It consists of creating a hierarchy of tests that ranges from the unit level to the system level. Each test must be implemented in such a way that reuses verification components of the lower verification levels. The following are the suggested verification levels for testing any module in the SHMAC platform:

1. Unit level test: verifies an individual component or unit in isolation before it is integrated to its corresponding tile on the SHMAC platform. For instance, it can consist of the verification of a new hardware accelerator, either embedded on a CPU

tile or on an independent tile.

2. Tile level test: verifies a complete tile within the SHMAC architecture by checking the correct integration of its internal components. For instance, it can consist of the verification of a tile containing a hardware accelerator and a router.

3. Top level test: verifies the correct integration between different tiles within the SHMAC architecture. For instance, it can consist of the verification of an accelerator tile interacting with a processor and memory tiles.

In addition, efficient UVM coding is also important when implementing a verification framework. There are common practices and conventions used in industry which allow to produce consistent, readable and reusable code. They mainly consist of class naming conventions, as well as general rules for reusable stimulus generation, functional coverage collection and message reporting. **Appendix F** contains a set of these UVM coding guidelines, which are based on [17].

Chapter 5

SHMAC verification framework

The implemented framework is mainly focused on the complete verification of the SHA1 accelerator, which is one of the main co-processing units of the SHMAC platform. Since the SHA1 core interacts with a CPU core and the main memory, its verification includes integration tests with these units which exercise many features of the SHMAC architecture. The same verification approach employed for the SHA1 can be applied to any new module incorporated to the SHMAC.

The SHA1 core verification process can be divided into four stages, which test the SHA1 operation at different levels. These are the following:

- SHA1 core test: is aimed at verifying the SHA1 accelerator as a single unit, isolated from the rest of the SHMAC modules.
- Router test: evaluates the correctness of the SHMAC router, so that it ensures the absence of bugs in data transmission. The verification environment built for this test also allows to monitor internal signals when reused in the following tests.
- SHA1 tile test: performs a test of the complete SHA1 tile, which involves the integration of the SHA1 core and the router. It reuses the verification environments implemented for both units.
- Top level test: verifies the correctness of the integration between a CPU core, the main memory and the SHA1 core. It employs a program written on memory which performs a SHA1 computation.

These tests are further described in the following sections.

5.1 SHA1 core test

The SHA1 accelerator employs the SHA1 algorithm to hash an input message of variable length and produces a fixed-length message digest. A complete description of the SHA1 core operation can be found in **Appendix A**. This document provides information about the SHA1 external interface and high-level behaviour, which is relevant for designing a verification plan.

The verification plan for the SHA1 core, included in **Appendix B**, was designed with the aim of testing different features of its functionality under different scenarios. As mentioned in the test plan, input data is suggested to be organized as a random block of 512 bits (16x32), which is the same size of the message block in the SHA1 algorithm. Larger input data blocks can also be used, but they will not exercise any additional functionality and may also increase the overall simulation time. The start address and size words needed for the configuration of the SHA1 engine are also generated randomly, so that a random portion of the 512-bits input block is selected for each test.

The suggested test goals in the verification plan are grouped into three different categories. The first is associated with the use of different sets of input data. One special case is when the size of the input block is between 56 and 64 bytes, since the SHA1 engine needs to pad the input message with additional zero bits, as required by the SHA1 algorithm [37]. The second group contains goals related to the correct operation of the status bits, and the last includes those associated with other relevant properties of the SHA1 operation.

Each verification goal listed in the verification plan has a certain associated coverage type and priority. The first one consists of checking that the SHA1 is able to operate with any random input message. Since there are many possible combinations of input data and none of them represents a corner case, no cover groups are used and the checking is done by obtaining satisfactory output data. The second and third test goals involve different combinations of start address and size input parameters, and need the use of cover groups for measuring coverage. As explained before, the case when the size of the input block is between 56 and 64 bytes is considered a corner case. The rest of the goals require assertions to verify specific properties of the SHA1. In addition, the priorities assigned to each verification goal indicate the relative importance of each one. The ones with priority 1 are considered to be essential for the SHA1 verification since they ensure that the core can operate correctly with any input data. The goals with priority 2 are also relevant, and make it possible to ensure complete correctness of the SHA1 core operation.

The following subsections describe the SHA1 SystemC reference model built for generating the expected output data necessary for performing the functional checking, as well as the UVM-based environment implemented for the verification of the SHA1 core.

5.1.1 SHA1 SystemC reference model

To make a SystemC reference model of the SHA1, an already written C++ algorithm/model was taken as a basis. The C++ algorithm was wrapped in SystemC together with TLM2.0. Then it was connected to a SystemVerilog module through UVMC. Communication between the modules went without problem, but communication between the SystemVerilog module and the UVM framework was never completed. This was because fatal errors where found, and they could not be fixed on time.

SHA1 C++ function

The SHA1 function takes in input from an update function and uses this to update a ”final” function, which then returns a string. TLM 2.0 generic payload transfers data as unsigned char* so the input data is converted from this to a string through a for loop using the ostringstream stream operator. The string is then converted, through a conversion function, from a binary string to an Ascii string before it is inserted in the SHA1 function. The SHA1 function returns a hex string, so this is converted again, through another conversion function, to a binary string. Then this data string is converted to a char unsigned [vector] and sent out through the analysis port.

SystemC wrapper for the SHA1 core

[33] The wrapper is implemented according to TLM 2.0 as a class. The wrapper is designed to receive and process data into a simple C++ data. A simple_target_socket is used as input port. It uses a TLM generic payload by defaults, and only needs to be instantiated with the class name. As an output port we use a tlm_analysis_port with a tlm_generic_payload as this can connect to the analysis port of the scoreboard. Because the function is a normal C++ model and will finish ridiculously fast, the blocking transport is used. It is registered with the input port and is defined in a function, b_transport. In the function the generic payload functions get_data_length() and get_data_ptr() is used to get the length of the input data and the start address to the input data. Then when the SHA1 function is finished the generic payload function set_data_ptr is used to set the start of the output array and the generic payload function set_data_length is used to set the length of the array. After this a wait() function is used with a constant, SC_ZERO_TIME, that represents 0 time delay, to have a delta time-out before the generic payload is written to the analysis port.

Connecting to UVMC

To connect the SystemC wrapper to a SystemVerilog component, the wrapper is instantiated in a top level function sc_main, and then each of the wrapper ports are put into a uvmc_connect function together with a string. The string is used as an identifier to identify which ports that connect to which ports across the different languages. Lastly sc_start function is called to start the simulation, it is called without any arguments so that the scheduler will run until it completes.

The same is done on the SystemVerilog side. The component sending data to the wrapper has to be instantiated in a sv_main top level function. Then a connect function from the uvmc_tlm class library, uses a port and a string to connect that port to the desired port, with the string as an identifier. Lastly a run_test function with no argument is used to start the simulation.

The SystemVerilog component sends data to the SystemC module which translates, which processes the data, computes new output which is translated and sent back to a SystemVerilog module.

5.1.2 SHA1 UVM-based verification environment

The verification environment built for the SHA1 core follows the test plan previously presented. However, due to time limitations of the project, it is only aimed at checking the design features with the highest priority (priority 1) specified on the test plan. Figure 5.1 shows the test environment implemented, also included in **Appendix B**. It essentially runs tests which generate random blocks of input data, as well as random start addresses and sizes, that are then driven to the SHA1 core. The hash values produced by the core are finally compared with the expected ones. The following subsections describe the operation of each relevant UVM component.

Sequencer

The transactions employed by the sequencer include the following variables:

- rand logic [31:0] input_array [0:ARRAY_SIZE-1]
- rand logic [31:0] size
- rand logic [31:0] start_addr
- logic [31:0] hash_array [0:4]

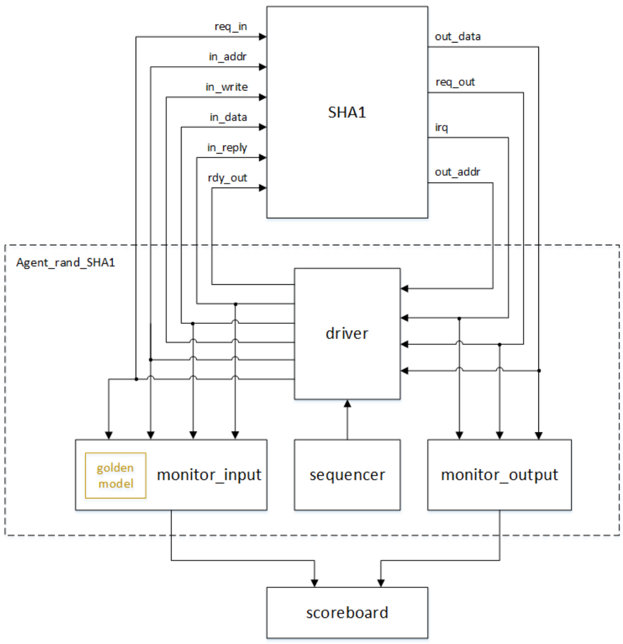

Figure 5.1: Verification environment built for the SHA1 core

The variable *input_array* represents the input array of data to be sent to the driver. Its length is defined by the constant *ARRAY_SIZE*, which has a default value of 16, since the suggested size for the input data block is 512 bits (16x32). The variables *size* and *start_addr* represent the size (in bytes) and start address of the portion of the input data block to be hashed. The following constraints are employed for these values:

- constraint size_c { size inside {[1:(ARRAY_SIZE*4)]};}
- constraint start_addr_c {
  if (size[1:0] == 2'b00) {
  start_addr >= 0 && start_addr <= (ARRAY_SIZE - (size/4));
  } else {
  start_addr >= 0 && start_addr <= (ARRAY_SIZE - (size/4) - 1);
  }
  }

The contraint *size_c* indicates that the variable *size* must be randomized with a value in the range from 1 to *ARRAY_SIZE*4* (64), since the size is expressed in bytes. The constraint *start_addr_c* points out that the variable *start_addr* must have valid values so that a portion of the *input_array* with a given *size* is selected.

Finally, the variable *hash_array* represents the five hash values obtained as a result of the SHA1 computation. A certain number of these transactions (100) are randomized, with the constraints previously defined, and grouped into a sequence, which is sent to the driver.

Driver

The run_phase of the driver is implemented as a sequence of states which get the data contained in each transaction and configure the core to run a SHA1 computation, and finally store the result back to the same transaction. The following listing describes the high-level operation of the driver.

Listing 1: Pseudo-code of the run_phase of the driver of the SHA1 core test

```
1  Reset the DUT.
2  Get a new transaction from the sequencer.
3  Overwrite the start_address register with the value specified in start_addr.
4  Overwrite the size register with the value specified in size.
5  Overwrite the status register with any value (e.g. 1) to start the SHA1 computation.
6  for $i < size$ do
7      Wait until the core requests data (req_out = 1), and provide a new word from the
       input_array based on the address (out_addr) indicated by the core.
8  end
9  Wait for the core to finish the SHA1 computation (irq = 1).
10 for $i < 5$ do
11     Read the $H_i$ hash register and store the hash value in the ith element of the
       hash_array.
12 end
13 Finish the current transaction and go back to step 2 until the sequencer stops sending new
   transactions.
```

Input Monitor

The run_phase of the input monitor is also implemented as a succession of states. It essentially monitors the input signals shown in figure 5.1 and extracts information in order to get the values of the variables of the current transaction. Based on these values, it generates a prediction of the expected hash outputs by employing the SHA1 SystemC reference model described previously, and stores these hash values in the *hash_array* of the current transaction, which is then sent to the scoreboard. In addition, it makes a functional coverage analysis of the transaction. Listing 2 describes the high-level operation of the input monitor.

Listing 2: Pseudo-code of the run_phase of the input monitor of the SHA1 core test

```
1 Wait until (req_in = 1 & in_write = 1 & in_addr = 0), and store the value of in_data in the
    start_addr variable of the current transaction.
2 Wait until (req_in = 1 & in_write = 1 & in_addr = 4), and store the value of in_data in the
    size variable of the transaction.
3 for $i < size$ do
4     Wait until (req_in = 1 & in_reply = 1), and store the value of in_data in the ith element
        of the input_array of the transaction.
5 end
6 Calculate the expected result of the SHA1 computation by using the SHA1 SystemC
    reference model, and store it in the hash_array of the transaction.
7 Make a coverage analysis of the transaction (size and start_addr are used as cover points).
8 Send the transaction to the analysis port, which is connected to the scoreboard.
```

Output Monitor

The run_phase of the output monitor follows a sequence of states which monitor the output signals shown in figure 5.1 and extract the hash values produced by the core. These are then stored in the *hash_array* of the current transaction, which is sent to the scoreboard. Listing 3 describes the high-level operation of the output monitor.

Listing 3: Pseudo-code of the run_phase of the output monitor of the SHA1 core test

```
1 Wait until the core finishes the current computation (irq = 1).
2 for $i < 5$ do
3     Wait until (req_out = 1), and store the value of out_data in the ith element of the
        hash_array of the transaction.
4 end
5 Send the transaction to the analysis port, which is connected to the scoreboard.
```

Agent

The sequencer, driver and monitors previously described are constructed in the build_phase of the agent. Also, two analysis ports are created in order to connect the monitors to the scoreboard. The connect_phase connects the sequencer to the driver, and also each monitor to its respective analysis port.

Scoreboard

The build_phase of the scoreboard creates two analysis exports that are used to retrieve transactions from both monitors, as well as two FIFOs which are employed to synchronize the transaction streams coming from the two monitors. The connect_phase connects each analysis export to its respective FIFO, and finally the run_phase gets the two current transactions from the FIFOs and compares them. If they match, a message indicating a satisfactory result is printed, or otherwise an error message is generated.

Environment

The build_phase of the environment instantiates the agent and the scoreboard, while the connect_phase connects each analysis port of the agent with the respective analysis export in the scoreboard.

Test

The build_phase of the test instantiates the environment, while the run_phase starts the test by enabling the generation of a sequence of transactions which are sent from the sequencer to the driver.

Top block

The top block instantiates the DUT (SHA1 core) and creates a virtual interface which holds all the input and output signals of the DUT. This interface connects the DUT with the driver and the monitors. In addition, it generates the clock signal which is driven to the DUT and the virtual interface, and finally runs the test.

5.2 Router test

The SHMAC implements a 2D mesh-based Network-on-Chip (NoC) interconnect in order to provide efficient on-chip communication. Each tile in the SHMAC has a router which allows the communication with the neighboring tiles. A complete description of the router operation can be found in **Appendix C**. This document provides information about the router external interface and high-level behaviour, which is relevant for designing a verification plan.

The verification plan for the SHMAC router, included in **Appendix D**, is aimed at testing its correct operation under different scenarios. The suggested verification goals are grouped into two different categories. The first is associated with all the possible ways in which data can be sent from the input ports to the output ports. These goals do not require any coverage metric, and are assigned the highest priority since they ensure that data can be transferred correctly. The second group of verification goals is related to the correct operation of the request and ready outputs. They require assertions to verify specific properties, and are assigned a lower priority.

A SystemC behavioural model for the router was built in order to generate expected output data. However, the implemented model did not follow the router operation accurately, so it was decided not to use it in the verification environments. The implemented tests were then only based on checking that input data was exactly the same as the output data obtained in the selected output ports. These tests were based on the ones suggested in the verification plan. However, due to time limitations, only the first three tests were performed. These are described in the following subsections.

5.2.1 Router single data transfer test

It consists of a single data transfer from one input port of the router to one of the output ports. The goal of this test is to check that data sent to any input port (only one at a time) can be propagated to the desired output port. Figure 5.2 shows the test environment implemented, also included in **Appendix D**. The following subsections describe the operation of each relevant UVM component.

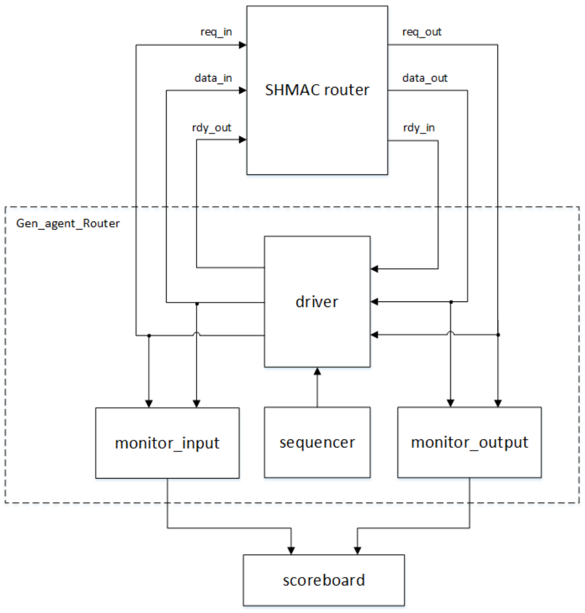

Figure 5.2: Verification environment built for the SHMAC router

Sequencer

The transactions employed by the sequencer include the following variables:

- rand int nr_inport
- rand logic [89:0] indata
- logic [89:0] outdata
- logic [4:0] out_dir

The variable *nr_inport* indicates which input port is used, and ranges from 0 to 4 (east, north, west, south, local port respectively). The following constraint is used to define this range:

- constraint nr_inport_c { nr_inport inside {0, 1, 2, 3, 4}; }

The variable *indata* represents the data to be sent from the input port specified by *nr_inport*. The destination field of *indata* indicates the destination output port. In order to have a reference, the current tile is assumed to have coordinates $x = 1$ and $y = 1$, so the destination field can be constrained to have x and y destination coordinates in the range from 0 to 2, as defined by the following constraint:

- constraint indata_c {
  indata[‘DEST_X_END:‘DEST_X_BEGIN] inside {4’d0, 4’d1, 4’d2};
  indata[‘DEST_Y_END:‘DEST_Y_BEGIN] inside {4’d0, 4’d1, 4’d2}; }

The variable *outdata* represents the output data obtained in the output port which asserts a request, while *out_dir* indicates which output port asserts this request.

A certain number of these transactions (100) are randomized, with the constraints previously defined, and grouped into a sequence, which is sent to the driver.

Driver

The run_phase of the driver is implemented as a sequence of states which get the data contained in each transaction and drive the router in order to perform a single data transfer. The following listing describes the high-level operation of the driver.

Input Monitor

The run_phase of the input monitor is also implemented as a succession of states which monitors the input signals *req_in* and *data_in*, and predicts the output data and which output

Listing 4: Pseudo-code of the run_phase of the driver of the Router single data transfer test

```
1 Reset the DUT.
2 Get a new transaction from the sequencer.
3 Assert the bit in req_in which is indicated by nr_inport, and overwrite the portion of
    data_in which corresponds to the input port selected with the value of indata.
4 Wait until the selected input port asserts the ready output (rdy_in = 1), and deassert req_in.
5 Wait until there is a request in any of the output ports (req_out = 1).
6 Finish the current transaction and go back to step 2 until the sequencer stops sending new
    transactions.
```

port will be used. Listing 5 describes the high-level operation of the input monitor.

Listing 5: Pseudo-code of the run_phase of the input monitor of the Router single data transfer test

```
1 Wait until (req_in = 1 & rdy_in = 1) in any of the input ports, and store the number of the
    stimulated input port in nr_inport. Also store the input data in indata.
2 Predict the number of the output port to be used by analyzing the destination field in
    data_in.
3 Send the transaction to the analysis port, which is connected to the scoreboard.
```

Output Monitor

The run_phase of the output monitor is also implemented as a succession of states which monitors the output signals *req_out* and *data_out*. Listing 6 describes the high-level operation of the output monitor.

Listing 6: Pseudo-code of the run_phase of the output monitor of the Router single data transfer test

```
1 Wait until there is a request in any of the output ports (req_out = 1), and store the number
    of the stimulated output port in out_dir. Also store the output data in outdata.
2 Send the transaction to the analysis port, which is connected to the scoreboard.
```

The agent, scoreboard, environment and test modules for this test have the same structure as the ones implemented for the SHA1 test.

5.2.2 Router sequential data transfer test

It consists of continuous data transfers from one input port of the router to one of the output ports, and exercises all the possible combinations of input to output ports transfers. The goal of this test is to check that any input port can send consecutive data packets to any output port. The test environment implemented has the same structure as the shown in Figure 5.2. Each relevant UVM component implemented is described in the following subsections.

Sequencer

The transactions employed by the sequencer include the following variables:

- rand int nr_tests
- rand logic [89:0] input_array [0:TESTS*5-1]
- logic [89:0] output_array [0:TESTS*5-1]

The variable *nr_tests* indicates the number of consecutive data transfers to be done from one input port to each output port. The variable *input_array* is an array which contains the data to be sent from one input port to the five output ports, in the order east, north, west, south and local. The destination port in each data packet is selected by employing constant values for the *x* and *y* coordinates in the destination field of the packet. As there are *nr_tests* transfers per output port, the length of the array is *nr_tests**5. Finally, the variable *output_array* is an array that contains the output data obtained in the output ports which generate a request, in the order east, north, west, south and local. Then, if the test is correct both *input_array* and *output_array* should be the same. The length of this array is also *nr_tests**5.

A certain number of these transactions (10) are randomized, with the constraints previously defined, and grouped into a sequence, which is sent to the driver.

Driver

The run_phase of the driver is implemented as a sequence of states which get the data contained in each transaction and drive the router in order to perform multiple sequential data transfers from each input port to each output port. The following listing describes the high-level operation of the driver.

Listing 7: Pseudo-code of the run_phase of the driver of the Router sequential data transfer test

```
Reset the DUT.
Get a new transaction from the sequencer.
for $i < 5$ do
    for $j < 5$ do
        for $k < nr\_tests$ do
            Generate a request (req_in = 1) in the ith input port.
            data_in = input_array[j*nr_tests + k]. Wait until the selected input port asserts
              the ready output (rdy_in = 1), and deassert req_in.
            Wait until there is a request in any of the output ports (req_out = 1).
        end
    end
    Select the (i + 1) input port.
end
Finish the current transaction and go back to step 2 until the sequencer stops sending new
  transactions.
```

Input Monitor

The run_phase of the input monitor is also implemented as a succession of states which monitors the input signals *req_in* and *data_in*, and obtains from them the values of the *input_array* variable of the current transaction. Listing 8 describes the high-level operation of the input monitor.

Listing 8: Pseudo-code of the run_phase of the input monitor of the Router sequential data transfer test

```
for $t = 0; t < 5; t++$ do
    while $req\_in[1] \neq 1$ do
        If (req_in[0] = 1 & rdy_in[0] = 1), store data_in[0] in the nth element of the
          input_array and increase n.
    end
    for $i = 1; i < 5; i++$ do
        for $j = 0; j < n; j++$ do
            Wait until (req_in = 1 & rdy_in = 1) in the input port i, and store data_in in the
              (i*n+j)th element of the input_array.
        end
    end
    Send the transaction to the analysis port, which is connected to the scoreboard.
end
```

Output Monitor

The run_phase of the output monitor is also implemented as a succession of states which monitors the output signals *req_out* and *data_out*. Listing 9 describes the high-level operation of the output monitor.

Listing 9: Pseudo-code of the run_phase of the output monitor of the Router sequential data transfer test

```
for $t = 0; t < 5; t++$ do
    while $req\_out[1] \neq 1$ do
        If (req_out[0] = 1), store data_out[0] in the nth element of the output_array and
          increase n.
    end
    for $i = 1; i < 5; i++$ do
        for $j = 0; j < n; j++$ do
            Wait until (req_out = 1) in the output port i, and store data_out in the (i*n+j)th
              element of the output_array.
        end
    end
    Send the transaction to the analysis port, which is connected to the scoreboard.
end
```

The agent, scoreboard, environment and test modules for this test have the same structure as the ones implemented for the SHA1 test.

5.2.3 Router multisource data transfer test

It consists of multiple data transfers at the same time, from the five input ports to different output ports. The goal of this test is to check that all the input ports can be used simultaneously and propagate data to the output ports. The test environment implemented has the same structure as the shown in Figure 5.2. Each relevant UVM component implemented is described in the following subsections.

Sequencer

The transactions employed by the sequencer include the following variables:

- rand logic [4:0] dir_tests
- rand logic [89:0] input_array [0:4]
- logic [89:0] output_array [0:4]

The variable *dir_tests* in an array which indicates the direction (number of the output port) of the data transfers from the five input ports in the order east, north, west, south and local. The variable *input_array* is an array which contains the data to be sent from the five input ports, also in the order east, north, west, south and local. The destination field in each data packet is set according to the values of the *dir_tests*. Finally, the variable *output_array* is an array that contains the output data obtained in the output ports, in the order east, north, west, south and local. Then, if the test is correct both *input_array* and *output_array* should have the same elements but in different order. A certain number of these transactions (100) are randomized, with the constraints previously defined, and grouped into a sequence, which is sent to the driver.

Driver

The run_phase of the driver is implemented as a sequence of states which get the data contained in each transaction and drive the router in order to perform data transfers from the five input ports at the same time. The following listing describes the high-level operation of the driver.

Listing 10: Pseudo-code of the run_phase of the driver of the Router multisource data transfer test

```
1 Reset the DUT.
2 Get a new transaction from the sequencer.
3 Group the elements of the input_array in a single word (from the element 0 to the element
   4) and overwrite data_in. Also assert the five input request signals (req_in = 1).
4 Wait until there is a request in the five output ports (req_out = 1).
5 Finish the current transaction and go back to step 2 until the sequencer stops sending new
   transactions.
```

Input Monitor

The run_phase of the input monitor is also implemented as a succession of states which monitors the input signals *req_in* and *data_in*, and predicts the output data and which destination output port is used by each input port. Listing 11 describes the high-level operation of the input monitor.

Output Monitor

The run_phase of the output monitor is also implemented as a succession of states which monitors the output signals *req_out* and *data_out*. Listing 12 describes the high-level operation of the output monitor.

Listing 11: Pseudo-code of the run_phase of the input monitor of the Router multisource data transfer test

```
1 Wait until (req_in = 1 & rdy_in = 1) in the five input ports, and store the input data of the
    five ports (data_in) in input_array, in the order east, north, west, south and local.
2 Calculate the order of the destination output ports by analyzing the destination field in
    each element of the input_array.
3 Send the transaction to the analysis port, which is connected to the scoreboard.
```

Listing 12: Pseudo-code of the run_phase of the output monitor of the Router multisource data transfer test

```
1 Wait until there is a request in the five output ports (req_out = 1), and store the output data
    of the five ports (data_out) in output_array, in the order east, north, west, south and
    local.
2 Send the transaction to the analysis port, which is connected to the scoreboard.
```

The agent, scoreboard, environment and test modules for this test have the same structure as the ones implemented for the SHA1 test.

5.3 SHA1 tile test

A SHA1 tile in the SHMAC mesh contains a SHA1 accelerator and a router, which allows the communication with CPU cores and memory located in other tiles. Figure 5.3 shows a circuit that replicates the structure of a SHA1 tile. As it can be seen, the local input and output ports of the router are connected to the SHA1 core, while the east input and output ports of the router are connected to the general input and output ports of the tile, respectively. Any input data packet is sent from the east input port of the router to the local output port, and then processed by the SHA1 core. Any output data produced by the core is sent from the local input port of the router to the east output port, so it is available in the output port of the tile.

The input data signals of the SHA1 core are grouped into a single signal named *data_in*, since the router employs a single 90-bit data signal in each port. The output data signals are also grouped into one signal named *data_out*. Table 5.1 shows the distribution of the bits in these two signals. It can also be seen in Figure 5.3 that the input *rdy_out* of the SHA1 core depends on both *rdyE_out* and *rdyL_in*. The reason for this is that the SHA1 should only be able to provide new data when the external module (memory) is ready (*rdyE_out*) and the corresponding port in the router is ready (*rdyL_in*).

The SHA1 tile test is mainly aimed at checking the correct integration between the SHA1 core and the SHMAC router. At this level, it is assumed that the router has a correct functionality and is free of bugs. Then, this test is based on checking that the SHA1 can

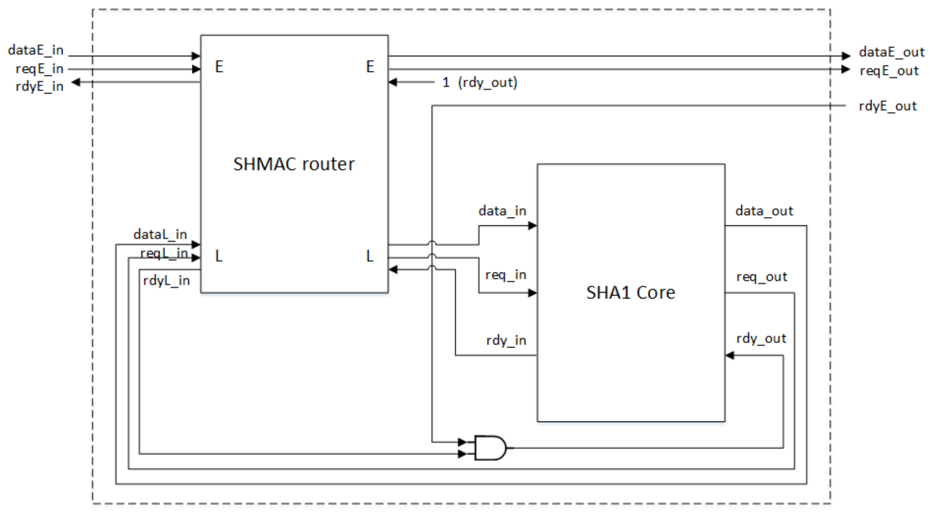

Figure 5.3: Circuit which replicates the structure of a SHA1 tile

still operate correctly while working with another SHMAC unit. The verification goals for this test are the same as those specified in the SHA1 verification plan (**Appendix B**).

5.3.1 SHA1 tile UVM-based verification environment

The verification environment built for the SHA1 tile test reuses the UVM agents implemented for testing the SHA1 core and the SHMAC router (single data transfer test). Figure 5.4 shows the structure of this verification environment, which contains three agents in total. Both router and SHA1 agents are configured in passive mode, and are only used for monitoring internal signals and checking the correctness of partial results. The SHA1 tile agent has the same operation as the SHA1 agent. The only difference is that the data sent to the tile has a specific destination address so that it is directed to the SHA1 core through the router.

The following subsections describe the UVM environment and test classes implemented for this test.

Environment

The build_phase of the environment instantiates the three agents and scoreboards. Both sha1_agent and router_agent are configured as passive agents, while the sha1tile_agent is configured as an active one. In addition, the connect_phase connects each analysis port of the agents with the respective analysis_export in the scoreboards.

Signal	Bits	Description
data_in	31-0	in_addr: input address
	32	in_reply: input reply signal
	33	in_write: input write signal
	37-34	in_mask: input mask signal
	38	in_exclusive: input exclusive signal
	39	not connected
	40	in_burst: input burst signal
	72-41	in_data: input data
	73	not connected
	77-74	send_x: x coordinate of the sender tile
	81-78	send_y: y coordinate of the sender tile
	85-82	dest_x: x coordinate of the destination tile
	89-86	dest_y: y coordinate of the destination tile
data_out	31-0	out_addr: output address
	32	out_reply: output reply signal
	33	out_write: output write signal
	37-34	out_mask: output mask signal
	38	out_exclusive: output interrupt signal (irq)
	39	not connected
	40	out_burst: output burst signal
	72-41	out_data: output data
	73	not connected
	77-74	send_x: x coordinate of the sender tile
	81-78	send_y: y coordinate of the sender tile
	85-82	dest_x: x coordinate of the destination tile
	89-86	dest_y: y coordinate of the destination tile

Table 5.1: Distribution of the bits of the SHA1 data signals

Test

The build_phase of the test instantiates the environment and the configuration classes of the three agents, which indicate whether the agent is active or passive. In addition, the run_phase starts the test by enabling the generation of a sequence of transactions in the sha1tile_agent.

It is important to mention that a bug was found in the SHA1 core while running this test. When the SHA1 asserts a *req_out* and no data is replied within the next three clock cycles, the core starts providing new data and the SHA1 computation gets corrupted. This was then confirmed by the SHMAC developers, and finally fixed.

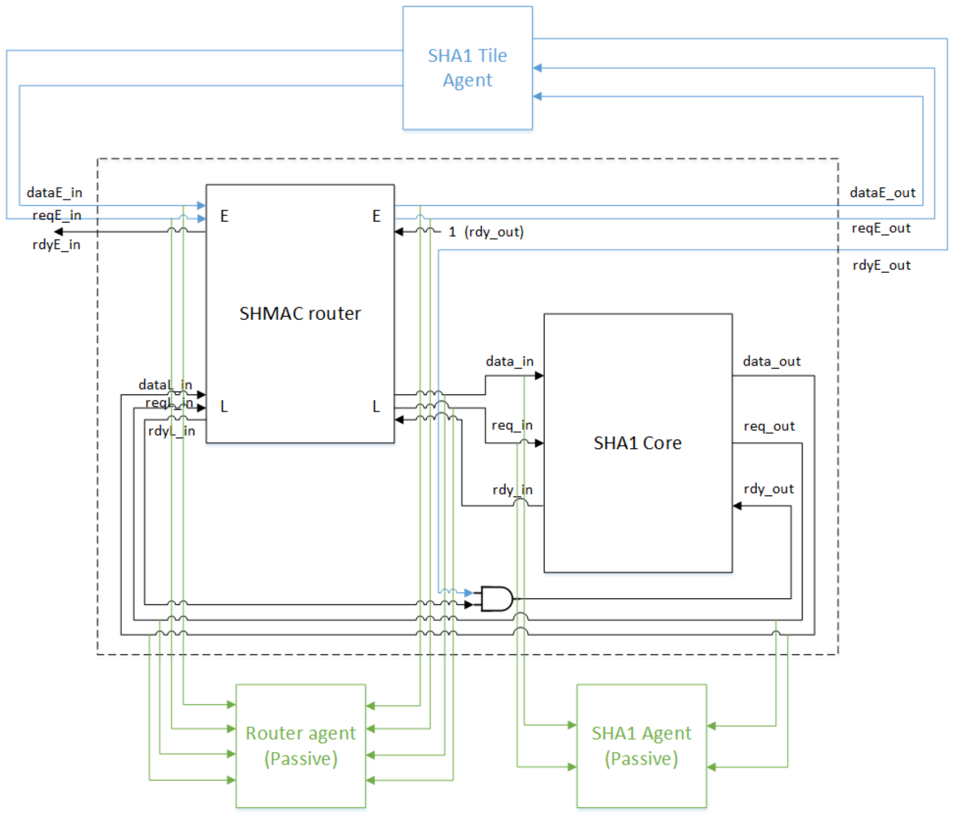

Figure 5.4: Verification environment built for the SHA1 tile

5.4 Top level test

The aim of this test is to check that the SHA1 core still operates correctly in a real environment, while working along with a RISC-V CPU core and the main memory. It instantiates a SHMAC mesh which contains four tiles, as shown in figure 5.5: an APB tile, a CPU tile, a SHA1 accelerator tile and a ZBT RAM tile. This SHMAC top level file (shmac_toplevel.vhd) was obtained by employing the *makefile* available in the master_branch of the SHMAC project. The APB tile allows communication with an external module (testbench in this test), and makes it possible to write on the main memory and initialize the SHMAC modules and interconnects. In addition, the ZBT RAM tile allows other tiles to interact with the main memory, which in this case is modeled in the testbench.

The testbench built for this test is only written in SystemVerilog. The reason for this is that the *makefile* of the SHMAC previously mentioned does not allow to monitor internal signals, so it is no possible to reuse the UVM agents previously implemented for the router, SHA1 core and SHA1 tile. Then, the test only runs one predefined SHA1 computation at

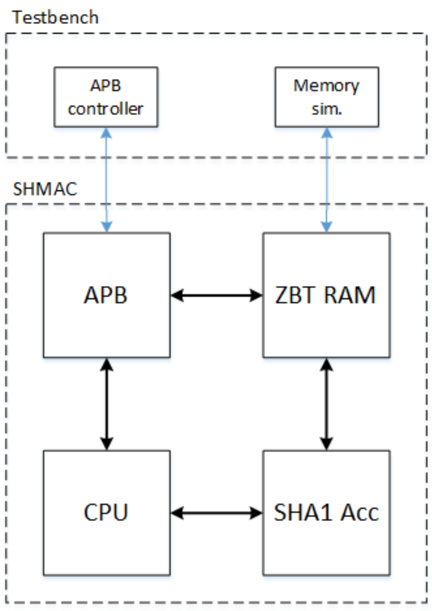

Figure 5.5: Testbench built for the top level test

a time.

Listing 13 describes the operation of the testbench built for this test. It includes a task which implements both APB write and read operations (APB controller), as well as a ZBT RAM simulator.

Listing 13: Pseudo-code of the top level testbench

```
Reset the SHMAC.
Run the interconnects within the SHMAC.
Write the program to be run by the CPU in RAM.
Read and verify the data in RAM.
Write the configuration data as well as the data to be hashed by the SHA1 in RAM.
Read and verify the data in RAM.
Initialize the CPU tile.
Run a delay routine (10000 clock cycles).
Read the hash values stored in RAM by the CPU core.
Compare the computed hash values with the expected ones.
```

Finally, listing 14 describes the operation of the C program run by the CPU core in order to perform a SHA1 computation.

Listing 14: Pseudo-code of the C program run by the CPU core in the top level test

```
Read the start address value stored in RAM and use it to overwrite the start address
  register of the SHA1 core.
Read the size value stored in RAM and use it to overwrite the size register of the SHA1
  core.
Overwrite the status register of the SHA1 core with any value (e.g. 1) to start the
  computation.
Run a delay routine (9000 clock cycles).
Read the status register and wait until the done bit is 1.
Read the hash values computed by the SHA1 core and store them in RAM.
```

Chapter 6

Discussion/Evaluation

The set of UVM-based tests described in the previous chapter made it possible to ensure the correctness of both the SHA1 accelerator core and the SHMAC router, and even could detect the presence of a bug in the operation of the SHA1 core. The methodology applied in these tests took into account the requirements specified in chapter 1 for the new verification strategy. Table 6.1 shows these requirements and indicates how the verification methodology implemented satisfies them.

Table 6.1: Strategies implemented for meeting the requirements for the project listed in table 1.1

Requirement ID	Description	Strategy implemented
REQ-Q1	The verification strategy must clearly define which metrics and level of those metrics that is needed in order to decide when verification is complete. If different environments require different metrics to be collected, that must be clearly stated.	Both functional and code coverage were employed in the tests. The first was implemented when using cover groups for certain variables in the UVM transactions, while code coverage reports were generated automatically by Questasim (which mainly include line and branch coverage).
REQ-Q2	The verification strategy must define the different types of environments that should be implemented and high level features of those environments such as stimuli generation strategy and checking strategy.	The verification plans implemented include suggested UVM-based tests and verification environments.
REQ-Q3	The verification strategy must enable the user to do as efficient debugging as possible.	The use of passive agents in the tests allows to monitor internal signals and detect the source of possible bugs. In addition, assertions made it possible to check specific properties of the designs and easily detect the source of errors found in observable nodes.
REQ-Q4	The verification strategy must enable verification environments that are robust against future changes to the design and that can easily be reused.	The UVM verification environments were coded in such a way that each UVM class can be easily reused in another verification environment.
REQ-Q5	It must be easy to determine if the design has a HW bug.	The top level test of the SHA1 core assumes that the SHA1 hardware operates correctly, and any error detected at this level is most likely to be a software bug in the CPU core.
REQ-Q6	Assuming a certain quality level has been achieved, the verification strategy must describe how the same level can be maintained over time even if modifications to the RTL is ongoing.	Any modification to the SHMAC architecture should be documented so that appropriate test cases can be generated to exercise the new functionality.
REQ-Q7	The verification strategy must give guidance on how the team should prioritize in order to get from the current state to the new and improved way of doing verification.	The verification methodology proposed previously includes a sequence of steps that can be followed when testing a module of the SHMAC platform.
REQ-Q8	The full system contains both hardware and software. The verification strategy should also describe how hardware and software can be verified together.	The top level test of the SHA1 core checks the integration between a CPU core and the SHA1 accelerator by running a program which consists of a SHA1 computation. Both software and hardware are exercised and checked in this test.
REQ-Q9	The verification strategy should also describe typical design for verification guidelines that designers should follow when writing RTL.	Appendix E contains a set of useful design for verification guidelines.
REQ-Q10	The verification strategy must list required documentation needed before a verification engineer can efficiently start on investigating how to e.g. verify a block. A outline of the content of the necessary documents must be specified.	Appendix A and appendix C are examples of circuit description documents, which essentially describe the structure and functionality of a module.
REQ-Q11	The verification strategy must describe what document that must be produced when architecting a new verification environment and the content of those documents.	Appendix B and appendix D are examples of verification plan documents, which essentially describe the features to be tested in the corresponding modules and suggest UVM-based tests.
REQ-Q12	The verification strategy must describe how bugs will be managed and tracked.	This is left as future work.

The methodology employed in the verification of the SHA1 accelerator core can be replicated in the verification of any main component within the SHMAC architecture. This bottom-up approach allows to check that a certain module can operate correctly first while working independently (in isolation from the rest of the SHMAC modules) and then while interacting with other modules such as routers, processors, memories and others. For instance, this test methodology can be employed in the verification of a cache coherence system, which is a future work in the SHMAC project. The first tests can be focused on checking proper behaviour of the module while working with a local cache, and further tests can include the integration of the module with the main memory and a CPU core.

In addition, in order to maintain the same quality of verification over all the SHMAC platform, it is important to follow the same sequence of steps when designing and testing any module of the SHMAC. As described in chapter 4, this process starts by elaborating a circuit description document in which the developer must explain relevant information about the structure and functionality of the module. Then, while coding the hardware description of the module, it is essential to follow some good design for verification practices. Next, a verification plan document should be elaborated based on the circuit description document and possible test cases detected in the design stage. Finally, only after having performed the previous steps, the UVM-based testbench of the module can be implemented. In this stage, it is also important to follow good UVM coding practices in order to produce consistent, readable and reusable code.

The implementation of this verification methodology in the SHMAC project will allow to bring the platform up to industry quality standards. Even though the UVM has a longer learning phase compared to the methodology of directed tests, it provides a faster and more thorough verification process over time. This is due to the fact that it makes it possible to efficiently reuse verification components already implemented, and also because it can run a large amount of random tests in a same verification environment, which increases the probability of finding bugs in the design. The developers would then have a higher certainty that their designs have a correct functionality and are free of errors.

Chapter 7

Conclusion and future work

7.1 Conclusion

The verification framework implemented for the SHMAC platform, mainly based on the complete verification of the SHA1 accelerator core, was aimed at providing a set of tests that can check the correctness of SHMAC units in a more efficient and thorough way than the current verification strategies do. The verification methodology proposed in chapter 4 also provides guidelines and good practices that can be followed to increase the quality of verification.

This new verification methodology overcomes many of the limitations of the existing verification strategies for the SHMAC platform. One of these main deficiencies involves the use of directed tests in both testbenches and bare-metal testing on FPGA. Hand-coding all the possible test cases not only represents a time-consuming and tedious task for the developers, but also usually leads to verification incompleteness. The proposed methodology defines first verification goals (in a verification plan) and then runs several random tests aimed at checking that the design can achieve each of these goals. Coverage metrics are employed to measure how thoroughly the design features are tested.

Constrained random stimulus, which were employed in all the UVM-based tests in this work, make it possible to exercise most of the scenarios in which a design can work, and then increase the probability of finding bugs. In fact, they helped to detect a bug in the operation of the SHA1 accelerator core, as explained in chapter 5. This error is unlikely to be detected by a methodology of directed tests, since it involves a not very usual test case, where a random delay is used in one of the input ports of the core.

The verification strategy proposed in chapter 4 provides a formal and efficient approach to verify any module in the SHMAC platform. Each stage in this suggested verification method has an impact in the overall quality of verification of the design. Design aimed at verification is essential as it enables to identify all the possible relevant test cases. Then, the verification plan can be designed with less effort. Also, an efficient verification plan makes it easier to implement a UVM-based testbench, as it includes goals and test structures that it must follow.

In addition, the verification framework was implemented with a strong focus on reusability, as explained in chapter 5. This means that any verification component developed in this work can be easily reused in a different verification environment, and also can be readily extended or modified to verify new functionality of the DUTs. This same reusability principle must be employed in other test environments in the SHMAC in order to make the verification process faster and with less effort from the developer.

A verification framework for the complete architecture of the SHMAC can be built by taking this work as starting point. The bottom-up verification approach and verification strategy used for the SHA1 core can be replicated when testing any other module of the SHMAC. The following section contains future work regarding the tasks that can be implemented to build a complete verification framework for the SHMAC.

7.2 Future work

This section describes the tasks that can be performed to improve the verification quality of the SHA1 core, as well as those that can be implemented to build a framework for the verification of the complete SHMAC platform.

7.2.1 Attach the SHA1 SystemC reference model to the SHA1 verification environment

The SHA1 SystemC model now stands as an independent module which can only communicate with its own separate SystemVerilog module, but not with the SHA1 verification environment. Some difficulties were faced when implementing the connection between these modules, which need to be fixed in order to run SHA1 tests faster. Also, the model can be improved to be cycle-accurate in order to model the SHA1 core more precisely.

7.2.2 Create a SystemC router model

An attempt was made at making a router model, but it ended up being a collection of incomplete parts. A router software model should be created in SystemC and be fitted with TLM2.0 connection, so it can be used as a reference model in the verification suite.

7.2.3 Rewrite the top level SHMAC makefile

The top level SHMAC makefile should be rewritten so that internal signals can be propagated to outputs. This would make it possible to reuse agents implemented in unit level tests, and check partial results of the current computation so that the source of a certain bug can be more easily detected. Also, this would enhance the bottom-up verification approach proposed in this work.

7.2.4 Build verification environments and reference models for the rest of the SHMAC components

New verification environments should be implemented for the rest of the SHMAC units using the strategies and guidelines proposed in this work. They should focus on reusability, so less effort and time is spent in future tests. SystemC reference models of the rest of the SHMAC units should also be built in order to generate predicted outputs of the modules, and also to enable software developers to implement firmware at an earlier stage.

Bibliography

[1] Accellera. About us. http://accellera.org/about.

[2] Freescale Amit Hermony. Design for verification. http://www.aceverification.com/LeadershipSeminar/DESIGN%20FOR%20VERIFICATION\discretionary{-}{}{}Amit\discretionary{-}{}{}Freescale.ppt.

[3] Fotis Andritsopoulos, C Charopoulos, Gregory Doumenis, Fotis Karoubalis, Yannis Mitsos, F Petreas, Ioanna Theologitou, Stylianos Perissakis, and D Reisis. Verification of a complex soc; the pro 3 case-study. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 224–229. IEEE, 2003.

[4] John Aynsley. Osci tlm-2.0 language reference manual. *Open SystemC Initiative*, 24, 2009.

[5] John Aynsley. Easier uvm for functional verification by mainstream users. In *Design and Verification Conference (DVCon)*, 2011.

[6] Mike G Bartley, Darren Galpin, and Tim Blackmore. A comparison of three verification techniques: directed testing, pseudo-random testing and property checking. In *Proceedings of the 39th annual Design Automation Conference*, pages 819–823. ACM, 2002.

[7] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. The temporal logic of branching time. *Acta informatica*, 20(3):207–226, 1983.

[8] David C Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the ground up*, volume 71. Springer Science & Business Media, 2009.

[9] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[10] Dominique D Borrione, Laurence V Pierre, and Ashraf M Salem. Formal verification of vhdl descriptions in the prevail environment. *Design & Test of Computers, IEEE*, 9(2):42–56, 1992.

[11] David Brier, Ramakrishnan Venkatasubramanian, Sampath Rangarajan, Abhishek Arun, Daniel Thompson, and Neelima Muralidharan. Verification methodology of heterogeneous dsp+ arm multicore processors for multi-core system on chip. In *Microprocessor Test and Verification (MTV), 2013 14th International Workshop on*, pages 112–117. IEEE, 2013.

[12] Jonathan Bromley. If SystemVerilog is so good, why do we need the UVM? Sharing responsibilities between libraries and the core language. In *Specification & Design Languages (FDL), 2013 Forum on*, pages 1–7. IEEE, 2013.

[13] Intilop Corporation. SOC Design Verification, White Paper http://www.intilop.com/resources/white_papers/SOC_Design_Verif_White_Paper.pdf.

[14] Markus Damm, Javier Moreno, Jan Haase, and Christoph Grimm. Using transaction level modeling techniques for wireless sensor network simulation. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1047–1052. European Design and Automation Association, 2010.

[15] Robert H. Dennard, Fritz H. Gaensslen, Hwa nien Yu, V. Leo Rideout, Andre Ernest Bassous, and R. Leblanc. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE J. Solid-State Circuits*, page 256, 1974.

[16] Doulos. Getting started with tlm-2.0. `https://www.doulos.com/knowhow/systemc/tlm2/tutorial__1/`.

[17] Doulos. Summary of the easier uvm coding guidelines. `https://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_guidelines/summary/`.

[18] Doulos. Detailed explanation of the easier uvm coding guidelines. `https://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_guidelines/detail/#link17`, apr 2016.

[19] Rolf Drechsler, Christophe Chevallaz, Franco Fummi, Alan J Hu, Ronny Morad, Frank Schirrmeister, and Alex Goryachev. Panel: Future SoC verification methodology: UVM evolution or revolution? In *DATE*, pages 1–5, 2014.

[20] Eivind Gamst Edward Mitacc. Shmac verification framework. Technical report, NTNU, 2015.

[21] EECS. The Single-ISA Heterogeneous MAny-core Computer (SHMAC). http://www. ntnu.edu/ime/eecs/shmac.

[22] EECS. Wiki, Technical Documentation, SHMAC HW, System Overview http://redmine.idi.ntnu.no/projects/shmac/wiki/System_Overview.

[23] Adam Erickson. Introducing uvm connect. *Mentor Graphics Verif. Horiz*, 8(1):6–12, 2012.

[24] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[25] T Fitzpatrick. Realizing advanced functional verification with questa. *Mentor Graphics Corporation (white paper)*, 2005.

[26] Dipti Girdhar and Neeraj Kr Shukla. Design and verification of amba apb protocol. *International Journal of Computer Applications*, 95(21), 2014.

[27] J. L. Hennesy and D. A. Patterson. *Computer Architecture, A Quantative Approach*. Morgan Kaufmann Publishers Inc., 2012.

[28] Alan Hunter, Andrew Piziali, Avi Ziv, Kelly Larson, and Shankar Hemmady. Ensuring Functional Closure of a Multi-core SoC through Verification Planning, Implementation and Execution. In *Microprocessor Test and Verification, 2008. MTV'08. Ninth International Workshop on*, pages 7–13. IEEE, 2008.

[29] Sasan Iman and Sunita Joshin. *The e Hardware Verification Language*. Springer US, 2004.

[30] Kelin J Kuhn. Moores Law Past 32nm: Future Challenges in Device Scaling. In *13th International Workshop on Computational Electronics, IWCE09)*, pages 1–6, 2009.

[31] Yves Mathys and André Châtelain. Verification strategy for integration 3g baseband soc. In *Proceedings of the 40th annual Design Automation Conference*, pages 7–10. ACM, 2003.

[32] Ashok B Mehta. *SystemVerilog Assertions and Functional Coverage*. Springer, 2013.

[33] Corp. Mentor Graphics. Tlm review. `https://verificationacademy.com/verification-methodology-reference/uvmc-2.3.0/docs/html/files/docs/TLM_REVIEW-txt.html#Legal_TLM_Connections`, 2003-2015.

[34] Marcelo Montoreano. Transaction level modeling using osci tlm 2.0. *Open SystemC Initiative (OSCI)*, 2007.

[35] Gordon E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics. *Solid-State Circuits Society Newsletter*, 11(5):33–35, 2006.

[36] Guy Mosensoson. Practical approaches to soc verification. In *Proceedings of DATE User Forum*, pages 05–08. Citeseer, 2002.

[37] National Institute of Standards and U.S. Department of Commerce Technology. Secure Hash Standard
http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.

[38] Mads Bergan Roligheten. Uvm verification framework. Master's thesis, NTNU, 2014.

[39] Chris Spear. *SystemVerilog for verification: a guide to learning the testbench language features*. Springer Science & Business Media, 2008.

[40] Nacsa Sndor. ARM's big.LITTLE technology
https://lazure2.files.wordpress.com/2013/04/image21.png.

[41] Rui Wang, Wenfa Zhan, Guisheng Jiang, Minglun Gao, and Su Zhang. Reuse issues in soc verification platform. In *Computer Supported Cooperative Work in Design, 2004. Proceedings. The 8th International Conference on*, volume 2, pages 685–688. IEEE, 2004.

[42] Young-Nam Yun, Jae-Beom Kim, Nam-Do Kim, and Byeong Min. Beyond UVM for practical SoC verification. In *SoC Design Conference (ISOCC), 2011 International*, pages 158–162. IEEE, 2011.

[43] Hu Zhaohui, Arnaud Pierres, Hu Shiqing, Chen Fang, Philippe Royannez, Eng Pek See, and Yean Ling Hoon. Practical and efficient SOC verification flow by reusing IP testcase and testbench. In *SoC Design Conference (ISOCC), 2012 International*, pages 175–178. IEEE, 2012.

Appendix

Appendix A: SHA1 core circuit description

SHA1 CORE CIRCUIT DESCRIPTION

1. Overview

The SHA1 core follows closely the SHA1 algorithm as specified in the SHA standard FIPS-PUB 180-4 (http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf). It is used to hash a message, M, having a length of l bits, where $0 < l < 2^{64}$. The algorithm uses 1) a message schedule of sixteen 32-bit words (w0-w15), 2) five working variables of 32 bits each (a-e), and 3) a hash value of five 32-bit words (h0-h4). The final result of SHA-1 is a 160-bit message digest.

2. External Interface

The SHA1 core interacts with external modules using the standard SHMAC router interface. The following figure shows the input and output ports of the core:

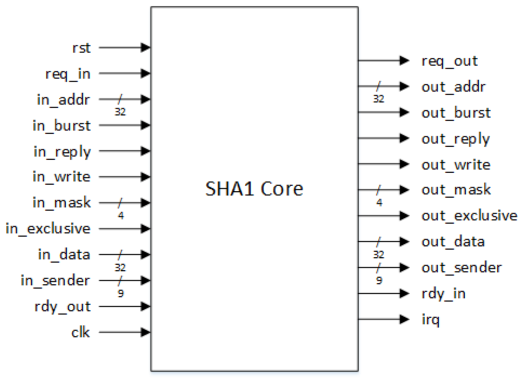

Figure 1. External interface of SHA1 Core

The operation of each port is described in the following table:

Port name	Type	Length (bits)	Description
req_in	input	1	indicates that new data is introduced
in_addr	input	32	memory address of write destination
in_burst	input	1	not used, default value = 0
in_reply	input	1	indicates a reply after data is requested
in_write	input	1	indicates a write operation
in_mask	input	4	not used, default value = 0
in_exclusive	input	1	not used, default value = 0
in_data	input	32	input data
in_sender	input	9	not used, default value = 0
rdy_out	input	1	indicates that external memory is ready to provide new data
req_out	output	1	indicates that new data is requested

out_addr	output	32	memory address of required data
out_burst	output	1	not used
out_reply	output	1	indicates a reply to data previously requested
out_write	output	1	not used, default value = 0
out_mask	output	4	default value = 15
out_exclusive	output	1	not used
out_data	output	32	output data
out_sender	output	9	default value = 0
rdy_in	output	1	indicates that the core is ready to receive new data
irq	output	1	indicates that hash is done
clk	input	1	clock signal
rst	input	1	reset signal

Table 1. Input/output ports of SHA1 core

3. Operation

The SHA1 core includes internal registers to store the necessary data. These are the following:

Register name	Offset	Length (bits)	Direction	Description
addr	0	32	W	specifies the start address of the data block to create the hash from
size	4	32	W	size in bytes of the data block to create the hash from
status	8	4	R/W	For writes: starts the hash computation engine (any value). For reads: reads the current status. The following status bits are defined: - Bit 0: done (hash is computed and interrupt line is asserted) - Bit 1: hashing (currently running the hashing algorithm) - Bit 2: prefetching (still doing prefetching of data) - Bit 3: error (requested hash operation cannot be performed)
H0	12	32	R	hash value, word 0
H1	16	32	R	hash value, word 1
H2	20	32	R	hash value, word 2
H3	24	32	R	hash value, word 3
H4	28	32	R	hash value, word 4

Table 2. Internal registers of SHA1 core

In addition, sixteen 32-bit registers (w0-w15) are used to store the message schedule, and five 32-bit registers (a-e) are used to store the working variables. They are employed exactly as in the standard document.

A state machine is employed to update all the registers and control the transfer of data between them. It is represented as an oval on figure 2, which represents the architecture of the SHA1 engine.

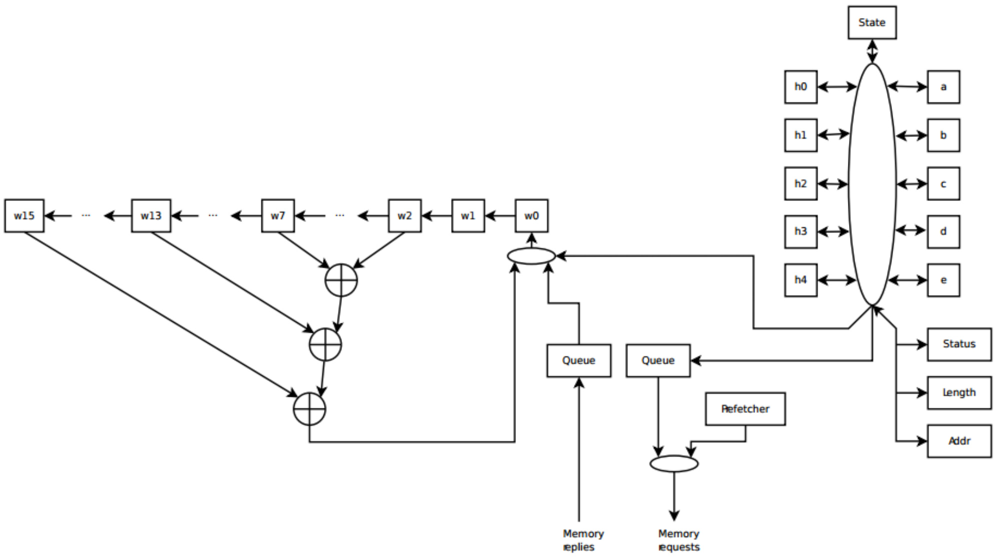

Figure 2. Architecture of the SHA1 engine

Programmers' reference

The SHA1 engine can be used to perform a hash computation in the following way:

1. Write address of data to *addr*

   The *addr* register represents the initial address of the block of data to create the hash from. In order to overwrite this register, the following signals have to be employed:

```
req_in <= 1;
in_addr <= 0;
in_write <= 1;
in_data <= (target address);
```

2. Write number of bytes to *size*

   The *size* register represents size in bytes of the data block to create the hash from. In order to overwrite this register, the following signals have to be employed:

```
req_in <= 1;
in_addr <= 4;
```

```
in_write <= 1;
in_data <= (size of data block);
```

3. Start hash engine by writing any value to the *status* register.

   In order to do so, the following signals have to be employed:

   ```
   req_in <= 1;
   in_addr <= 8;
   in_write <= 1;
   in_data <= 1 (or any value);
   ```

4. Wait for the hash computation to be done.

   Every time the core requests data from memory, the following output signals are asserted:

   ```
   req_out = 1;
   out_addr = (required memory address);
   ```

   In order to reply with the next data element from memory, the following input signals are employed:

   ```
   in_data <= data[i];
   req_in <= 1;
   in_reply <= 1;
   ```

   and in the next clock cycle the *rdy_out* signal must be asserted in order to indicate that the external memory is ready to provide new data

   ```
   rdy_out <= 1;
   ```

   Once all memory requests have been done and the hash has been computed, the *done* bit in the *status* register is asserted. This register can be read in the following way:

   ```
   req_in <= 1;
   in_addr <= 8;
   in_write <= 0;
   ```

   Once the output `req_out` is asserted, the *status* register can be read from `out_data`.

   Alternatively, the user can wait for the *irq* output to be asserted, and then clear it by reading the *status* register.

5. Read hash from H0, H1, H2, H3 and H4.

   Each hash value can be read using the following signals:

   ```
   req_in <= 1;
   in_addr <= (address of hash register);
   in_write <= 0;
   ```

   Once the output `req_out` is asserted, the hash value can be read from `out_data`.

Appendix B: SHA1 verification plan

SHA1 CORE VERIFICATION PLAN

1. Overview

This document lists all the design features to be tested on the SHA1 core, and provides a suggested UVM-based verification environment for this module.

2. Test Plan

Input data should be organized as a block of 512 bits (16x32) which is generated randomly. Each test should employ a different set of random data. *Addr* (start address of the data block to create the hash from) and *size* (size in bytes of the data block to create the hash from) inputs are also generated randomly so that a random portion of the 512-bits block is selected for the test.

The following table contains the features to be exercised for verifying the SHA1 core. It includes the type of functional coverage required for each test, as well as suggested priorities for the order of execution of the tests.

Section	Description	Coverage Type	Priority
Input data			
Data block to create the hash from	The module can operate correctly for any set of random data	Test result	1
Start address and size of input data block	The module can operate correctly for any start address and size values lower than 56 bytes (448 bits)	Cover group	1
	The module can operate correctly for size values between 56 and 64 bytes (448 and 512 bits)	Cover group	1
	A size value of zero asserts the error flag in the status register	Design assertion	2
Status register			
Status bits are asserted when certain conditions occur	Done bit is asserted when hash is computed	Design assertion	2
	Hashing bit is asserted when the hashing algorithm is running	Design assertion	2
	Prefetching bit is asserted when prefetching input data	Design assertion	2
Others			
Rdy_out input	The rdy_out input is never set (which indicates that the external memory is not ready to provide data) and no hash is computed	Design assertion	2

Req_in input	The req_in input can be asserted *n* clock cycles after the req_out output is asserted	Design assertion	2
Irq output	The irq output is 0 on reset.	Design assertion	2
	The irq output changes from 1 to 0 after reading the result	Design assertion	2

Table 1. Test plan for the SHA1 core

3. Verification environment

The following verification environment, shown on figure 1, is aimed at ensuring the correctness of the features with the highest priority (priority 1) specified on the test plan previously presented. It generates random blocks of data, as well as random start addresses and sizes, and drives them to the SHA1 core using the standard SHMAC router interface. The hash values produced by the core are then finally compared with the expected ones.

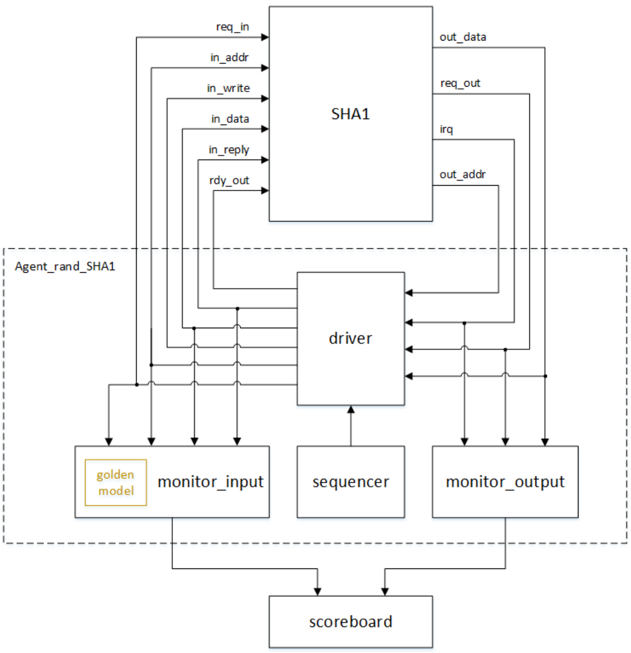

Figure 1. Verification environment for the tests with priority 1 on the test plan

The sequencer generates, for each test, a random data block of 512 bits (64x8), as well as a random start address and size. Then, the driver uses the standard SHMAC router protocol to configure and control the SHA1 core. First, it overwrites the internal registers (*start address* and *size*) and starts the hash computation. It then feeds the core with the data requested (specified on *out_address*) until all the memory requests are done. Once the hash is computed (indicated by the *irq* output), it reads the final five hash values produced by the core.

The monitor_input block gets the input data block generated by the driver, as well as the *start_address* and *size*, and employs a golden model of the SHA1 engine to produce the expected final hash values. The monitor_output block captures the hash values computed by the SHA1 core, and finally both expected and actual hash values are compared on the scoreboard.

Appendix C: Router circuit description

SHMAC ROUTER CIRCUIT DESCRIPTION

1. Overview

The SHMAC implements a 2D mesh-based Network-on-Chip (NoC) interconnect in order to provide efficient on-chip communication. Parameters of the interconnect are as follows:

Parameter	Value
Topology	2D mesh
Routing	Dimensions-ordered (XY)
Switching	Store-and-forward
Flow control	On/Off
Router delay	1 cycle

The routers in the network are implemented with a traditional multi-stage architecture. As shown in figure 1, all routers contain five ports, for the East, North, West, South and Local connections, respectively. The router implements two stages: 1) route computation and 2) switch allocation and traversal. Route computation is done using the XY-routing algorithm. The switch allocation stage performs arbitration for competing inputs (using Round-Robin scheme), and sets the crossbar according to the result of arbitration. A packet which has been granted the access traverses the switch during the same stage.

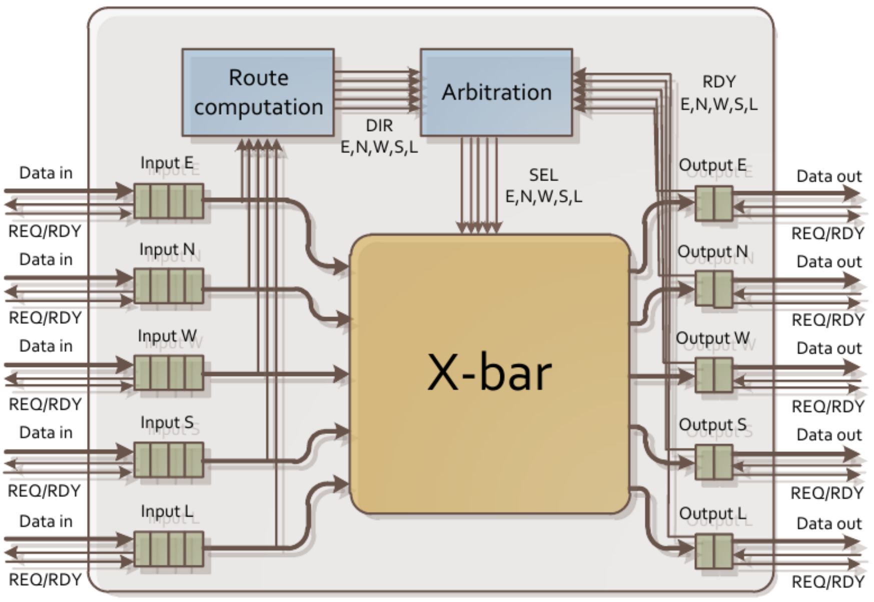

Figure 1. Internal architecture of the SHMAC router

Router FIFOs implement a write-through model, meaning that the packet at FIFO head becomes available for reading immediately after it has been written to the FIFO. This allows a packet written to an output FIFO propagate to the input of the downstream router within the same cycle, which is referred to as a link traversal cycle.

In general, it takes only one cycle for a packet to travel between the two adjacent routers. Packet traversal can be summarized as follows:

- Cycle 1: The packet is written into the input FIFO.

- Cycle 2: The packet goes through the crossbar and, if the output link is ready, is fed directly to the output link and into the input FIFO in the next router. If the output link is busy, the packet is inserted into the output FIFO and will wait there until link is ready.

The interconnect link includes a set of parallel wires to transfer the packet data and two control signals. The valid and ready control signals represent whether the data is available at the link source and whether the link destination is ready to receive the data, respectively.

2. External Interface

The following figure shows the input and output ports of the SHMAC router:

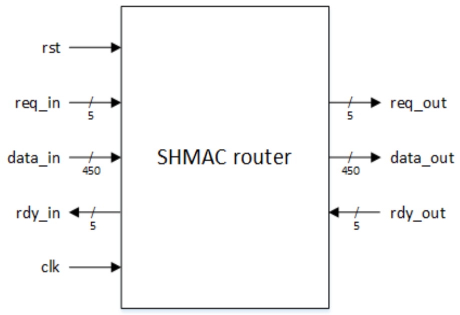

Figure 2. External interface of the SHMAC router

The operation of each port is described in the following table:

Port name	Type	Length (bits)	Description
req_in	input	5	indicates requests sent to the input ports: local, south, west, north and east (1 bit each)
data_in	input	450	input data sent to the input ports: local, south, west, north and east (90 bits each)
rdy_in	output	5	indicates that the input FIFOs are not full: local, south, west, north and east (1 bit each)

req_out	output	5	indicates that data is available in the output ports: local, south, west, north and east (1 bit each)
data_out	output	450	output data in the output ports: local, south, west, north and east (90 bits each)
rdy_out	input	5	indicates that an external module is ready to receive data from the input ports: local, south, west, north and east (1 bit each)
clk	input	1	clock signal
rst	input	1	reset signal

Table 1. Input/output ports of the SHMAC router

3. Operation

The SHMAC router is composed of the following internal modules:

- Input ports (5): each port contains a fixed-size FIFO which stores the incoming data when a request is done (req_in = 1) and the FIFO is not full (rdy_in = 1).
- Route computation module (5): based on the coordinates of the destination tile in the SHMAC (indicated in the input data packet) and the coordinates of the current tile, it determines the output port where the input data will be sent.
- Arbiter (5): performs arbitration for the competing input ports using a round-robin scheme (priority is updated every clock cycle), where only one gets granted at a time. It only grants if the input port is not empty and the destination output port is not full. The results of the arbitrations (in the 5 arbiters) are used to set the crossbar.
- Crossbar (1): based on the grant signals from the arbiter, it directs the packets in the input ports to the desired output port.
- Output ports (5): each port contains a fixed-size FIFO which stores the output data. It asserts a request signal (req_out = 1) when the FIFO is not empty, and data is read from the FIFO when rdy_out = 1.

Appendix D: Router verification plan

SHMAC ROUTER VERIFICATION PLAN

1. Overview

This document lists all the design features to be tested on the SHMAC router, and provides suggested UVM-based tests for this module.

2. Test Plan

The following table contains the features to be exercised for verifying the SHMAC router. It includes the type of functional coverage required for each test, as well as suggested priorities for the order of execution of the tests.

Section	Description	Coverage Type	Priority
Data routing			
Single data transfer	The router can direct data from one input port to one output port (one data transfer at a time)	Test result	1
Sequential data transfer	The router can direct continuously data from one input port to one output port	Test result	1
Multisource data transfer	The router can direct data from multiple input ports to certain output ports (different destination ports)	Test result	1
	The router can direct data from multiple input ports to the same output port (in multiple clock cycles)	Test result	1
Others			
Rdy_in output	The rdy_in output of any input port is asserted when the FIFO is full	Design assertion	2
	If the rdy_in output of any input port is not active, input data is not directed to the desired output port	Design assertion	2
Req_out output	The req_out output of any output port remains active if the rdy_out input is not asserted	Design assertion	2
Rdy_out input	If the rdy_out input of any output port is asserted after n clock cycles, the data in the head of the FIFO remains the same	Design assertion	2

Table 1. Test plan for the SHMAC router

3. UVM-based tests

The following tests are aimed at ensuring the correctness of the features with the highest priority (priority 1) specified on the test plan previously presented.

A. Single data transfer test

It consists of a single data transfer from one input port of the router to one of the output ports. The transactions in this test can include the following variables:

Variable	I/O	Description
Data_in	I	Data to be sent from one of the input ports, where the destination field in the data packet is predefined
Nr_inport	I	Indicates which input port is used
Data_out	O	Output data obtained in the output port which asserts a req_out
Nr_outport	O	Indicates which output port asserts a req_out

A successful test occurs when both *data_in* and *data_out* match, and the value of *nr_outport* matches with the output port associated to the destination field in *data_in*.

B. Sequential data transfer test

It consists of continuous data transfers from one input port of the router to one of the output ports. It should exercise all the combinations of routing between the five input and output ports. The transactions in this test can include the following variables:

Variable	I/O	Description
Nr_tests	I	Indicates the number of consecutive data transfers to be performed from one input port to each output port.
Array_in	I	Array which contains the data to be sent from one input port to the five output ports (in the order east, north, west, south, local). The length of the array is Nr_tests*5.
Array_out	O	Array which contains the output data obtained in the output ports which assert a req_out (in the order order east, north, west, south, local). The length of the array is Nr_tests*5.

The same transaction can be used for testing each input port, so a full test will employ five of these transactions. A successful test occurs when both *array_in* and *array_out* match.

C. Multisource data transfer test (different output ports)

It consists of multiple data transfers at the same time, from the five input ports to different output ports. The transactions in this test can include the following variables:

Variable	I/O	Description
Dir_tests	I	Array which indicates the directions of the data transfers from the five input ports (in the order east, north, west, south, local). The length of the array is 5.
Array_in	I	Array which contains the data to be sent from the five input ports (in the order east, north, west, south, local). The length of the array is 5.
Array_out	O	Array which contains the output data obtained in the output ports (in the order east, north, west, south, local). The length of the array is 5.

In order to check the correctness of the test, the *array_in* is resorted based on the directions specified in *dir_tests*, and compared with the *array_out*.

D. Multisource data transfer test (same output port)

It consists of multiple data transfers at the same time, from the five input ports to the same output port. The transactions in this test can include the following variables:

Variable	I/O	Description
Array_in	I	Array which contains the data to be sent from the five input ports (in the order east, north, west, south, local). The length of the array is 5.
Array_out	O	Array which contains the output data obtained in the output port which asserts a req_out for five consecutive times. The length of the array is 5.

Since the data in the output port can arrive in any order, the scoreboard in the test must not consider the order of the elements in both *array_in* and *array_out*, but must check that the elements are the same when comparing both arrays.

Figure 1 shows a generic verification environment which can be used for performing these four tests. The sequencer in each test generates the transactions suggested and sends them to the driver, which controls the execution of the router (DUT). The input monitor generates the expected outputs (output variables in the transactions) from the data sent to the DUT, while the output monitor obtains the actual outputs (output variables in the transactions) from the data produced by the DUT. These two outputs are finally compared in the scoreboard.

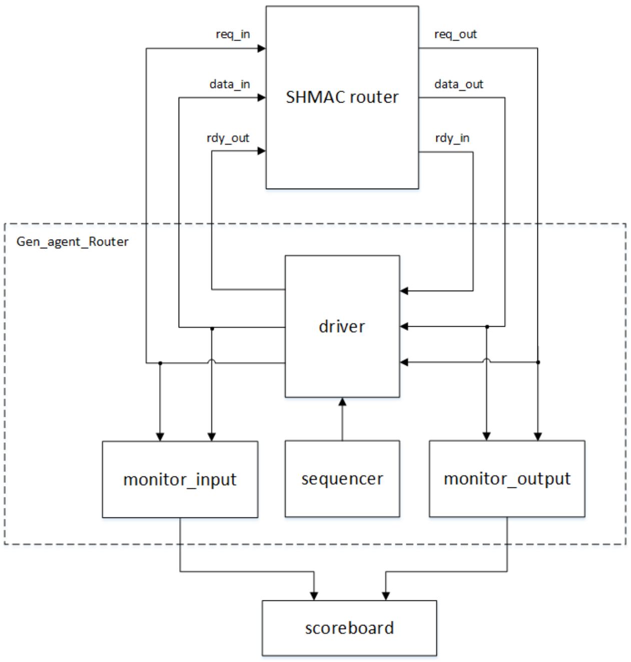

Figure 1. Generic verification environment for the tests with priority 1 on the test plan

Appendix E: Design for verification guidelines

DESIGN FOR VERIFICATION

Guidelines:

Pre design phase:

1. **Collaboration** between design and verification engineers
2. Make a design **specification**
3. Make verification **requirements**
4. **Emphasize** verification enablers and considerations in the specification
5. Document shared **resources** and **dependencies**

Designing phase:

1. **Update** design specification continuously as the design evolves
2. **Update** and keep a high level dialog with the verification engineers
3. Write **assertions** on the go, as design evolves assertions **should** be added in parallel
4. Write **appropriate** assertions for interfaces
5. Analyze and model any potential conflict
6. Watch out for copy-paste errors, exercise cation when doing copy-pasting
7. Watch out for old comments, they might be misleading
8. Make verification features accessible
9. Get a peer review of behavior scenarios, corner cases and areas of risk

Rules:

1. Use an automated code checking tool (Linting) to ensure that the RTL follows standard coding rules
2. RTL should be maintainable, possibly reusable
3. RTL should be understandable for the designer and independent designers
4. RTL can be in range from equation-based to behavioral
5. FSMs can be either explicit or implied
6. Write behavioral code in such a way that it is self-explanatory, if possible
7. Use parameterization thing that changes/might change in the design, it accelerates corner case reachability
8. Use error injectors, e.g. random invalidates, cache evicts
9. Make interfaces as clean as possible
10. Simplify where possible but within design goals and constraints

Appendix F: UVM coding guidelines

UVM CODING GUIDELINES

General Guidelines

- Do not use any features of UVM that are specifically marked as deprecated in the UVM Class Reference or base class library.
- Do not use internal features of the UVM base class library that is not documented in the UVM Class Reference.

Lexical Guidelines and Naming Conventions

- Only one declaration/statement per line.
- User-defined names for SystemVerilog:
  - **Variables and classes**, use lower-case words separated by underscores e.g. derp_fish
  - **Enum literals, constants, and parameters**, use upper-case words separated by underscores e.g DERP_FISH
- Restrict all user-defined UVM instance names (that is, strings such as component instance names) to the character set a-z, A-Z, 0-9 and _ (underscore).
- Use shorter names for local variables and longer, more descriptive names for global items such as class names and package names.
- Use the prefix **m_** before the names of user-defined class member variables (officially known as class properties in SystemVerilog).
- Use the names **m_sequencer**, **m_driver**, and **m_monitor** as the instance names of the sequencer, driver, and monitor respectively within every agent.
- Use the suffixes **_env** and **_agent** after the instance names of every environment and agent, respectively.
- Use the name **m_config** as the instance name of the configuration object within any component or sequence that has one.
- Use the suffix **_config** after user-defined configuration class names.
- Use the suffix **_port** after user-defined port names.
- Use the suffix **_export** after user-defined export names.
- Use the suffix **_vif** after user-defined virtual interface names.
- Use the suffix **_t** after user-defined type definitions introduced using the keyword typedef
- Use the suffix **_pkg** after user-defined package names.
- Write comments wherever they add value to the source code and help the reader to understand the purpose of the code.
- Include white space (blank lines, indentation) wherever it helps to make the code more readable.
- When overriding built-in UVM virtual methods, do not insert the virtual keyword at the start of the overridden method definition.

General Code Structure

- In structuring and coding the verification environment, think primarily about reuse.
- Use a consistent file structure and a consistent file naming convention throughout.
- Each class should be defined within a package (as opposed to defining classes within modules or at file scope).
- Use `include directives within a package to allow each class to be placed in a separate file

- Use conditional compilation guards to avoid compiling the same include file more than once.
- Do not use wildcard import at compilation unit scope.
- Include **uvm_macros.svh** and import **uvm_pkg::*** inside each package or module that refers to the UVM base class library
- Use one agent per interface, with a passive monitor and optional sequencer and driver whose existence is determined by the value of the **get_is_active** method of class **uvm_agent**.
- An agent should not instantiate components other than one sequencer, one driver, and one monitor.
- Use virtual sequences to co-ordinate the stimulus generation activities of multiple parallel agents, that is, to start sequences on the sequencers belonging to multiple agents.
- Checking and functional coverage collection should be performed in checkers, scoreboards, coverage collectors, and other ad hoc subscriber components that are instantiated externally to any agent and connected to the analysis port of the monitor.
- In general, connect agents, checkers, scoreboards, and coverage collectors using analysis ports and exports.
- UVM environments should be written such that they can be used as top-level environments or reused as sub- environments in other larger verification environments.
- Use factory overrides and/or the configuration database to adapt the behavior of repurposed UVM components to the needs of a new verification environment.
- A top-level module should set configuration parameters that are retrieved by the test, the test should set parameters retrieved by the environment, and the environment should set parameters retrieved by lower-level environments or agents.
- Represent layered protocols by having multiple sequencers, each with their own transaction type.

Clocks, Timing, Synchronization, and Interfaces

- Generate clocks and resets in a SystemVerilog **module**, never in a SystemVerilog **program** and never in the UVM class-based verification environment.
- Use SystemVerilog **modules** in preference to SystemVerilog **programs**.
- Use clocking blocks within a SystemVerilog interface in order to sense and drive a synchronous DUT interface.
- Use modports to enforce the use of clocking blocks when accessed through virtual interfaces from the UVM verification environment.
- Use modports that combine a clocking block with asynchronous signals in order to access an interface that combines synchronous and asynchronous signals.
- In the verification environment, try where possible to confine synchronization to signals in the DUT interface and explicit delays to drivers and monitors, with other UVM components being untimed.
- A driver should pull transactions from a sequencer using the non-blocking **try_*** methods in order to maximize reusability in the scenario where the author cannot know whether the sequence will block the execution of the driver.
- A driver should only pull down transactions from the sequencer when it needs them.
- If a driver needs to insert variable delays within or between transactions when driving the pins of an interface, this should be handled by storing delays in the transaction passed to the driver.
- Use the **uvm_event** or **uvm_barrier** for ad hoc synchronization between sequences and/or analysis components such as scoreboards.

- A monitor should not assign values to variables or wires in the SystemVerilog interface.
- Use concurrent assertions and cover property in interfaces for protocol checking and related coverage collection.

Split Transactors for Emulation/Acceleration

- For emulation/acceleration, have two top-level SystemVerilog modules, one module that runs on the host computer and instantiates the UVM verification environment and a second module that is synthesized and runs on the emulator or accelerator.
- The UVM verification environment running on the host computer should be untimed. It should not contain any delays or refer to any clocks. Any delays and clocks should be moved to the emulator/accelerator.
- Split each UVM driver and monitor into two parts, an untimed part that runs on the host and a synthesizable part (BFM) that runs on the emulator/accelerator.

Transactions

- Create user-defined transaction classes by extending the class **uvm_sequence_item**.
- Try to minimize the number of distinct transaction classes. Use the same transaction class for the driver and monitor of an agent.
- Register the transaction class with the factory using the macro `**uvm_object_utils** as the first line within the class.
- Do not use field macros.
- After the factory registration macro, declare any member variables.
- Use the **rand** qualifier in front of any class member variables that might need to be randomized, now or in the future.
- After any member variables, define a constructor that includes a single string name argument with a default value of the empty string, a call to **super.new**, and is otherwise empty
- After the constructor, always override the **convert2string, do_copy, do_compare, do_print,** and **do_record** methods.
- Consider overriding the **do_pack** and **do_unpack** methods.
- When overriding **do_pack** and **do_unpack**, use the packing and unpacking macros (e.g. `**uvm_pack_int**) where they will simplify the code.
- When overriding **do_record**, use the recording macros (e.g. `**uvm_record_attribute** and `**uvm_record_field**) where they will simplify the code.
- When overriding the **do_print, do_record, do_compare**, and **do_pack** methods, do not make use of the printer, recorder, comparer, and packer policy object arguments to those methods within the body of the overridden method.
- Always instantiate transaction objects using the factory.
- In general, the string name of the transaction should be the same as the variable name.

Sequences

- Create user-defined sequence classes by extending the class **uvm_sequence**, parameterized with the type of the transaction to be generated by the sequence.
- Register the sequence class with the factory using the macro `**uvm_object_utils** as the first line within the class.
- After the factory registration macro, declare any member variables (using the prefix **m_** as a naming convention).

- Use the **rand** qualifier in front of any class member variables that might need to be randomized, now or in the future.
- After the member variables (if any), define a constructor that includes a single string name argument with a default value of the empty string, a call to **super.new**, and is otherwise empty.
- Any housekeeping code associated with the execution of a sequence, such as raising and lowering objections, should be placed in the **pre_start** and **post_start** methods of the sequence. The body method of the sequence should only execute the raw functional behavior of the sequence.
- When generating transactions from the body task of a sequence, do so using procedural code with the following general pattern:

  ```
  req = tx_type::type_id::create("req");
  start_item(req);
  if ( !req.randomize() ) ...
  finish_item(req);
  ```
- Do not use the `**uvm_do** family of macros.
- Use the built-in transaction variables **req** and **rsp** within a sequence, unless there is a specific reason to choose different variable names.
- Only generate sequence items (transactions) from UVM sequences, not from ad hoc classes and not from UVM components.
- Always instantiate sequence objects using the factory. Instantiations should take the form:

  ```
  seq_name = sequence_type::type_id::create("seq_name");
  ```
- The string name of each sequence object should be the same as the variable name
- When creating a sequence object, always call the randomize method of the sequence object before starting the sequence.
- Always check the value returned by the randomize method and report an error should randomization fail.
- Start sequences procedurally by calling their start method.
- Only override the **pre_do, mid_do**, and/or **post_do** callbacks of a sequence class as a way to modify the behavior of a pre-existing "immutable" sequence class.
- Use the macro **uvm_declare_p_sequencer** to declare a variable **p_sequencer** in situations where a sequence needs access to the sequencer on which it is running.
- Where a sequence needs access to a sequencer other than the sequencer on which it itself is running, add a member variable to the sequence object and assign that variable to refer to the sequencer before starting the sequence.

Stimulus and Phasing

- Use a virtual sequence to coordinate the behavior of multiple agents.
- Virtual sequences should be started on the null sequencer.
- Have a top-level sequence running on each agent that selects between all permissible child sequences at random.
- Keep sequences as generic as possible: avoid writing directed sequences except where absolutely necessary.
- Sequences should not be phase-aware.
- Do override the run-time phase methods **reset_phase, configure_phase, main_phase, shutdown_phase** to generate stimulus, typically by starting sequences, but never in a driver, monitor, subscriber, or scoreboard component.

- Do introduce user-defined run-time phases where the above predefined run-time phase methods are inappropriately named or would cause confusion.
- When integrating multiple environments that each override the predefined or user-defined run-time phase methods, take care to order the phases correctly by introducing phase domains and synchronizing phases across domains.
- Do not override the predefined pre- and post- phase methods (e.g. **pre_reset_phase**), but reserve these phase for use when synchronizing phases across domains.
- Do plan any phase jumps carefully to ensure UVM components are left in a consistent state.

Objections

- Determine when to end the test by raising and dropping objections in any classes that may need to extend the test while they complete some processing. (This rule has changed significantly since the first preliminary release of these guidelines.)
- Call the **set_propagate_mode**(0) method of every objection to disable the hierarchical propagation of that objection.
- Consider the simulation speed impact of raising and dropping objections in inner loops, e.g. for individual transactions. Remove objections from inner loops if the simulation speed penalty is significant.
- Where a sequence is to raise and drop objections, it should call **raise_objection** in its **pre_start** method and **drop_objection** in its **post_start** method.
- Always perform the test if (**starting_phase != null**) before calling **raise_objection** or **drop_objection** within a sequence.
- When starting a sequence that can raise and drop objections, if you want the sequence to raise and drop objections, set the **starting_phase** member of the sequence object before starting the sequence.
- When calling **raise_objection or drop_objection**, always pass a 2nd argument describing the objection to help with debug.
- If the **kill** method of a sequence is called and the sequence can raise an objection, ensure that the **do_kill** method of the sequence is overridden to drop the objection.

Components

- Create user-defined component classes by extending the appropriate subclass of class **uvm_component** in order to indicate intent.
- Register the component class with the factory using the macro `**uvm_component_utils** as the first line within the class.
- After the factory registration macro, declare any ports, exports and virtual interfaces (using the suffixes given in the section on Lexical Guidelines and Naming Conventions above).
- After the ports, exports, and virtual interfaces, declare any member variables (using the prefix **m_** as a naming convention).
- After any member variables, define a constructor that includes string name and parent arguments with no default values and a call to **super.new**.
- Instantiate any components from the **build_phase** method.
- Always instantiate components using the factory. Instantiations should take the form:

```
var_name = component_type::type_id::create("var_name", this);
```

- The string name of the component should be the same as the variable name.
- The second argument to create should be the reserved word this.

- Where a user-defined component class extends another user-defined component class, care should be taken to insert calls of the form **super.<phase_name>_phase** wherever appropriate, that is, where the corresponding base class phase method performs some action.
- Where a user-defined component class directly extends a class from the UVM base class library and overrides the standard **build_phase** method, do not call **super.build_phase**.

Connection to the DUT

- Use one SystemVerilog interface instance per DUT interface.
- Use virtual interfaces to access the SystemVerilog interfaces from the UVM verification environment.
- Encapsulate virtual interfaces inside a configuration object in the configuration database.
- Copy virtual interfaces from the top-level configuration object to the configuration objects associated with agents or lower-level envs in the **build_pha**se method of the top-level env.
- An agent should check that its virtual interface has been set.

TLM Connections

- Make TLM port/export connections and assign virtual interfaces in the **connect_phase** method.
- Communicate between UVM components using ports and exports, including analysis ports and exports where appropriate.
- Use analysis ports and analysis exports (or objects of class uvm_subscriber) when making one-to-many connections between UVM components.
- When making peer-to-peer connections between components, connect a port (or analysis port) directly to an export (or analysis export) without any intervening FIFO.
- Communicate with an agent in one of two ways: either connect the analysis port of the agent to a subscriber or access the sequencer within the agent using a direct object reference from outside.

Configurations

- Use the configuration database **uvm_config_db** rather than the resource database **uvm_resource_db**.
- Group the configuration parameters for a given component into a configuration object and set that configuration object into the configuration database.
- Create user-defined configuration classes by extending the class **uvm_object**.
- Use the class name **<component_class>_config** or **<sequence_class>_config** for the configuration class associated with a component or a sequence, respectively, where **<component_class>** is the class name of the component and **<sequence_class>** is the class name of the sequence.
- Use the field name **"config"** for the configuration object in the configuration database.
- Do not register user-defined configuration classes with the factory.
- A component should typically get and set configuration parameters (typically configuration objects) in its **build_phase** method, as opposed to any other phase methods.
- Always check the bit returned from **uvm_config_db#(T)::get** to ensure that the configuration parameter exists in the configuration database.
- A sensible default value should be chosen if **uvm_config_db#(T)::get** returns 0.

- Each component should get the configuration object associated with that specific component instance, and should not get the configuration object of any other component instance.
- The configuration object associated with any given component instance should be set by its parent or by some other direct ancestor of that component instance, and not by any other component instance.
- Avoid using an absolute hierarchical path name as the 2nd argument to **uvm_config_db#T(T)::set**, and provide the shortest possible unique path name.
- A component instance may be associated with one configuration object or with no configuration object, and several component instances may be associated with the same configuration object.
- For an agent, include a variable **is_active** in the configuration object to determine whether the agent is active or passive. Override the virtual **get_is_active** method to return this value. Check **get_is_active** before creating and connecting the sequencer and driver within the agent.
- If a sequence is to be parameterized, the parameters for the sequence should be put into a configuration object in the configuration database. The configuration object should be associated with the sequencer on which the sequence is to run.
- The code that starts a sequence should get any configuration object associated with that sequence from the configuration database and should assign a variable in the sequence object to refer to that configuration object.
- If a component directly assigns the values of variables (including virtual interfaces) in its child components, it should do so in its **build_phase** method after creating those child components.

The Factory

- Always instantiate transaction, sequence, and component objects using the factory.
- When using a factory override to substitute a transaction, sequence, or component object with another object whose class extends the class of the original, the factory override should take one of these forms:

```
old_type_name::type_id::set_type_override( new_type_name::get_type() );
old_type_name::type_id::set_inst_override( new_type_name::get_type() ... );
```

- Call the static method **uvm_factory::get()** when you need access to the factory.

Tests

- Do not generate stimulus directly from the test, but use the test to set configuration parameters and factory overrides.
- Set up the fixed aspects of the verification environment and generate default stimulus in the env class, not the test class, so that the env will run even with an empty test.
- Where appropriate, use test base classes to define structure and behavior that is common across a set of tests, and create individual tests by extending these base classes.
- For reuse, avoid making tests dependent on the specific details of the verification environment.
- Use the command line processor to modify the behavior of tests without the need for recompilation.

Messaging

- To report a message, always use one of the eight standard report macros **`uvm_info**, **`uvm_info_context**, and so forth, rather than ad hoc **$display** statements or similar.

- Set the message id either to a **static string** or to **get_type_name()**.
- Set message verbosity levels thoughtfully and methodically to avoid unnecessary data in the simulation log file and to differentiate between messages intended for use during the development and debug of the verification environment itself and messages intended for use when running tests and debugging the DUT.
- By default, set the verbosity level of each message to a high number such that the message is less likely to be reported, rather than to a low number such that the message is always reported.
- Set message severity levels thoughtfully to differentiate between messages that are purely informational, messages that may represent errors, and messages that are certainly errors.

Register Layer

- If you use a generator to create the SystemVerilog code for the register model, do not modify the generated code.
- The top-level UVM environment should instantiate the register block using the factory and should call the **build** method of the register model.
- In the case of a hierarchically organized UVM environment where child environments use register models, there should be a single top-level register block that instantiates the register blocks associated with the child environments, and so on down the hierarchy.
- Any UVM environment that uses a register model should have a variable named **regmodel** that stores a reference to the register block for that specific environment.
- A UVM environment that has a register model should set the **regmodel** variable of any child component that also uses a register model to the corresponding sub-block of its register block.
- A UVM environment should only instantiate a register block if the value of the environment's **regmodel** variable is **null**.
- The variable name and the UVM instance name of each child register block in the register model itself should correspond to the name of the associated agent.
- A register block should only model DUT registers that are accessible by the UVM sequences associated with the immediately enclosing UVM environment.
- A UVM environment that uses a register model and that instantiates an agent should instantiate and connect a register adapter and a register predictor for that agent.
- A register model should use explicit prediction to keep its mirror values synchronized with the register values in the DUT.
- The address map variable **.map** of the predictor in each child register block should be assigned to refer to the corresponding address map of the top-level register block.
- A register sequence that reads or writes registers in a register model should extend **uvm_sequence** and should have a variable named **regmodel** that stores a reference to the corresponding register block.
- Before starting a sequence that reads or writes registers, set the **regmodel** variable of that sequence.

Functional Coverage

- Collect functional coverage in the UVM verification environment using the SystemVerilog **covergroup** construct.
- Where appropriate, collect functional coverage information in SystemVerilog interfaces using the **cover property** statement.

- Either place a **covergroup** in a class as an embedded **covergroup** or place a **covergroup** in a package and parameterize the **covergroup** so that it can be instantiated from classes in that package.
- **Covergroups** should be instantiated within UVM component classes as opposed to within transaction or sequence classes.
- **Covergroups** should be instantiated within UVM subscribers or scoreboards that are themselves instantiated within a UVM environment class and are connected to the analysis ports of monitors/agents.
- Instantiate the **covergroup** in the constructor of the coverage collector class.
- In order to collect functional coverage information for internal signals within the DUT, encapsulate references to hierarchical paths to the DUT in a single SystemVerilog module (or interface), then access that module from the UVM environment using a virtual interface and SystemVerilog interface in the usual way.
- Where coverage collection spans multiple DUT interfaces and thus depends on analysis transactions received from more than one agent, use the **`uvm_analysis_imp_decl** macro to provide multiple analysis exports in the coverage collector class.
- Group **coverpoints** into multiple **covergroups** in order to separate coverage of specification features from coverage of implementation features.
- Use a variable **coverage_enable** within the configuration object of the coverage collector to enable or disable coverage collection.
- Sample **covergroups** by calling their sample method as opposed to specifying a clocking event for the **covergroup**.
- Do not sample **covergroups** more frequently than necessary. Consider using a conditional expression **iff (expression)** with each **coverpoint** to reduce the sampling frequency.
- Sample values within the DUT or at the outputs of the DUT. Do not sample the stimulus applied to the inputs of the DUT. Sample DUT registers when the register value is changed by the DUT, not when it is changed directly by the stimulus.
- Consider setting the **option.at_least** of each **covergroup** and **coverpoint** to some value other than the default value of 1.
- Do not set **option.weight** or **option.goal** of a **covergroup** or **coverpoint**.
- Design **coverpoint** bins carefully to ensure that functionally significant cases are covered.
- When designing **coverpoints**, specify any illegal values or values to be excluded for coverage as **ignore_bins**. Do not use **illegal_bins**.

Appendix G: Survey conducted among SHMAC developers

New SHMAC Verification Framework

The goal of this survey is to gather information regarding automated testing needs on the SHMAC development

1. **During your development on SHMAC, you worked on:**
   Mark only one oval.

   - ◯ Hardware infrastructure (processor cores, system architecture, etc.)
   - ◯ System software (OS, libraries, etc.)
   - ◯ Application-specific accelerator
   - ◯ Other: ....

2. **Did you test the hardware while implementing your design? How did you do that? What tools did you use?**

   ....
   ....
   ....
   ....
   ....

3. **What difficulties/challenges did you face regarding verification when implementing your design?**

   ....
   ....
   ....
   ....
   ....

4. **What features should the new verification framework include in order to overcome these challenges?**

   ....
   ....
   ....
   ....
   ....

5. **Did you have documented test requirements and/or specifications?**
Mark only one oval.

- ◯ Yes
- ◯ No

6. **Were your test cases documented?**
Mark only one oval.

- ◯ Yes
- ◯ No

7. **Did you use any formal verification method?**
Mark only one oval.

- ◯ Yes
- ◯ No

8. **If yes, which kind of verification method did you employ?**

....

....

....

....

....

9. **Would you use formal verification if it was easy to apply?**
Mark only one oval.

- ◯ Yes
- ◯ No

10. **How did you check for corner cases? Which of the following did you use?**
Check all that apply.

- ☐ Didn't check for corner cases
- ☐ Constrained randomisation
- ☐ Fully randomised tests
- ☐ Directed tests
- ☐ Other: ....

11. **How did you get information out of a device under test?**
Check all that apply.

- ☐ Through printfs
- ☐ Using a scope
- ☐ Using a simulator and examining signals
- ☐ Other: ....

12. **Additional suggestions on the implementation of the new verification framework**

....

....

....

....

....

Powered by Google Forms