POLITECNICO DI TORINO

Master of Science in Electronic Engineering

Master Degree Thesis

Study of Verification Techniques for Digital Architectures

Supervisor
Prof. Mariagrazia Graziano

Co-Supervisor
Prof. Marco Vacca
PhD. Fabrizio Riente

Candidate
Zaid Rawhi Mohaidat
Matricola: 250156

Academic Year 2018 - 2019

Abstract

Nowadays digital systems are continuously growing in complexity and the ASIC industry is struggling to meet schedule. About two thirds of the industry are behind with their planned projects. Similarly, the industry is struggling to keep pace in terms of quality. In accordance with the studies reported in the literature, it can be noticed that there is a gap between the ability to fabricate and manufacture according to Moore's law and what can actually be designed in reality within a given project schedule. The first study was carried out by ITRS and refers to the productivity gap. The other one is a Collett study that describes functional verification and adoption of technology and again makes reference to the gap between what can be verified and what can be designed. A lot of organizations used to struggle to adopt advanced techniques: they were still using 1990 best practices in terms of directed tests and code coverage, implying that the industry has not necessarily kept up with verification techniques. It is clear that the ability to verify would improve significantly if organizations adopted more advanced verification techniques, instead of relying on older techniques that were state of the art in the early 1990s. Today roughly 40% of the industry has adopted functional coverage and roughly 40-41% of the industry has adopted constrained random. Assertions, which enable formal verification and go after these concurrent problems, have been adopted by roughly 37% of the industry. Such data show that the industry has failed to move forward in terms of advanced functional verification. Considering the importance of verification of digital architecture and the fact that in industry this takes up to 60% of time resources in a design project, this thesis is concerned with studying advanced techniques and methodologies for the verification of digital systems. The main objective is to apply the Universal Verification Methodology (UVM), now considered to be the standard approach to this matter, on both simple and complex integrated circuits. UVM is based on SystemVerilog and nowadays, it has become the first methodology adopted in the industry because it is adaptable and reusable, due to the fact that the code for a project can be used for a similar one with proper modifications, thus resulting in a significant saving in terms of resources. The terms "verification with SystemVerilog" and "verification with the UVM" are broadly synonymous. Indeed, the UVM is becoming so pervasive in verification practice that tool vendors are already beginning to offer precompiled versions of the UVM's BCL

(base class libraries), and built-in UVM-specific debug capability, as integral parts of their SystemVerilog simulation tools. It seems natural, then, to ask whether the facilities provided by the UVM should perhaps be integrated into the SystemVerilog core language and its IEEE-standardized language reference manual (LRM). SystemVerilog already has a range of verification specific constructs – in particular, temporal assertions, coverage, and constrained random generation. It is clear that the combination of SystemVerilog and the UVM provides users with a powerful toolkit that can be applied effectively to a wide range of problems in the domain of functional verification of digital hardware designs, increasing significantly the efficiency in the verification stage of a design project.

Acknowledgements

Prima di tutto grazie a Dio.
Vorrei ringraziare la Professoressa Mariagrazia Graziano per tutto il supporto durante lavoro perché senza di Lei non avrei mai concluso in tempo. Ringrazio la famiglia del 196 Ing. Ruben, Bianca, Cecilia, Alberto, Filippo, Lorenzo, Vincenzo, Vito e i miei Comandanti, Dario e Giuseppe. Un ringraziamento speciale al mio Anzianissimo Mohanad, a Mustafa e Hashem.
Un ringraziamento dal cuore per il mio fratello Alberto Nicolella e per mia sorella Laura.
E alla fine ringrazio tutta la mia Famiglia e le due persone più care nel mondo perché senza di loro non sarei mai arrivato a questo punto e soprattutto senza uno di loro non sarei nemmeno arrivato in Italia, Mamma e Papà grazie con tutto il cuore.

Contents

Introduction

Nowadays digital systems have become very complex and heterogeneous, especially the integrated systems that include on the same chip different parts of the system, such as the memory, the DSP, the A / D and D / A converters up to the microprocessor. With all this evolution in the field of integrated systems we always try to make products faster, more efficient and less expensive in terms of area and power consumption and obviously in terms of money. In order to give a product to the final user with low-cost with the time to market, it is very important to minimise the time and effort invested in the life cycle of the product. However a digital design before arriving to final user must pass within several transformations starting from the original set of specifications. In the process of manufacturing a Very Large-Scale Integration- Integrated Circuits (VLSI IC) three different steps are present:

1. **Design:**
   The design phase can be referred to as the transformation phase because this is when an idea is actually transformed into a real working system. So starting from the consumer's specifications to synthesis the system.

2. **Verification:**
   To ensure that the functionality of the system is the same of the waited one as described in the initial specifications.

3. **Test:**
   During the life cycle of the digital circuit or the system in general, periodically it is needed to check if this system including processor cores and components are working as expected. Generally, in order to satisfy in-field testing requirements this task is performed by running short, fast and specialised test programs.

Firstly, It will be given an overview of how the design of the digital system is done then it will explain what is the verification in details, and illustrate the importance of the verification in manufacturing the VLSI-IC and techniques are employed so as to optimize the verification and having better results.

There are two types of verification:

1. Formal.

2. Functional.

The formal verification has the purpose to verify that the design coincides with all the input specification trying to find the best input-output combinations which can be extensive and expensive in terms of time and resources. It can also be

referred to as static methodology and it is usually carried out using mathematical calculations and numerical methods.
Formal methods implicitly take into account any possible behaviour of the models that are describing the specifications for the desired system. They include many techniques such as equivalence, checking model checking and theorem proving. Such methods; formal verification, can be considered as valid way to prove that the system is behaving properly is to use. For instance, these methods can be used to verify few systems like cryptographic protocols, combinational circuit and some of digital circuit with internal memory.
However, to achieve a high level of accuracy and completeness using formal method it is not likely, in addition to these limitations this method requires a huge amount of available computational resource which intern can be of a negative influence of the overall process performance.
The verification process has to be carried out the by verification engineers in order to be able to determine whether the system will work or not prior to the test phase so as not to have any relative issues to the design in terms of logic and electrical design. In order to achieve the above-mentioned results functional verification will be adopted. Functional Verification is mostly considered to be the process of verifying the correspondence between an RTL design and its specification from a functional perspective.

Chapter 1

Design of Digital Systems

Before diving into the discussion of the various verification techniques, it is important to understand how digital ICs are developed.

During the development of a digital design, it goes through multiple stages. Starting with the very first set of specifications and arriving to the final product as it can be seen in the figure 1.1.

Every single stage of the design in the previous figure, is a transformation that corresponds, to a different description of the system, which intern is incrementally translated into a detailed description which has its own specific semantics and set of primitives.

The flow in the figure shows a top-down simplified design approach. The actual process adopted by industrial development companies is relatively more complex, which involves many iterations through various portions of this flow, until the final design converges to a form that meets the specification requirements of functionality, area, timing, power and cost. Typically; in the VLSI flow, the design starts with system specifications, which are technical representation of design intent. These specifications comes from the customer's requirement and describe in details a list of point of the device or the product, and it must include how the design should be executed. All of this are documented on paper without any practical execution. Once the documentation is performed that include the specification design are finished, the High-level synthesis (HLS) algorithms are adopted to convert the design specifications into Register Transfer Level (RTL) circuits [1].

High-level synthesis is a process that transform an algorithmic description to form of hardware that include the desired behavior to be implemented. Synthesis start whit the higher level of the specification and use a specific language to synthesis the circuit. The hierarchical approach is adopted in the development of a functional design. This is related to the issues arising because of the large-scale integration,

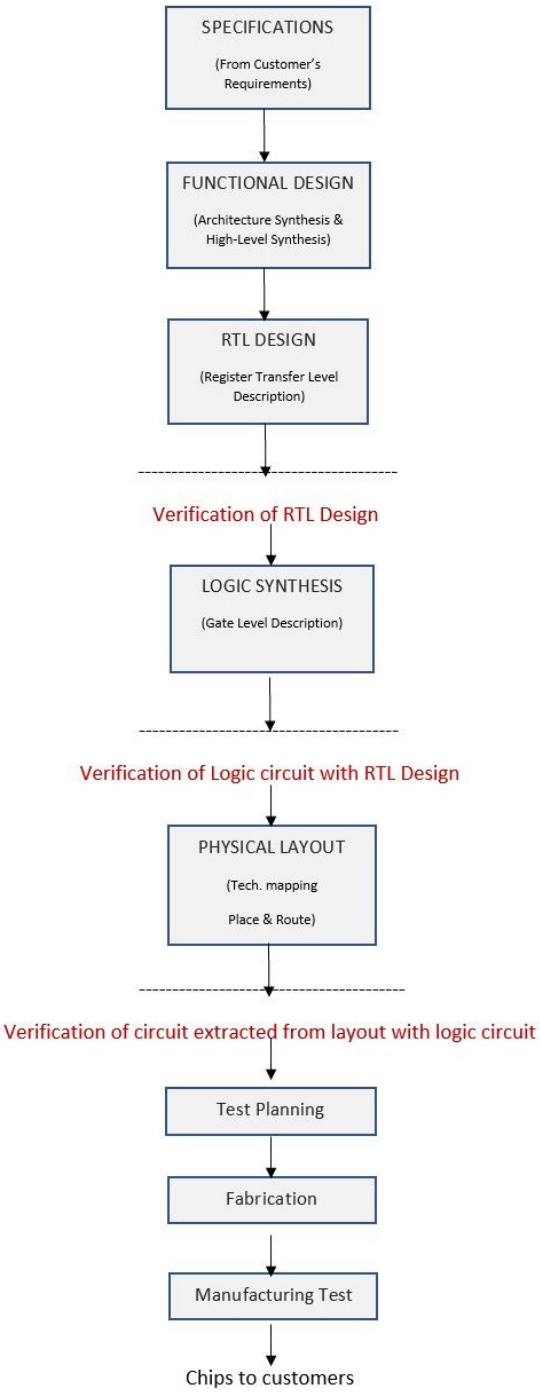

Figure 1.1. Phases of the Design life cycle of a Digital System.

so that a single designer can concentrate on a portion of the model at any given time.

Considering the functional design model aspects, the hardware design team proceeds from the Register Transfer Level (RTL) design phase. During this phase, the architectural description is further modified and optimized: by the adoption of one of the Hardware Description Language to be used in the design of functional component or other component such as memory element. RTL aims to the creation of high level representation of the digital circuit and it involves transferring sequences of data from registers, across other data path component likes multiplexers or adders, and back to registers. RTL Verification is the process of verifying the functional aspects of the design by generating different input stimulus and checking for correct behavior of the design implemented using an HDL.

Verification of RTL is done for the first time in term of verification for the design and it aim to remove errors that it will be more expensive to remove them if they are move on to the next phases. functional errors are highlighted the model is to be modified in order to fulfill with the proper behavior. During RTL verification, the verification handlers develop and adopt various techniques and numerous testing suites to evaluate that the design behavior meets the initial design specifications. Whenever these specifications are not met then the functional design model needs to be modified to provide the correct required behavior specified and the RTL design updated consequently. It is also possible that the RTL verification phase reveals incongruences or overlooked aspects in the original set of specifications and this latter one needs to be updated instead. In the diagram of Figure 1.1, RTL verification appears as one isolated phase of the design flow. However, in practical designs, the verification of the RTL model is brought forward up to the chip layout.

When the first verification of the RTL is done the Gate Level Description phase is come to represent the design as a netlist with gates (AND, OR, NOT, ...) and storage elements, all with cell delays. It generates a detailed model of a circuits which is optimized based on the design restraints. The design phases usually have minimal support from Computer Aided Design (CAD) software tools and are entirely hand crafted by the design and verification personnel. Automating the RTL verification phase, is the aim and to be performed for digital systems development. Once the synthesis and optimization are carried out the verification with no error is introduced into the design process, and this phase is an automatic activity to guarantee the functional correspondence between the model before and after synthesis.

When the output of the logic level is verified, the stages of the physical design will start. In this phase the conversion to the geometric form will be done. It consists of several steps to have the circuit layout. In order to do the steps of this

phase there are some of universal tools that can be used such as Synopsys or Cadence. The first thing to be done in the physical design flow is floorplanning which evaluates the architecture decisions by: providing an early feedback, estimating delays, estimating of chip areas, and congestion caused by wiring. Then the second step will be the Partitioning process, which divide the chip into a small number of partitions in order to get easier the next steps by separating the different functional blocks. After this step the Placement process remove all Wire Load Models (WLM) then it start optimization. The aim is to minimize the total area and the cost of interconnects.
The fourth step is clock tree synthesis (CTS) which aims to the minimization of skew and insertion delay and it insures that the clock gets distributed evenly to all sequential components in a design. The last step in this phase is the Routing process which locate a set of wires in the routing area that connect all the nets in the net list. However, in case of Detailed Routing the connection with more constraints such as DRC, timing and wire length will be done. The quality of routing is highly determined by the placement process. At the end of the physical phase a verification of the circuit designed until now will be done.

Then the project can be sent to be fabricated and the specific patterns can be applied in order to test the device, finally the new device can be putted i the market.

Chapter 2

Functional Verification Techniques

This chapter will overview the types of the second manufacturing phase; the verification process. Talking into consideration the design flow as explained in the previous chapter it can be noticed that the verification phase could be carried out in order to meet the desired specifications different times. This is obvious since it is common in integrated circuits design that there is a vast set of combinations of domains of the desired starting specifications, such as functional, timing, area, performance, and electrical.

Integrated Circuits trend has been moving in the direction of an exponentially increasing complexity which resulted in an exponential growth of the time required to be suitable to carry out the testing phase to the functional verification. Taking into account these facts the verification process of the functional specifications numerically utilizes approximately 60% of the resources [2].
In the complexity in ever changing due to various from simple systems and very complex ones; therefore a verification technique which is suitable for all digital systems is not possible, which make it fundamental to find a technique that can decrease the time and energy needed during the verification phase.

In the next sections SystemVerilog and Universal Verification Methodology (UVM) will be highlighted. It can be noticed that it is possible to compare a new project with a previous one, so requiring minor modifications the same verification environment can be reused to verify new projects, this would save a considerable amount of time and effort especially in a testbench.

The importance of the fairness and accuracy of the initial functional verification plan has to be taking into account in order to be successful in the functional verification approach. Coverage measurement, definition of the verification problem, different metrics are aspects to be used in order to give a qualitative measurement of the design progress; generation of stimuli, providing the necessary stimulus to

handle the project thoroughly following the given instructions; and study and control of the response, describing how to demonstrate the behaviour of the device in accordance with the specifications [3].

Then considering the various verification techniques, it can be noticed that there are two categories of the verification: static and dynamic verification techniques.

2.1 Static Functional Verification

Static Functional Verification known also as the Formal Verification can be based on the usage of formal mathematical methods in order to verify the design functionality. It has to be considered all possible combinations of behaviours for the circuits by representing the system and its specification [4]. The word "static" is given to this category because these techniques do not need stimuli and simulations to be able to verify the behaviour of the design. Verification engineers are not required to write a testbench or test vectors when they use the static method.
Using the static method could be useful for providing the correctness of systems such as: cryptographic protocols, combinational circuits, digital circuits with internal memory and software expressed in source code [5].

formal verification saves time and effort but it still has limitations and cannot be used as a complete substitute in place of simulation. Formal verification techniques are divided in two classes [6]: the ones that check the design against specification and the ones that check the equivalence between an already verified model (golden model) and the model to be verified. They are called intent verification and equivalence checking respectively.

2.1.1 Intent Verification

There are two well-known techniques in this verification class: the first one is Model Checking. This particular class generates all the possible inputs the are necessary for the system into consideration under all possible states and perform checks in order to determine if the properties obtained from the design based on the specifications of the model to be verified are correct. This technique can be used in the verification of small circuit of control logic. In order to obtain a considerable high level of coverage in this challenging method it is important to choose the desired proprieties with a minimum amount precision.

Model testing tools interface with the state explosion problem (rising in combinatorial of the state-space), that have to be addressed to be helpful in the solving of the most real-world problems. To resolve this problem many approaches can be

adopted, for example to avoid ever explicitly constructing the graph for the finite state machines (FSM) Symbolic algorithms can be used.

The second techniques under static intent verification is Theorem Proving, this method in formally involve axioms, interface rules and properties to express the specification and the model to be verified. Verification can be done by demonstrate the properties with use of inference rules and the axioms. The grade in converting the specification into proprieties can affect the quality of this technique. The advantage of this method is there is no limit in term of inputs or the size of the design state space and is well made for verifying datapath-oriented designs.

2.1.2 Equivalence Checking

This technique adopts a model already verified to be compared with the model under verification. It implies transformations of the two models to a formal logic representation and controls whether these representations of both models are identic. Using the model already verified gives the possibility to avoid the creation of proprieties. Equivalence checking is offered by an increasing number of commercial tools, like Tuxedo [7].

2.2 Dynamic Verification

Dynamic verification refers to "Testbench Simulation". Designers can use simulation by applying a pattern set to verify the functional correctness of DUV (Design Under Verification). To assure the correctness of the Design, its responses are compared with expected ones. In this approach it is necessary to create testbench, stimuli and responses by hand. In this approach there are also a couple of methods: Intent Verification and Equivalence Verification.

2.2.1 Intent Verification

In this approach to create the patterns it can be based on specifications. This leads to consume time because for each statement in specification we have to create a pattern to control it, and while the amount of patterns increase quickly, there is no guarantee to obtain design fully verified.

2.2.2 Equivalence Checking

This technique is performed by comparing responses of the model to be verified whit the responses of a model already verified (perfect model). In this case, pattern coverage can be improved by using pseudo-random stimuli in addition to deterministic

ones. This is common practice for covering corner cases in complex blocks like processor cores [8][2]. Sometimes a hardware model of the block is built using FPGAs to speed up random verification [2]. This is known as circuit emulation. Equivalence checking can be used in the verification of the gate-level designs against a perfect model in RTL or higher levels.

Chapter 3

Verification Planning and Management

The verification process is a task that requires successful providence in the form of formulating, architecting, strategizing and documenting an overall verification schemer. In this chapter verification planning and management will be considered in order to architect an overall verification overtures and to document it to become easily extracted, useful, maintainable documents that allow to save time and effort in the verification process.
Before start training of verification, a plan is adopted to be applied to the design. In the next section an overview about making a verification document is presented, but the real value add of this document, is to take it and do it in a verification and design team, and take it and apply it to a real design, so that get maximum exposure to it, and maximum integration into verification environment, and maximum retention.

3.1 Reasons about the need of plan

Why verification planning needed to be done at all? The first reason of planning is that verification is now the toughest job to be done. The ability to fabricate has just gone skyrocket, when they can fit huge systems on a chip now, and so the ability to design that and because they are reusing things, that has gone flying up too, and what has not kept pace is the verification. Verification's ability to verify has not kept pace. So there is this verification gap, and it just gets in the way of things coming back, and working the very first time. And so verification planning closes this gap, it is the best thing. We have lots of methodologies we can have verification for, but it is so big we have to plan, in order to figure out how it all fits together, in order to close that gap as much as we can.
Gathering some metrics, about what are the biggest concerns in the land of verification for most customers, well be noticeable that generally it is managing the

whole process, and then the second thing is gathering and then finishing coverage. And so this is another reason.
On large systems, many different possibilities have to be covered. So a plan beforehand in order to get the coverage closure eventually is needed. Also the number of spins that people are experiencing with their chips, has gone, it is gotten worse and that is again because they are bigger, and the reason for those respins, one is just design errors. So it is human nature and it is people cutting and pasting and not changing something.
It is little errors that can be founded all the time in the code. The second biggest cause, the next two have to do with the specification, and that the specification is incomplete or it was changed, and it did not track with the fixes in the design. Verification is about building a realistic world around the design, and to stimulate it and to check it, and to ensure that the intent of the design specification is preserved in the RTL. So having a specification, having RTL, making sure that they match, and building, a world around a design, that is going to be representative of the real world, and to verify and try to shove everything at it that might come at it, once the chip is out of the field. So the verification process itself includes verifying the specification itself. The specification might not be complete, it might have things left out, it might have things that are in error, it might get updated, and then be an error and not match.
So specifications itself have to be verified. The second thing is, making sure the verification, can handle anything that is going to be thrown at it, whether it be typical things and if a customer does wacky things with it, it is needed to be able to respond and come out of it, in a sane way, and then along with the classic verification, is once we're throwing all this stuff at it, making sure that it functions inside correctly has to be done, and that is the area of correctness, and lastly there is not all the time in the world, really and verify the most that we can in the allotted time that we're given, before they have to tape out the chip.

Designers start with a specification and they start building their RTL, and then they take it through synthesis and timing closure, and area closure and power closure and all those things, and they eventually converge it at the end. Subsequent to this, in parallel, most modern groups, are also doing this with a second set of eyes, a separate team as a check and balance. They are doing verification on the verification path, and they too have to build a world around the design, verify the verification environment itself, and then use it to verify the RTL and then reach some sort of closure, and hopefully the two meet at the same time, converge at the end where a tape out can be done.

The first metric to talk about is, what do you think? Is there more bugs in the design path, in the design path, or are there more bugs in the verification path? And most people, if you were here as a regular design team, you would probably guess

there is more in the design path, but the truth is that especially when one is new to verification, he is probably going to end up with more bugs in the verification path, than he will in the design path, which is somewhat problematic. After he get good at verification, it will probably get to be about 1:1, and again, if it is a team, it would to ask, what is the bugs schema? How many bugs there are in the design path? If metrics can be gathered on a project, things like how many bugs were in the design will be known, how many were in the verification, and furthermore, how many bugs were in the specification, or in the design, which ones were cut and paste errors, which ones were file errors, whatever kind of errors wanted. After this it is clear the importance of verification plan in order to make the verification path go smoothly, this is not building the design, the designers are doing that. This is building the world around the design.

So a verification plan is needed, with specification that says, how verification environment can be architected, and use it to the best of the ability? And so the first thing to do in order to make this go smoothly, is to make this plan. It is just logical. On modern chips of any size there are a whole family tree of specifications, to outline how the RTL is going to be built and how the code is going to be implemented.

In the same way, a verification plan, or a family of documents are needed to describe what this verification path is going to look like, and that is one of the things to be done, to accelerate the verification process. Now some old ways of doing this of course, is to just do all the RTL first, and then do the verification, and if you have the luxury of doing that, that still works. It still recommend to be a separate set of eyes doing it, but the days of doing that is over, because the convergent point needs to happen much sooner. Most people have a deadline, if they do not sell the chip by this time, they lose a whole bunch of revenue. So that path does not work anymore.

A common practice is starting the design and then starting the verification a little bit later would work, if it is not that complicated. But the bigger the chip gets, the bigger the verification problem gets. The verification path takes 70% of the effort [9] [10]. The verification work is always more than the design work, and it seems like whenever advances in verification is made, there is equal advances in design, and the designers are really doing a lot of reuse, so they can fit more in a chip, they can coat it up in RTL much quicker, than they ever have before, and so the verification path is just longer, and we've been having metrics on it for 10 years and it is around 70 to 80 percent, most of the time.

So does that mean that we have to write a lot more code in the verification? Are there really 7 lines of verification code for every 3 lines of RTL that we write? And the answer to that is no. One would think that the verification code should

be less. Because for example if one are making a multiplier, he have to build that whole multiplier out of a bunch of adders, or use some standard scheme in order to do that. In the verification path, he can just go A times B and it works fine, because it is not synthesizing that code, unless we're emulating.
And so there really should be, there is not that much code that needed to write. In fact metrics tell if the question is sked, it is known how many lines of code did usually to end up, with verification-wise and design-wise in this system, or source lines of codes. So how many source lines of code do you have in the RTL? How many source lines of code do you have in the HDL? And the metrics tell over several years is that it is about 1:1.

So if your design has 7000 lines of RTL, it is probably going to need about 7000 lines of verification code. And so that is a rough estimate, but that is what can be seen when metrics are taken, and that is one of the beauties of metrics, is it provide some form of feedback. So the lines should be about the same, so it must be something else that is causing this 70%, to be a phenomenon that does not go away. If we were doing it with your design, I would ask you what your ratios have been, if you've been gathering those metrics. So why is it at 70%? Everybody throws around this 70%, it is been around a long time, so why is it 70%? one reason is, I'm the first user as a verification engineer of your design, I'm your first customer basically, and I'm going to take that and put it in my world, in my environment, on my board, or combine it with these other chips to do something, and that world is always, almost always bigger than the design.

The second reason is that verification is open ended? start with the specification, build the RTL, there is going to be a beginning, middle and end, take that RTL into synthesis, take it into timing closure, and area closure and all that, and there is this deterministic way and you'll be done at some point.

Verification is not like that. Verification must be done forever and not trying every combination. There are certain things in verification that are just too problematic, and would take too long to verify. So we could verify forever so it is less deterministic, so that by its very nature, makes it longer and makes it push toward the 70%. The third reason is that most people do not plan. They verify just by happen stance, ad hoc. They just start coding and simulating, and by doing that because they do not do any planning, they do not layer their code, and if a language like SystemVerilog is used, which is designed to be object oriented, and it needs to be layered properly or have to be recoded, and they do not add coding for debug, so they end up taking a long time to make sure the code is right, and they end up spending a lot of time debugging and recoding. So it may be that they end up with the same amount of code and in reality, they might have rewrote that code 4 times, so it really might be 7:3 by the time they get done, because they have to

recode, because they did not sit down at the beginning, and have somebody with a software aptitude, to code things up in a way that you could easily extend, and add things and take things away, and that do not get into these code anomalies that are going to cause problems, taking a long time to debug and then maybe have to recode anyway. So it is about coding things right in the first path, in the first place. So this adds to the 70%, this recoding and this itself. So going up at least one level of abstraction, maybe 2, maybe 3 levels of abstraction, and if it is going to make a realistic world, to throw realistic stimulation at this, and take it into realistic places, that world's going to be a bigger world. And so that is the first reason why it is 70%. It's just a bigger world. Talking about the debugging problem if verification engineering are spending, 52% of their time debugging, they spend 34% of their time developing, the simulations that they are going to run, and 14% doing other things, like run management, file management, maintenance and all those types of things. So the big thing here is, why do they have to spend 52% of their time debugging? And, they are debugging the verification code, just as much as the design code usually and sometimes more, and so it is needed to plan upfront so that it is possible plan to debug. The aim is to get that 60% that it is at roughly now, and to get that down to like 30 or 40 percent.

The metric wanted is that one that can say the average time is like 6 days, or 5 days or something, in order to find a bug, recreate it, find a fix, make sure the bug is ready, and then test it on everything else. Whatever that number is, 5 or 6, it is better to get that down to 1 or 2 or 3 days, and that is the beauty of metrics. Top class verification teams will get that number down, into the 1, 2, 3 day time frame. But this is another reason why the verification, it takes 70% of the time, it is because time debugging spent is so much, debugging the code and debugging the RTL.

Every time a bug founded, it is necessary to figure it out, if it is a verification bug or a design bug, and even if design bug is founded, it might be way deep inside, and we have to be able to sleuth, and trace down to that, and so verification gets longer because of all that debug, so if debug time can be speeded up, verification can be speeded up also. Another reason is that the design path, has been around for quite some time, using an RTL like Verilog or VHDL and then synthesizing it down to silicone, has been around for about 20 years, and so there is a nice progression, they write a specification, they follow that specification and build the RTL, they take the RTL to synthesis, they do the physical on timing closure on it, but it has this nice progression, and one of the problems in verification is, there has yet to develop a nice progression that should be followed, that kind of mirrors the progression of the trends in the design community. Normally in the planning one need to architect a solution. Need to architect code. No software engineer worth his salt, would start a code, a project of this size, without first sitting down and planning it, architecting it, finding out what code is already available that he can

base it on, and so it is necessary to plan and that is about strategizing, what part of the code we're going to do, and how many sequences we're going to have, how many scoreboards we're going to have, how many assertions, where we're going to put assertions. All that stuff, you need to plan all that upfront, and then whatever we decide, we need to document that. And so that is the planning phase.

The next part is the building phase, and that is where verification environment is built and it is not necessary to have the RTL at first, because the goal is just to get that code all working and try to get as many verification bugs out as possible. We're going to end up with just as many verification bugs, so let's do that early. And so the testbench is built and then we kind of like, let it debug itself, a bunch of debug features is added, so that debugging is made easier and it is possible to try to get away from that 60%. Then the next phase is the run phase, and that is the classic simulation. That is when we're going to throw a bunch of stimulus at the design. We're going to check and see if it works. So we're going to simulate and we're going to hunt for RTL bugs, and when we find them we're going to prove that they are wrong, and then fix them and make sure the fixes work. Then the last part, is if constraint random is done, is you're going to let the stimulus environment, kind of randomly throw things at it, so it is necessary to see where it went, and more importantly where it did not go. So that is the coverage part, so it is possible to use structural and functional coverage, and a playground where it is possible to analyze that coverage is wanted, until reaching some closure. Use it is wanted to try and make mid course corrections, and take it into places we haven't been, and then eventually we want it to take us to closure.

So this progression is very positive. This progression is done and the time planning and building upfront is spent, and get all the, as many of the verification bugs out as possible, and it is going to make the run and cover time a lot smoother. We'll be able to do more within that verification time frame than if we don't plan upfront. And so you'll first notice that I've shifted the 70% over, because in reality that's the way it is. After 10 years of the metrics telling us that verification is taking longer, maybe we should just plan for that, and so upfront by making our verification plan can be started. How that can be done without the RTL? You know you're using the PCIX interface, you know you're using an ARM interface, you know your change management cost you days and days of problems left in design.

So whatever the problem is, one can solve beforehand and get solved, before you start, that's what can be done at the beginning, and can be started planning and then as soon as the specification arrives, it is possible to plan even more. So there is a lot of things we can do upfront, in order, that take up the 20%, so that we end up with about equal time to do verification and design. So the 50% is

somewhat of a pipe dream, it is unreachable. The other reality we see is that in most companies, this convergent point is a date and it is set in stone. They are going to ship something out then, whatever, it is just a matter of how much you can verify by the time you get there. It is going to go out as rev1, if it comes back working, great, if it does not, hopefully it will teach you some things, and they will just continue to verify after that and have a second rev, a third rev, but they need something out, right at that date, to put in the customer's hands, and so the convergent points are often not moveable nowadays.

The other thing that we see is that the verification team, should be about as big as the design team, if not bigger, it depends on the size of your design, but the statistics and metrics show that, it used to be about 30% were verification engineers, now it is very close to 50:50 and by 2010 it is probably surpassed, and on large designs, I've seen it to be 2:1, 3:1, even 4:1 in some cases, and so the verification team is now, the pivot point, of where you want to put your people, The last thing is that there is no silver bullet, you can't this verification path and make it optimal by getting, a tool's not going to fix it, a new methodology, a new library, it is a combination of things that you have to strategically sit down, and decide what you are going to do and what you're not going to do, to plot out what that verification path's going to look at to get to, and it is not going to happen in one project. It usually takes 2, 3, 4 projects to infiltrate, all these new good verification practices and for people to get up to speed on them, in order to get where we want to go. So there is no silver bullet [11].

The other thing is that this verification path, is all about software, and we don't like that as hardware engineers, but it is just the truth of the matter, and we kind of think, software is a dirty word, but for the hardware engineer, it is easier actually to teach some software engineer, a little bit of the hardware, then sometimes to take hardware engineer, and move them over into the land of software. And so on a verification project, you're going to need, several people who just get the whole object oriented program, who can just layer code and architect code in their sleep, because that's what they've done their whole life. We do need the hardware element, because we're driving hardware with it, but most of the recoding happens, because of just beginner newbie mistakes, in object oriented program, and the software engineer have moved passed that a long time ago, and know how to handle that.

So it is always a good idea, that if you need some high end verification people, to maybe look at your software environment in order to get them. The other thing that's very important, is to have progression, and there's plan, build, run and cover progression, it's just an excellent way to do it. Especially putting the plan and build part upfront, so that we have a plan that we're going to follow, with strategies and

architectures and we're going to build it first hand, and we're going to then find as many verification bugs as we can so that when we get the RTL, and we put it in there, primarily we're going to be finding RTL bugs. The other thing we've got to do upfront, is that the debug is taking up all our time, then we've got to plan for debug.
We need to have a whole debug strategy, that's going to make us pinpoint bugs, and just get that time to fixing a bug down to as short as possible. So the bottom line is that, and if you walk away with anything from this talk, is that that verification path is so complex and so important, that it's about verifying the most that we can in the allotted time. We need to choose our battles that we're going to do there.
We need to know what we're going to do and what we're not going to do and why. And this is the basic premise of doing verification, because we could verify forever and never get to anything, and we only have a certain chunk of time, so we need to strategically plot what we're going to do, and that is why we need to plan, and that's why we need a planning document.

3.2 The Difficulty of Verification and Planning to it

In general, a separate set of eyes is needed to look at the verification problem from the design, just so that you don't run into a problem where a designer reads a specification wrong, and he reads 2+2=4 and instead of that he reads 2+2=5, and then he puts in a checker assertion that checks that 2+2=5, and then, a year down the road you have a problem. It is better to have a second set of eyes look at it, and say 2+2=4 not 5. The verification process takes too much time, so it is necessary to start early with planning and doing the documentation. What will be done, in a sort of verification document, and then will be started from there so that it is possible to verify the maximum possible things in the available time. In the figure 3.1 that describe the normal verification process going on in parallel with design process, it can be noticed that the verification path needs a progression.

The design engineer have a nice progression, so we need a progression as well, and that progression that we suggest is to plan and to build the environment, find as many verification bugs in verification documents or in verification path as we can doing that, then fold in the RTL, and run simulations and find RTL bugs, and then check the coverage to make midcourse corrections and take it to closure, and that is the nice progression that we can have.

The other thing is generally 60% of the total time is spent for doing debug and verification, and so an environment can be made it has good debugging capabilities, and shorten that time down, shorten that percentage down, will be possible verify

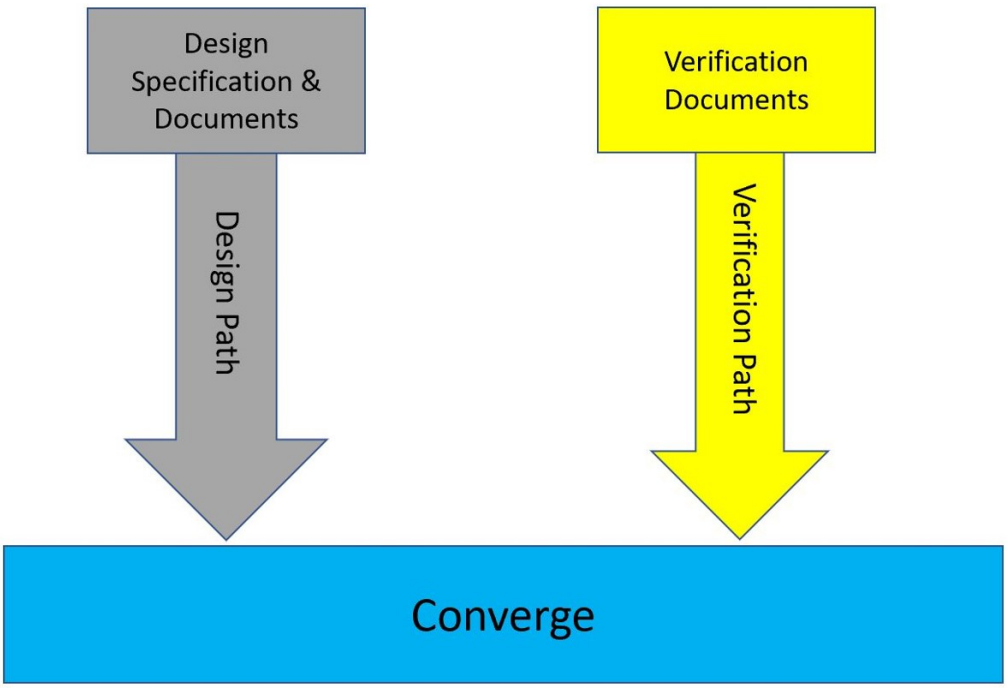

Figure 3.1. Verification Process with Design Process.

other things in the time available, and that would be a great thing to do [12].

The first reason of the difficulty of verification is that it's a lot of things needed to do in verification. The days of just putting a few lines on the input stimulus, and then looking at the output in a wave form, that is long gone on any design that has any amount of complexity to it, and so usually we start with some sort of model under verification, it might be an FPGA and small, it might be a huge system on a chip that's pushing the bleeding edge, and we have to build the entire world around that design, and so we start by building on every interface, we build a bus functional model, or so called a driver sometime, and that's going to wiggle the pins for us, as we tell it to do a read or a write or a send a packet or something like that.

On top of that monitors are needed on these interfaces, and maybe into the chip to see what it's doing, and so that we can see that things are operating properly, we usually put some sort of scoreboard between interfaces, that mimic what the chip is doing and so it makes an expected to the actual, and we can compare, so that the data went in at the right time, it came out at the right place, in the right format. Transfer functions to do that is also needed, protocol checkers on every interface, to make sure it followed all the timing rules, and nothing is wrong there. There are other checkers or assertions, that may breach into the design to check

state machines and stuff, just to see that they are all operating as expected, and then there are a stimulus generator or a sequencer, that are going to, on every interface, provide interesting stimulus, that is realistic and that will take things into the typical areas, and maybe even some error, or wacky errors, and there will be a lot of sequences that get run on these generations, to try different use models of different configuration, different kinds of traffic, anything that might break and find a bug inside the models under verification, and then if all that generation area is made random, a sort of feedback mechanism is needed to tell us where it went and where it has not gone, so a new sequences can be made to take it into places that it has not been, and that's functional coverage, so all this are putted together and usually that is called a testbench, and making one of these testbench require a lot of work, as it can be seen in the figure 3.2, there are a lot of components.

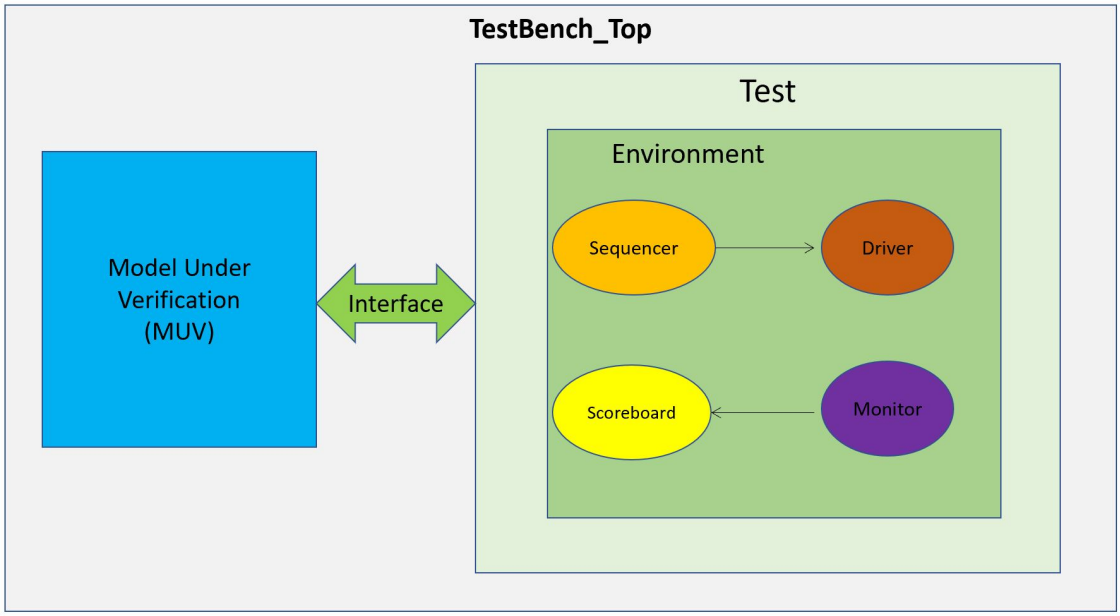

Figure 3.2. Main Components in the Verification Environment.

But there is even more, if we are going to do this coding right, following a language like SystemVerilog, then layer that code so that we can easily extend it, and add errors for example. So we have to layer the code, we usually don't want to start from scratch, so we want to use some library, that has a base methods in it, to like write out things, and build components and stuff on, like OVM, or UVM, and that's always useful to do, because it sets everybody on the same page, and saves you a lot of coding, and then we need some sort of usually configuration, or broadcasting mechanism, so the one interface knows, what the other interface is set at, so it can send the correct data, or the non correct data if we are testing an

error, and so we need broadcasting mechanisms.

Usually all of this is being driven by some sort of requirements database, that tells us all the things that we want to try, and we usually run multiple test cases with multiple seeds to keep that all running. We need scripts and conventions, so that everybody codes and name things, the right way, so we can easily find them when we're debugging.

We need to track what's going on and try to keep to a schedule. We need to gather metrics so that we can make midcourse corrections, and find out when we are done to closure. We want to follow a schedule and generate reports for our management. All that's part of doing the verification process, and all of this needs to be in some nice playground, where we have a database and file structures and change management, so we can keep track of all the files, So everybody has a nice little sandbox to play in, and we can keep it all, up to snuff and make sure that it works.

So, this verification land is more than just a testbench, it is a lot of different components, it is scripts, conventions, tons of code and layering the code properly. And then if planning it at the beginning is not done, there will be a problem. The term verification infrastructure is used instead of testbench because the word testbench is overloaded, and inside the verification infrastructure there are all things to do verification in safe mode.

However, verification can be seen complex and hard to do because there is a shopping list, and this is not an exhaustive shopping list. There are everything from the tools and learning them and solving them, to the stimulation, to reuse, to doing documentation for it. There is a lot of things that can be done, and in another way, it is the processes, there is all this verification processes to be followed, and this is not exhaustive either.

A verification plan, a regression plan and triage plan should be written, and then the debug must be checked so it is possible to get the debug numbers down. So, the first reason of difficulty is because there is just a lot to do. It is 70

The next reason is because engineer do not want to change the methods and strategies that they used from the beginning to change the society (for example the invention of smart phones).

The next thing when new tools and methodology are available to be used, it must be supported and trained, it should be easy to learn and compatible with other tools otherwise there will be some problems. Other reason of difficulty is that verification process is consist in several steps and in each step there is someone more able to work better than the others, may be one person is excellent in planning for verification but he have a limited knowledge in running in Questa. For this reason everybody must know his ability in order to put the right person in the right place.

Building an environment where all the simulation runs can be saved, and get them all to play nice with each other, and then triage the results and get closure, it is steep. Some of it is an art form, like closing on coverage is somewhat of an art

form. to be able to figure out which parts needed to cover, and which parts not needed to cover.

Organization do not want to change methodologies because they are doing a chip that is 10 times the size of project done before, that is why needed to change. And so the technology have to aligned, right people must be got, and realize that they take a lot of time to get it caught up to speed. People will just want to do their old legacy stuff, because it is familiar and it works.

In any process, the best thing to do is to have some visibility into what is happening, so information can be extracted out and then that information is measurable, and then track it over time and use it as a guide to make improvements, and that is the concept of metric driven verification, and more than one metric is needed. In the next chapter metrics will be considered in detail.

3.3 Planning and Attacking

Plan to verification is needed because verification takes more effort than the design, and because it takes a separate, informed, dedicated team, to strategically architect today's verification solution, and then to document it.

So this process, this team, this planning is needed, in order to be able to verify the most possible thing in the allotted time, and if moved away from just ad-hoc verification is done without strategically plan, then move to a place where we're planning and architecting is needed, to get the best results possible and as seen in the previous section it is hard because there is just a lot of thing to be done. It is way more than just building a little stimulator to wiggle some pins, and then to review the results. It is a whole verification package that, with a lot of infrastructure, in order to be able to run all these simulations, and to be able to triage the results and find out which ones are good, and bring it to some sort of closure and to guide us as we do our verification, so we can do the most verification that we can in the time that we have.

Verification is hard because people are resistant to change. Engineers are just like everybody else, they do not want to change. The number one reason why it is hard is not a technology problem, it is a resistance from people and from organizations, who just want a quick fix and there is no quick fix. It takes more than just a little tool, or a little library to do this, it takes all of that and training and expertise from somebody who's been there, who's architected a lot of these testbenches, and can help to deploy all of these solutions together, to make something that's viable.

Also that realized that it takes time to grow these people up, grow up the organization, and change them and migrate them and mature them, it takes, for the processes to mature, it takes for the people to mature, to become higher level and all of that takes time.

We want to not code ourselves into a corner and have to recode. We want to

choose which verification battles we're going to try to solve this time, and not try to solve them all. We're going to honor the reality that it's going to take time to grow this, and maybe schedule over 2 or 3 projects, to get us up to level 5, the top level, expert in this area. And so this is what it's about. I mean, we have a problem and we want to change to solve that problem, and so we have large quantities of data and stuff that we have to do, and large processes and methodologies we have to do, we have to wrap our arms around all of that in order to be able to do that, and so that's not feasible.

As engineers we take all that data, all those processes, all that information, and we break it into smaller, more doable chunks that we can do, and we can solve this problem, then we can solve that problem, and then we can solve this and we choose our battles, and that's what engineering is about and that's what effecting real change is about. And so we need to wrap our arms around this verification problem, and we need to have a plan of attack to do that.

Verification can be divided things up into 3 categories, Plan, Populate and Pilot. Now the Planning is the big picture, it is architecting, and strategizing an overall verification solution. It is starting at the high level, the 30,000 ft view, and going down to the 1,000 ft view, and then deciding what you're going to do and making decisions, and then documenting them in some form of verification documentation.

The second part is the Population. Once you have the plan and you start building the infrastructure for that plan, then you need to populate it with the details, and this is usually called the land of requirements, and this is the 1,000 ft view down to the 1 ft view, it is the nitty gritty details. So, for instance, in the architecture I might say that my chip, needs these 2 sequencers on these 2 interfaces, and then the super sequencer to coordinate between the two interfaces, and that is the architecture, that is in the red planning phase.

But the population phase is, once I have that, and the interface, need a bunch of sequences. So the detailed list of sequences are down in the population phase, and then the super sequencer on top needs to have a series of sequences, that try between the two interfaces and those are listed in the details. So you can see how the Plan and the Populate go hand in hand. Whenever you're doing any type of documentation or anything, there are certain things that are best done in small details, and packages and then like throwing them into spread sheets, those are your requirements, and there are other things that are better by drawing a diagram, like a block diagram, or making a table or a chart or a flow chart, where a picture is 1000 words, and then a couple paragraphs explaining things in like a Word doc or an HTML thing, and that is what is done in Plan.

So you need to find the sweet spot of what you're going to define in the planning part, and what you're going to define in the population part, what you're going to do in a Word doc and what you're going to do, in a detailed requirements spread sheet. So we divide things up into the Plan and the Populate, but then the ongoing day to day stuff is also a lot of work, and metrics tell us that people are spending anywhere between 25 to 40 percent, of their time, just managing their verification environment. Just making sure all the files are ready and everything, because they're so big and so complicated, and so the Piloting is where that comes in. It's the day to day operations. It's managing the entire verification process and keeping it on track, from the beginning to the end. And so we need all three of these to get going. So let's look at each one more in detail.
The first one is the Planning. The Planning is the big picture. It's the missions and goal. Your mission might be that we've only done directed test cases, and that's running out of gas now that we've got a chip that's this size. So our mission then is to move to constraint random, or maybe you've done constraint random but in the first two projects, you really didn't get any coverage closure, and you don't really understand how to do functional coverage right and close it.

So your mission then is to do functional coverage right for the first time. So there's high level things like that, that we discuss during the Planning phase, and come up with a mission and a goal, and maybe a prioritized list of things that need to be addressed or fixed or looked into. And so we're strategizing, we're architecting solutions, we're making decisions, we're dividing the verification environment, as you do with something like in SystemVerilog in object oriented program, into abstraction layers, transaction layers, and then we see that we have a lot of code to write, and so we can't code it all up at once, so we code up maybe this interface first and that's phase 1, and we have a goal, a milestone to that we're basically configuring the chip.

Then we have phase 2 where we add this other interface, and we start tracking, traffic, and then a third phase, then a fourth. So we bring things up in an orderly way where we get results as we go along, and we build the environment in an organized and strategic fashion. And we take all those decisions and ideas that we came up with them, and we're going to document them in a verification document. And so the questions and answers that we're doing during this planning phase, is what and how and in what order are we going to build the actual testbench. And the areas of focus are always the same, it's the generation.

Coverage, what would be interesting information to get, that would help us to guide the simulation into other areas? We start looking at the main areas of where we could get coverage. So this part is the hardest to automate. It's really just getting the right people in a room together, and brainstorming and coming up with solutions, and getting buy in from everybody. We can get a little bit of automation

or a little bit of help here, by using libraries and maybe reusing previous code. But the Planning phase, this 30,000 ft down to 1,000, architecting a solution, throwing it into a document, is basically a manual process at the beginning, and there's not much you can do to get around that.

So the results of the Planning phase, is making a Verification Architect Document (VAD) and a Verification Implementation Documents (VID), and we usually don't use the word verification plan, because what does that mean, That term is overloaded, and for some people it means a little less of test cases, to other people it means 1,000 page document with specific little details of tests in it. So we follow the rule of what good designers do, which is they make these macro architecture documents, and micro architecture documents and implementation documents, and we follow the same thing, or we start with kind of a parent document, a Verification Architecture Document, and it's going to outline the block diagram, our mission, our goal, the general flow of the testbench, a component list, the phases that we're going to use, the layers, and we'll probably even put some implementation information, but as soon as that document gets pretty big, we might break out a separate implementation document, for a specific generator, and in that document we're going to outline good coding practices, like they do in software, which these are the, this is the interface classes, this is how you extend the class.

These are the methods that are public that you can address, and these are the data fields that are public and that you can address, so you can make implementation documents. So out of this Planning phase we start with this VAD, and then that VAD, as we move through the project, might grow, and be split out into something more subsequent. But there's always this VAD, that is a parent document, that's going to drive everything we do, and specifically it's going to drive, the building of the verification infrastructure, and as explained in the previous section, what does mean by that is everything that you have to do in order to do verification, it's the testbench, the database, everything you need to do, and that's going to be outlined, how to do that, in the VAD, and then, so armed with a VAD, a verification team can build out a schedule on the component list in things, and then start building their verification infrastructure, and that's mainly the deliverables, it's the block diagram, list of components, what we're going to reuse, what we're not going to reuse, what version of OVM we're going to use, and OVM is pretty big, so which parts of OVM are we going to use.

And OVM is flexible to do things certain ways, so you can make choices and document them in your VAD. And so the VAD is kind of a catch all, of all of our decisions. It's our guiding document, just like a design architect document, drives the building of an RTL, this VAD is going to be building a world around the design, or the verification infrastructure, that's going to go around the design.
We need to talk about extracting of the requirements from Population details from

design documents, or from other methods, from other areas. We need to refine them so that they're clear and concise. We then prioritize them and then we want to keep track of them, in some way, shape or form. So during this Population phase, we're answering these questions of what and when specifically, we are going to throw at the chip and what are we going to apply to the DUT.

And so, what is going to be the sequences we're going to try? What are going to be the checks that we're going to perform? What is the scoreboard going to check and what is it not going to check? What are we going to cover? And these are the kind of questions we're answering here. And so the focus again is in the same areas, generation and sequences, score boarding and assertions for the checking and coverage.

Requirements can be design requirements or verification requirements. It might be one or the other or it might be both together. It depends on what you want to do. I've seen teams do it every single, a bunch of different ways. But if we extract out a bunch of design requirements, which are just little, kind of chunks of the specification, we're going to do this type of configuration. We're going to do this type of state machine. We're going to do this type of interface, these modes in the interface and such. And so we can extract out a bunch of design requirements and write those.

If the design requirement said that we have to do this particular, kind of configuration, and of course, that's going to branch out, into many verification requirements, we need to be able to have the sequencer generate and configure the chip, into that configuration, we need to then have it control other sequencers, to drive corresponding traffic depending on that configuration, and on down the line.

We also need to configure the scoreboard to handle that type of data and configuration, and then we might have to cover all these different permutations. So very quickly, you can see how one design requirement, can all of a sudden be translated and mapped into a whole bunch. You have this one to many problem.

All the results help us to analyze and debug the code, debug is the big thing that take the most time, so if we can streamline that, that's even better, and we want a playground that lets us debug and find out, which simulations were good, which ones were bad, which ones have a bug, what kind of bug it is, where do we go fix it.

We also would like to be able to extract all this information metric data, of the whole process. How many check ins are being done and what's the change management? Is it reaching a steady state?

The high level strategy and the VAD creation for the Planning. The requirements and the spread sheet for the Populating, and the Piloting is to sit down at

the beginning, and come up with kind of a users guide to how you're going to do things, so that everybody can follow it. some groups it's ideal to do the Plan and the Pilot, or the Population and then the Piloting, but of course you can do them in different orders.

When you run simulations, you store all coverage information, in the UCDB inside of Questa, and then you can backtrack it and connect it to your requirements, inside of a spread sheet and inside the database.

Then we have the Triage Analysis and the Results Analysis, that can be used to look at log files and figure out what's going on. That's automated, if you've ever looked at simulations, and had to do all your own awking and grepping to find things and everything, this is a way to do that, inside of the new version of Questa, it can marry with what you already might have, scripts and such. And then the Metric Driven Verification is being able, to have a bunch of metrics from the requirements, from the job management, from the simulation results, from the coverage, everything, and to be able to make a dashboard so that you can look at things.

To summarize, we have a lot to do in verification, so we need a plan of attack to do that, so we've divided things up into the three areas, the Planning, the Populating and the Piloting, and the Planning again is the 30,000 down to 1,000 ft view, the overall structure and the strategizing, and then documenting that in what we call a Verification Architecture Document, which is just a new name for the word Verification Planning, because that's so overloaded. But it's actually just the architecture, is a great way to think about it.

And so that's going to guide the whole building of the testbench, and describing what parts are important, and what parts we're going to solve this time, or what parts we're going to solve over the next three projects, and what parts we're going to leave till future projects and all that. That can be documented in this architecture document. This is the number one thing you can do in order to jump start, your verification environment development.

The second part is the Population. If you had that plan then this idea of requirements driven verification, where you can populate your verification infrastructure with specific things. So this is all about the detailed requirements, it's a list of sequences, a list of assertions. It can also include design requirements all connected to verification requirements, which are then connected to coverage, and that's in the Population part of it.

And the last part is the Piloting, because even if you have a good plan, and you build a good environment and you tie it up with requirements, and hook that to

coverage, it produces this kind of wild verification environment, that it's just hard to keep your hands on, and so you need to pilot it in a controlled environment. You need a safe playground for everybody to play in, and that's what the Piloting section is all about. It's having good change management and run management and triage, and hooking up a metric dashboard so that you can see what's going on, and make midcourse corrections.

Chapter 4

Metrics in SoC Verification

Process metrics mainly can provide a wide set of clear, quantitative and objective measures for the assessment of process performance and the actual accomplishments in terms of the specified process goals. SoC functional verification is to involve the integration of multiple IP blocks. Indeed, understanding the ways in which appropriate IP and system-level metrics can be defined, measured, correlated, and analysed is fundamental for improving performance and achieving quality goals.

Metrics in SoC Verification is considered one of the key areas since SoC verification is increasing in terms of difficulty in order to understand what has been verified and how to optimize the process. Peter Drucker is known as the Father of Metrics-Driven Processes and had a flair for many different types of quotes related to metrics; to be mentioned below three of these metrics are:
"What gets measured, gets done;
What gets measured, gets improved;
And what gets measured, gets managed."

Actually, if it can be thought as follow; what doesn't get measured, doesn't get done, doesn't get improved and likely isn't being managed properly, which is another way to think about these three metrics. In this chapter the philosophy of a metrics-driven process will be the main topic. In other words, the implementation will not be explained here. And the reason for that is that the implementation is very project-specific, but what is to be done at first is to understand what is required for a metrics-driven process before deciding how to implement it.

Theoretically, metrics are nothing but measurements. They are measurements that provide clear vision into the process, and once this vision is obtained, it will be useful to identify issues with the process, and it allows to take adjustments before things get out of hand. The other beautiful aspect of metrics is that once they are gathered these multiple metrics over a project, historical trends can be plotted

and this will facilitate furthermore the future planning which will also be more predictable. Actually, there is not a single metric that can portray a project state; it would take multiple metrics. And since it requires multiple metrics, it's important to pick the proper metric. After all, metrics can be expensive to implement, so actionable metrics must be picked.

Before talking about how to pick actionable metrics it is important to know a little bit about the history behind metrics in terms of metrics-driven processes. Probably one of the earliest references in metrics-driven processes go back to Deming awarded for quality. He was also known for the 14 Points of Quality. He went into Japan after World War II and revolutionized their manufacturing and auto industry by introducing quality, quantitative, and metrics-driven processes. He was a big believer in metrics-driven processes. In fact, he was often known as saying: "If you can't measure it, you can't fix it." [13]

Now, other work that's been done in metrics-driven processes relates to the capability maturity model. Capability maturity models came about in the late 1980s, and essentially what was happening is that addressing issues in the industry related to software quality and productivity was the main target. So what came about was the development of this capability maturity model. In the figure 4.1 the five different levels of maturity an organization typically goes through are listed. The important thing is that metrics become a foundation of a capability maturity model. Starting about level three, projects typically start introducing metrics into the process. Once they get to level four, they are actually using these metrics not only to identify problems, like they were in level three, but to predict future projects. They can become a lot more predictable because they have historical trends. At level five they are using these metrics to identify inefficiencies in the process and optimize their process.

Getting back to how actionable metrics can be defined, it can be viewed in three steps. The first one is the identification of goals in process; what it is that is to be accomplished. Once identification of these goals is completed, it has to be checked the determination of achievement of the goal coming up with a set of questions will be helpful. Then, coming up with a set of measurements in order to answer those questions. Following this process helps to identify actionable metrics. The important thing is that the metrics have to be tied back to a goal. A broad set of metrics without being tied back to a goal lead to obtain very inefficient process. This chapter concentrate on: what is changed in today's SoC verification flow that is driving the need for metrics-driven processes; the measurements that could be introduced into the flow, and then how these measurements can be used; the considerations needed to think about during the architecting of a metrics-driven process; considerations related to implementation; and what to expect once

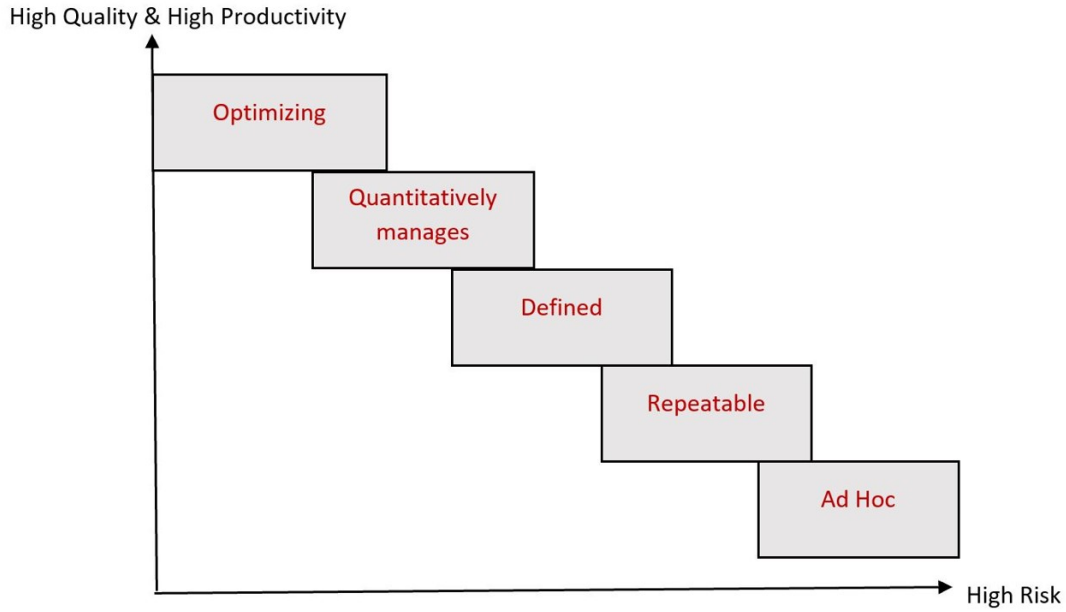

Figure 4.1. The Five Different Levels of Maturity an Organisation typically goes through.

a metrics-driven process have adopted and implemented.

4.1 Driving Forces for Change in SoC Verification

System on Chip verification has been changing in the last few years, the number and types of IP blocks were changing. Previously each IP block pretty much operated independently of each other in general, and SoC verification was mostly a matter of taking existing, well-known working IP blocks, arranging them together and making sure that the interconnection worked. For the most part, the blocks didn't interact much with each other, and they might have been sharing some things, but not much and so the SoC verification was mostly a: are they hooked up, are they talking to each other, and then system were done.

In the earlier designs, understanding the internals of any IP block wasn't necessary at the SoC verification level. One had to know how to read and write registers, maybe do some level of programming, but not too much. And metrics could be

used, they could provide bus utilization if it is wanted to, but they weren't important for tests, they were not important for performance analysis. The trend with IPs is towards increasing complexity, and as a result, the SoC itself increases in complexity as well. And there is now greater necessity to test IPs together inside the SoC environment. Even when the IPs are operating independently of each other for the most part, they're still likely to share resources, they may be fighting for resources, and that means that the system behaviour is first really seen at a higher integration level. It may be at the SoC, it may at a multi-IP.

Nonetheless, it's beyond the scope of one IP group. And it means that as part of the integration effort, other measurements are necessary - perhaps bus utilization, perhaps resource sharing, measurements of what resources are being shared and how effectively they are being shared across the IPs. And those need to be done now often at an SoC level to determine correct behaviour of the full system.

Now IPs are becoming more common, they are significantly higher in complexity, and there are two major changes, one of which is that the IPs will often talk directly with each other. One example, the most obvious example is a coherent cache. Each IP has its own cache, it can be tested independently to some level at the IP level, but the full distributed cache environment really is only going to get turned on for the first time when there are multiple IPs within an SoC environment. The first time that one really can look at the interactions are when it has multiple IPs all running together. And this causes some complexity. It is no longer enough to simply plug the IPs in and make sure whether they are connected because there are now going to be new interactions that have never been seen until stay at the SoC level. So, this creates some new challenges for the SoC integration team. The IP blocks, they tend to be very complex. They will often have entire teams doing design and verification of just the IP block, which means to the integration team, they are really a very big, complex, black box. It's challenging to look inside of them, and it's very difficult to see if they work correctly, if the stimulus worked correctly.

Understanding what happened can be very hard. Similarly, the IP team has an equivalent of the other side of the problem where they understand their IP but they do not often see what the integration looks like, which means the first time that their IP is really turned on is happening in an environment that's outside of their control, and it may be difficult for them to get a reasonable understanding of how their IP is functioning with other IPs easily. They relying on the integration team to see that. And, as it was mentioned before, as a rule of thumb: if it cannot be measured, it cannot be known. And that means when there are black boxes and it is not known what is going on inside of them, it is impossible to know if: it is tested, has it turned on, is it behaving the way it was expected. And that

means that productively verifying complex interactions requires understanding of what is happening inside a component, even when it is difficult to understand the insides of the component. So, determining what happened in a simulation without metrics has become very difficult. The things inside the component cannot be seen, the component engineers cannot necessarily see into the big system. In both cases people are operating blind and that tends to mean that you don't know where have you been sufficient, have you achieved the goals of your verification plan.

So, thinking about the traditional metric which would be coverage, coverage metrics are a classic measurement that most people understand where can be looked and see for every feature or every function within an IP, within an SoC, it can be asked has that feature in fact been exercised. It is useful to know if it is never been exercised. That actually tells something very clear: the module was not tested, it is necessary to understand it. But the coverage metrics are fairly broad and imprecise. They do not give an indication about why it was covered, if module was checked nor they do not tell a whole number of things of what was going on when that happened. And as a result, the goal here is to talk about how to move beyond that traditional coverage so it is necessary to have more insight into what happened, what's going on, what has been verified, and know if the wanted goals are met. They can give an idea of what was involved in the verification. It can be anything from simple measurements of what IP was included in an integration, what abstraction level was the IP at, and perhaps what revision was the IP at. So, if a system can be considered, theoretically it may have had an old version of one IP, or it may have had a whole part of the system was at a higher abstraction level simply to get the throughput needed.

Understanding what was in the system and what happened can be very helpful. And that will now help both in terms of was the test successful, when the coverage items that it hit were of interest - were they of interest given that were at some abstraction level. If hit is done a low-level coverage point in one block but it was only connected to abstract models of other blocks, that may not be a valid coverage point. And so, measuring when did we hit what, is a useful thing to have. And if we are looking at the IPs themselves, when an IP is inside an SoC, understanding the environment in which it was tested may be important to understanding the level of coverage that was achieved within that IP. For example, there could be a need to know that an IP was stimulated with particular stimulus, such as particular diagnostic code or software code so we know that it went through that. It would also be interesting in not only having tested the power-aware components of a particular IP, but also knowing about the interactions between the power-aware on one component and how it interacted with all the other components when they're at various levels.

So the goal for metrics is to provide system-level visibility into IP blocks, what was in this system, what IP blocks were there, perhaps to some extent - what were they doing. And vice-versa - for an IP block, how did it act? When it was at a system level - how can it be seen it behave with other IP blocks? Did it meet what the IP designers had in mind? And it's through the correlation of metrics that we can get this kind of information. Any one metric will give you one data point. and coverage, to an extent, fits within that category. It tells you something, but it doesn't tell you the more complex interactions that are happening within an SoC. And so for a complex build process, we will be looking at things - what was instantiated? At what integration level/ Did we get the features in there that we wanted? When we ran a particular kind of stimulus - did the checker that was required to tell us if that stimulus worked - was it in there? Did the coverage actually record the coverage that was important to that particular stimulus? And did all of these play together in the way that we expect? It's metrics that can be used to give us the answers to those kinds of questions.

4.2 Information extracted from metrics

Metrics across the entire design and verification flow are of importance interested, as much as possible. That means, from the point where RTL design is being checked in, to the build, simulation and regression process, across various abstraction levels, and across various aspects of the overall project. That can be a lot of metrics, with many, many points to cover. So, while one wants to be measuring across all of them, it is also important to keep in mind that the metrics need to be only where there is something that really will give you useful information. Are they actionable?

If one cannot answer the question, what would this particular measurement, what needs to be done with this particular measurement, then it is probably not a useful one. So, in this section examination of the kinds of things that one can measure will be considered. And bring up the question, why would it be useful if it is useful.

So, for example, in the figure 4.2 there are some examples of various kinds of metrics. And they go across the design, stimulus, build, debugging and regression range. So, if one looks at the kinds of metrics that are available, they go across the entire range of design, build, simulate, regress and bug tracking environment. And they carry many different pieces of it. From the design, there are likely to be different abstraction levels. Performance may be an issue. Both simulation performance and simulated performance. And the simulation performance is an area where looking at specific metrics into it, to make sure that while going into a large environment, there is no slow down too much in the simulation, or at least slow

down at the minimum of hit things.

The stimulus sources, the types of sources, the effectiveness of the sources, the productivity of the sources be of interest. And the checking. How does one do the checking, what are the results of the checkers, whether if it provides false positives, false negatives things of that sort. Looking at the IP. What are the interface activities? Possibly being able to look internally in some of the key states so it can be considered as an abstracted level of an IP to the SoC verification environment. Coverage environment's obviously going to be important.

What coverage want to be had, at what abstraction level, is any particular coverage point interesting. Having a low-level coverage point when considered to be at a very abstract level of simulation is not likely to be very interesting. So, it is important to make sure that it has been considered correctly. What are the sources of the files? What are the revision numbers of the file? If we ran a particular test and it succeeded, but there is a new revision of a file, out, that may not be of interest, that success may no longer be interesting. It is necessary to know that the newest version of a particular IP in fact passes a particular test. Looking at the run. What simulator was being used? Was it a simulator, was it an emulator, was it an FPGA? Are there differences between those? Looking at the host machines, looking through a farm and farm performance can be of interest.

And within the regression, obviously, which simulation were run. What errors were found? Looking at regression efficiency is likely to be something that one does with metrics. If there is a particular failure in a test, then it is probably unlikely to be of interest to rerun that test until that failure has been fixed.

Metrics can provide some clue at which tests are still of interest, and which tests are waiting for some engineer to look at them first. And then the bug status. And using the bug status as a way of understanding at which point pf project flow the execution is. Where are the areas that are likely to need some more focus more concentration? And how stimulus is done, across the design, and across the project, overall. So, while one can have individual metrics that provide specific information across the design, and across the build, across the bugs, any one particular metric is unlikely to be very interesting. It's really only when the metrics are correlated together that allow to ask the questions that are specific and give broader answers to how the verification is going on.

So, for example, knowing that a particular test passed, may only be useful when knowing which IPs were in there, what were the correct checkers in there, when it is at a sufficiently low abstraction level for that particular test. All together may can confirm this test in fact passed and these coverage metrics are in fact useful for

Process and Focused Area	Process and Focused Area Attributes and the Information Provided by Associated Metrics		
Design	Abstraction level	Performance of the simulation	Instantiated blocks
Checking	Sours of the checkers	Results	Checker abstraction levels
IP	Interface Activity	Key internal states	
Build	Source and rev of files	Initial configuration used	
Coverage	Types of coverages	RTL/ stimulus/ checker/ reference model	Coverage abstraction level
Run	Simulator/ Emulator	Host Machine info	Performance of simulation
Regress	Which simulations	Errors	Errors re-found

Figure 4.2. Example of Various Verification Metrics.

that test. Where any one of those particular points would not have provide that information. As part of the build process, metrics can be useful in a large SoC environment. Particularly if there is a fair amount of churn, within the verification or within the IP themselves. Metrics can provide some insight in what is happening within that build. So, some of the examples there are which IP blocks where put into a particular environment? Occasionally there will be an SoC where some IPs have been left out. That is going to be okay for some tests, but for some other tests that may mean a test passed, but the fact that one of the key things that would have done the stress, provided stress for that environment was not in place. Where did an IP block originate? Where the right version of an IP block is used? There are all the correct pieces to make a particular test valid? For example, knowing which IP blocks were used during the build process will be important. There are all the necessary IPs? If another IP block was necessary in order to provide stress or provide some other side functionality the test may have passed, but the purpose and goal of that test were not met.

Metrics, as a part of the build process can tell which pieces where in, where did they come from, what does our entire design contain. That allow to ask questions about the validity, the stress level, and the completeness of any particular test. Metrics are often considered to be a measure. How often did something happen? Coverage metrics would be an example of that. Where the coverage will say, yes, hit this particular point is done and in fact it was hit seven times. But they need not always be a count of an event. Metrics can be used in a lot of other ways, to simply say, it was at this particular abstraction level. This component was instantiated. This is the revision number of a particular IP block. Or, it took this many wall clocks seconds in order to run this test.

There are a lot of types of metrics, and it is by broadening the scope of what is defined as a metric, that one can ask more interesting questions conceivably. It is true the correlation of event based and non-event-based metrics that can be got a more complete overall picture of what happened in a particular part of the verification environment and the validity of a particular test. For example, maybe it wanted to know that a particular test was run at the correct abstraction level, with the necessary IP blocks, at the latest revision level, and the checker was in there, to make sure that this thing in fact, passed. And then the coverage points inside an IP block are seen. It is through that correlation that can be got a complete view if the goal considered to make the verification test plan is hit.

As part of the simulation process, there are similar goals with metrics. Which provide many individual pieces of information about the simulation. Not only the what happened during the simulation that could not only include coverage, but may actually have more domain specific measurements of any particular state that could have been hit ? a particular correlate wat it hit, distributed state across multiple pieces? Which pieces of the simulation were, in the environment were used, and were they playing together as expected? So, at that point may be is focussed on, sort of, the bandwidth requirements, or maybe on some interface issues. May be is looked at a stimulus and a coverage mechanism and the usage on some internal busses, to answer that question.

Metrics also look at the aspects of the simulation process that may be a field of interest. Anything from the effectiveness of stimulus sources, checking mechanisms. So, from a stimulus source may be is looked at different classes of stimulus that is a field of interest. Be it firmware, be it constrained random, be it graph based, how are they doing for a particular class of coverage that are looking for. If when can be looked at a class of stimulus correlated with a checking mechanism and correlated with coverage to ask about the productivity and effectiveness of any particular piece of the verification environment. This can also really matter when looking at simulation performance start, and simulated performance as well. Both of those are going to be important. Is an IP performing as effectively per clock cycle that would have been expected? And is that IP performing reasonably from a simulation, the simulation throughput is the one expected to have. Simulation metrics can give a great deal of information about all of this.

Large SoC designs are likely to be using a number of sources of, not only stimulus, but also design IP itself. Metrics can provide a fair amount of information of which sources were used and identify the kinds of traffic that was generated and the effectiveness of those particular sources. If a particular source is generating heavy traffic in one area and no traffic at all in the other, it may be interesting

to pull out another source. By looking at how a system was tested, and how it reacted, and understanding what was going on in each piece, which is where the metrics that can be helpful, actually it is possible to measure the productivity of a stimulus method or measure the productivity of checkers. And it is possible to look at how that can be improved. So, this is a place where the balance between where metrics are putted, because it is necessary to understand the broad things, but it is not wanted to have metrics that are just going to take up time and simulation resources that do not tell something. Finding that balance of, where measurements have to completed and where they do not, is something that becomes important. And needs to be looked at architecturally within the verification environment. So, checking metrics, when correlated with other metrics can provide a better understanding of what is happening in our system overall.

In looking at a particular check one may be worried about how frequently is that check happening. The simulation resources are been efficient? How accurate is that check? Can right stimulus be obtained? is it possible to have the right information happening beyond that? Was there a sufficient noise generation? Were there other issues that had to be hit in order to make this test actually be effective? It is only by looking at this combination of metrics which is likely to be understood whether everything was hit and if it was wanted to hit these.

So, where coverage metrics are definitely useful for providing information about very specific spots. So, it is obviously giving, it provides information, but it is not necessarily to have useful information to answer the question, have the verification goals been hit? So, it is only through answering multiple questions here that get the coverage information as desired.

By looking at the stimulus, and the coverage, and the checking mechanism together, there are enough information to say if a particular goal of the verification environment has been met. Looking at domain specific information may now be important as well, and metrics can be helpful. Understanding that the performance characteristics, not just of a particular IP, but of groups of IP working together can be a part of the verification architecture and verification requirements. Metrics can be used to measure domain specific operations across multiple blocks.

For example, if one is worried about multi cache environments, looking at specific points within each IP, such as, TLB performance, cache performance, memory utilization, one can begin to ask very specific questions about what the average best case is, worst case latency for a low and store operation, for example, metrics can provide the abstracted information out of each IP, so it is possible know when any particular piece of an operation occurred, put them together and begin to ask domain specific or system specific information about how did the system act when

it receive a particular test, and was that sufficient. By correlating multiple metrics, it may be possible to ask about bandwidth utilization and head room that you have within an architecture of an environment. That it has only to be seen once it has been integrated by multiple IPs together. What these types of measurements may actually have at a different abstraction levels, with different accuracy, there may be something that at an ESL environment, one can look at these to confirm the getting of the performance wanted. And then as going down an abstraction level, it is possible to say, it can be performed the measuring performance similar to what has been predicted at the higher abstraction levels and it is possible to look for correlations in there, and that may very well be part of the verification goal to see that there is a match as going down an abstraction level. That there are not messed up between the architectural planning and the RTL implementation. Starting with integrating more IPs into larger SoCs, simulation performance becomes more of an issue as well. And the metrics can sometimes help too.

One of the things that is happening, is that, in an SoC there is a very large environment, a whole lot of pieces, it is slow. Usually a lot slower than wanted. One of the things that can happen is that a change in a particular IP may be put in at the IP level and it may be quite a period of time, from the time that change has been checked into an IP to the point where it shows up inside an SoC. Sometimes weeks, sometimes even longer than weeks. And that means that, if something was coded incorrectly or inefficiently it may pass all its tests at the IP level, go through even a multi IP level thing and really only show up as a performance degradation.

Looking at simulation, and simulation configuration, it is a place where metrics are getting fairly well known. The obvious example is, running something with some level of randomness is done, and a particular random seed is picked. The most obvious metric is what was that seed. And there are very few environments these days that do not capture that and store that for a good long time. So that there are the ability to rerun a particular test if it is of interest. Configuration may very well be more than just the random seed, though. Any user switches that were thrown, anything that was run specifically that might have been included on a pro run basis, by the particular person running the test. By reporting the configuration, along with other metrics, and providing the correlation capability, one can look for patterns between coverage, stimulus, test effectiveness and the particular configuration that is happening within each simulation.

The regression process is another place where metrics can be very helpful. Regression environments tend to be very large; they can be between tens and hundreds of thousands of machines that are all running in parallel. They are way outside of the scope of what one is likely to understand and without very specific measurements. This is a place where there are probably already fairly reasonable metrics

to track efficiency and productivity within the regression environment. Two types of regression metrics that are fairly clear are information on the regression run. How fast is each run going, and is a regression run being effective? And also, on the simulation farm. Is the predictive use of the farm obtained? Are regressions going in and out of each simulator within that farm efficiently? Regression run information, may include, the test name, the frequency of tests, the seeds that were used, configuration that was used. Everything that provides some information about a particular regression run. By looking at the tests that provide the most coverage or that are most effective at hitting any particular bugs, looking at test run order, looking at where tests are already finding bugs and should not be rerun. A lot of different mechanisms here to improve the productivity and increase the both simulation farm utilization and the verification productivity overall.

And then, metrics as a part of the overall project. There are a number of non-simulation-based metrics. The most obvious one is bugs. The bug tracking system tends to be outside of the simulation environment, but nonetheless correlation of bug, bug reports, but open rates, with the simulation environment, can be useful for understanding the effectiveness of all parts of a verification environment. Knowing which simulation reported the bug. Knowing about the stimulus sources, the checking mechanisms, the version number, the versions of each IP, the abstraction levels provide some indication of the effectiveness of each of these. At what abstraction level it is possible to still effectively find bugs. How often is needed to go down below TLM level, for example. Or how often is needed a more detailed checker or can be done with a more simplistic checker. Bug tracking is a lovely way to find the hard data of what is actually capturing the RTL errors, and then correlating that back to which pieces of the verification environment have been most effective at finding those bugs.

This is one of the places where can really got some hard data on productivity and effectiveness of verification. Knowing when a bug was closed can also tell some information about project progress. And also, some insight into stability of code, effectiveness of verification environment, effectiveness of the regression runs. So, bringing the non-simulation-based metrics where they are required to be obtained, into the entire metrics environment can be very helpful.

Other project-wide external measurements may include, code check-in rates. Stability, churn rate, things of that sort. If it possible to look at the teams, how big is the team? How much time is the teams spending on this particular project? Where are they geographically? Is there some correlation? Is there a geography that needs some help? Is there a geography that is particularly good at doing something, and can they help everyone else? Simulation farm efficiency can go right back

into correlating the simulation farm with the bug rates. May tell how well is progressing. Determining which metrics wanted to have is going to be dependent on a number of factors. Again, this is the balance of having the metrics give useful information, and yet not having metrics that will simply take up simulation and people resources in building and maintaining them. This is likely to be dependent on a number of factors, such as, the integration level of the project, the corporate culture. How much metrics are elected to be used? How fixed one is in terms of developing stimulus, developing checkers. And the expected lifespan of IPs. If an IP is going to be used for many projects, it would probably be more interesting to get better measurements on that IP. If this IP is going to be used once or twice, and then never again, it is probably less interesting. So, balancing, what is wanted to know with the cost of implementing metrics is a critical part to getting successful metric implementation.

So, metrics provide visibility into all aspects of the build, the stimulus, the coverage, the regression environment, even the design environment. However, choosing metrics that are actionable, that will provide needed information is critical, so it is not necessary to have so many metrics that cannot be moved. And insight of the project really comes from looking at multiple metrics and asking a question if there is something which needs further information, what are the different pieces needed to put together in order to answer the wanted question. Getting that insight requires asking for multiple pieces together, in general. And the measurement from any one part of the project may very well be useful for other areas as well. Understanding bug rate is likely to be useful across multiple places. Understanding something about a particular IP may be useful not just this simulation but even in the IP development. And any particular metric, because of the ability to correlate with others, may be useful for various different places, not just in a simulation run. By understanding something about the IP, that may be useful in the build, it may be useful in regression, may be useful across the life of the IP.

4.3 Addressing the Problem

Successfully adopting metrics, like any other part of the verification process is expensive and requires planning. Recognizing the potential breadth of metrics, putting in the organization necessary for the classification, the reporting and control of metrics is critical to making metrics successful.

Metrics architecture, as with any verification architecture is expensive and requires planning, foresight and care. There can be a large number of metrics and the volume and it can overwhelm to the user. Providing mechanisms to control the execution, the classification and reporting is critical to managing the large volume

of metrics. Because of the volume and because of the distribution of metrics providing an API, providing modularity and providing an identical interface throughout an environment is important to allow block level metrics to be integrated and go up to a higher level and for the higher-level metrics to be useful wherever they may happen to be. Here we will look at some of the requirements of this architecture in order to make metrics successful.

The breadth and volume of metrics needs to be managed. And one of the obvious things to do is provide some level of classification for them. This allows metrics to be enabled, disabled, determined as being relevant or irrelevant, depending on what the goals of a particular simulation, or a particular regression are. There are lots of possible ways of organizing them, this is one example, which would be looking at test specific metrics, user specific metrics or project specific metrics. Again, this is not the way, this is a way to look at doing it. And this may be architecture dependent, may be dependent on the company corporate culture dependent.

Having the categorization that says, this is the goal of my test and here are the kinds of IP's, the kinds of metrics I'm going to want to turn on is likely to be of interest. And the ability to enable, disable and report on specific metrics needs to be considered when looking at a metrics architecture.

Choosing metrics to be able to fill out the metrics that's of value to you is going to be important. So that I know that when I'm looking at tests across my range of what I want to be doing from a verification architecture view point. I want to know that I'm going to have these kinds of metrics in place, so that I have the information to fill out this chart. This really is looking at what metrics are needed at what levels with what control at a verification architectural level. And then following through with the implementation.

If one looks at user specific organization, which would be different from a test specific, users tend to be focused on an area. One user may be mostly within an IP block, or even a piece of an IP block, another user may be not understanding any of the IP's in detail but may be much more interested in cross or maybe doing the SoC integration. So, a user may want to be changing which metrics are relevant, based on what they understand or what they specifically are looking for. This is likely to be focused on specific issues that are looked at. There's a bug, there's something doesn't look right.

There's something where it is needed to dedicate more focus. And being able to enable the metrics in specific areas based on what a particular user on a particular run is interested in can be important. So the user needs to be able to enable metrics that gives them the insight that they need for a particular environment

for a particular build, for a particular test, and be able to say, I need this class of metrics in place in order for me to be able to answer the questions that I'm looking for right now. And so that requires another set of organization that also need to be taken into account.

And then finally, looking at project specific organization. There are going to be a number of metrics that are likely to be of interest from a project, or even if they are running multiple projects in parallel that give you some view of that. So, looking at project specific metrics. This can be one of the things where we catch a, something, a trending kind of thing that went wrong.

Without the metrics one would have been going through tens of thousands of check-ins to ask, what made this happen. So, one way to view metrics is by trying to categorize them. How can we look at any particular metrics and put them into their buckets so that we have a way of saying which ones are we interested in and when.

There can be a set of metrics that are telling us about the key blocks. What's the FIFO utilization rate. What's the average number of entries within the ram that are filled? Something like that. Gives you broad information about how that IP is behaving at a level that somebody not familiar with the IP would still be interested in knowing and able to make use of. And then there are likely to be a whole bunch of detailed metrics within that IP.

Within a complex system verification environment there's likely to have different simulations, different simulators, different abstraction levels. And for each of these one is likely to be asking different kinds of questions.

Looking at the improve regression efficiency. If a re-run only has failures every now and then, it's probably useful to turn off metrics, turn off all kinds of things except for the basic project metrics. Because if you need to know more detail about it, you can go re-run it with things turned on. But there is still probably the need to have the project metrics. How fast did the simulation go, what generally happened? Understanding how often ran regressions are likely to be important no matter what. When a failure has occurred that's when a re-run with a particular metrics may be used just to begin to get some narrowed focus of which part is of interest. So, there may even be an automatic re-run in the regression environment.

In case of a failure; some level metrics have to be turned on.it has to be performed the re-run operation for that. So, when somebody shows up to look at it, it has already been able to provide the information necessary to narrow it down. And finding the trade-off here. When do you turn them on, which do you turn on, what do you do automatically, where do you ask for an engineer to come in and make that choice is going to be something that's possibly in the architecture,

possibly learning through experience. But having the mechanisms in place is likely to be important. And, packaging metrics with IP.

Standardizing category definitions and implementing them with a similar mechanism across the project means that the metrics are likely to have the reuse. In the same way that one tends to want to have both IP and verification code have reuse capabilities. Another part of the standardization is going to be the run time control. It may be important to have metrics that are enabled sometimes, disabled other times, providing a runtime control that's easy to handle is going to be important. So, because metrics can be expensive to run, wanting, and also kind of noisy in reporting, potentially, it's going to be important to have those metrics enabled that you're interested in.

However, the reporting of metrics is going to be important to have standardized in some way. Because there are many metrics in many places. If one metric report one way and another metric reports another way correlation is going to be difficult. Having a standard reporting mechanism, so that it's possible to look at any two metrics within a system, put them together and ask a more interesting question, means that a standard report is going to make this possible. So, not only which metrics are used, but how they are reported is going to be important.

In general, trend analysis is the thing that's going to be the most common use of metrics. So, trends are likely to show progress against the schedule, progress against a particular test. Progress against a particular IP. And, they could be anything from fairly simple, bug open rates, bug close rates, percentage of tests written over time, to more complex tests that show correlations.

When metrics are architected correctly it may give a sufficient rated. But the general coverage results would not provide this kind of information where it says if a particular block of code is hit. was it hit because of the right conditions and did it come out correctly? May not be known. Because some other test may have by accident caused a correct stimulus to generate this particular condition. It caused an event to occur. It was never looking for it, it never checked it, it had no idea why it happened. Coverage will give you a false positive in that case.

Metrics can be useful for giving us other possibly more interesting measurements of productivity per simulation cycle. For instance, rather than simply saying we get this number of cycles per second, is can be asked, have the cycles been used effectively? How many tests are being run through in that number of cycles. How many bugs are we catching? How many cover points do we hit? And look at a test productivity in a different way where we actually measure not simply, it can be run this number of cycles, if 80% of them were idle that's actually not a very interesting

number, but obtaining the kinds of checks and the kind of coverage? It is required in as few cycles as possible, so as to be able improve this specific simulation and productivity in a more intelligent way, potentially.

So, in summary, when looking at metrics, the biggest goal is to bring up the need for organization of metrics, APIs, modularity, standardization of reporting, standardization in categorization. The ability to take metrics, put them in a way where it provides the ability to enable and disable based on the need of a particular simulation, of a particular run, and control them. And the consistency of metrics that allows for metrics in one project to be used in another project, allows for the reuse of metrics. The standardization of reports, so that you can take metrics coming from multiple areas, bring them into one analysis and ask an interesting new question that would otherwise not have been available.

4.4 Aspects needed to adopt Metrics

A methodology is required to implement a metrics driven process. As with any part of verification, planning and architecture are critical to success. It is imperative to be able to have a flexible methodology that will allow this project to go and then allow for reuse of the metrics and hopefully other parts of the verification environment as well. Making the metrics part of the overall verification architecture is important. however, any part of verification is expensive and having metrics be a built-in piece at, just like any other piece, is important to the success.

Effective implementation of metrics must be architected to fit the project and the environment. And possibly multiple projects in the future for reuse considerations. Several things to look at before the implementation includes the type of metrics and what is going to be involved in getting them to work successfully. Several things need to be thought about before implementing metrics.

In the architecture, which include the type of metrics, what they are going to be measuring, how they are going to be built, and how they are going to do each of the pieces so that they are consistent, modular and reusable. Project independent metrics are one way of categorization of metrics where, an attention is paid to the metrics that go beyond projects, and the ones that are focused on any of the individual project. So, looking at project independent is likely to include things such as simulation farm, the languages and tools that are used across the company, things of that sort.

In looking at the metrics themselves, implementation rules, the languages that are used, and the mechanisms to control when the metrics are active is going to

be important, and reporting rules. How metrics are report, and where that goes in the standardization. So, when considering the multiple metrics across different areas is going to be important there as well. And finally, because there are so many metrics looking at how queries are stored, how the metric reports are stored and what those mechanisms are going to be important. There will be a lot of data coming out, and looking at how, where there will be the need to put for the project independent metrics is important. Because it won't be on the end of the project presumably, they'll be somewhere else, but they still are going to be across projects.

Another categorization might be environment specific metrics. Where it will be important looking at aspects that are still above a particular project but they are very involved in each project such as simulation farms, the run time, the kinds of things that a project requires in order to run a simulation or regressions. Moving into project specific metrics, these may include architectural types of metrics. Looking at the key busses, the processes, and possibly the top architectural issues within each complex IP to measure correctness, accuracy, performance, equivalence to architectural models, to make sure that things are working as they were expected to, as they were originally architected to.

And down to more bus specific, or functional specific metrics that are looking at functional coverage, possibly combinations of stimulus and coverage and abstraction levels to ask, has the performance been reached yet, has the utilization and the coverage types of things that we expect to hit have been hit? So, looking at how these metrics useful can be performed. Ensuring a consistent solution is important to making sure that they can be controlled and making sure that the analysis can be achieved across metrics. Consistency is important for both visibility and efficiency. Categorization, enabling, disabling, mechanisms to say what you want, based on the category, based on a classification of some sort. It's likely to require multiple categories. And that may not be accurate. A metric may well fit in five or six categories. And if it is somehow wanted a particular category needed that metric, It may be involved in both the low level and in a high level. It may be involved in understanding both through put on an API and some high-level things. So, looking at those categories, overlap is okay if it can be known why it's there and what it's going to indicate.

Providing similar levels and quantity of reports increases its likelihood of being useful. So, if one block is spewing every little detail about every little nand gate in there, and another block says that we started the simulation and we finished the simulation. Something is likely to be amiss. And finding the right balance for what one is trying to do in a simulation, and the kinds and quantity of reports that are coming out, is important. Because that means consistency in enabling, consistency in categorization, consistency in reporting, all together allow you to get a, the right

quantity and right quality of metrics so that you can get a single picture and ask intelligent questions that you wanted to ask when you first ran the simulation.

So, one other consideration is legacy IP. If you are like just about everyone else, you have IP blocks that came before this particular SoC project is being developed and it may have come 10 years before, it may have come in the last project. But in either case, it does not have the metrics inside of it, it does not follow the guidelines that you are trying to set up in terms of metrics reporting, and enabling and so on and so forth, and chances are that IP is not well understood. So the goal there is to be able to add sufficient metrics that you can understand your SoC and work with your SoC without disrupting that IP that is functional but you have no support for.

One of the ways of doing that may be to add just IP's around the boundaries so you can see when something went in to the IP and provide some correlation mechanism so that when you see when it came out, you say, it went in to here, out to there. That may be enough to ask performance questions. It may be enough to let you track how your system overall is performing without modifying that IP block itself. As with metrics throughout, the goal here is to have metrics that are determined at the architectural phase. What do we need to measure, including for the legacy IP, and what do we need to put, possibly just around that legacy IP so that you have consistency, in the API, in the enable and disable mechanism and reporting mechanism, for all IP's in your SoC, including the legacy IP.

4.5 Benefits obtained by adopting the Metrics

To be successful, Metrics need to be quantitative and actionable. In any large SoC environment no one person has a feel for the entire project. There tend to be gut feelings, there are impressions. Once you're in a larger SoC project, there's enough of the environment that no person sees, that it's very difficult to use that sort of intuitive approach in order to figure out how to improve productivity.

So Metrics are a place where there can be a quantitative measure that gives you a view across a project, across a test, across an environment, wherever you need to be looking, that let you know what is the state of where we are, and what in particular looks like it might be an area where we can improve.

Metrics can be effective for on the fly detection, where they can show project inefficiencies or test inefficiencies, or perhaps entire category inefficiencies where, without having a measurement it'd be very difficult to know that there was something going on. It may be weeks before a small change ripples into another environment. It may be weeks before a particular stimulus source and a particular

check source are actually linked together. Being able to track down how did these small changes fit into this bigger environment can show you basic inefficiencies. And, they give you a quantitative view. It is an actual quantitative measure of how good something is. What you measure may or may not be correct depending on how you ask the question. But it does give you an assessment of IP quality, IP efficiency. And component quality, and even overall quality of your system. Giving you a view in the system means you have the ability to actually first ask about productivity. Everyone wants to improve productivity, but before you can improve it, you need to know, where are you. You need a base line, and then the ability to make a query. And then in a reasonable amount of time do another query and ask, are we better, are we worse. So you actually have a way of going forward in productivity improvement.

By having Metrics that are available from the lowest level IP's up into an SoC, up into how an environment overall is running, means you have the ability to make that measurement of productivity and actually determine when you decide to make an architectural change or some other change to improve your productivity. What should it be, why do you think it'll be effective, and, later, make a measurement and say was it effective, and if so, should we do more, should we do less, should we go the right way, what have you.

In summary, Metrics provide you a quantitative feedback of what has become in any criticized SoC an arbitrarily complex environment. Any one group of people are no longer even able to understand how an environment overall works together. Each IP is likely to require a group of 10 or 15 people.

Multiple IP's, the people integrating together can only understand at a very high level what's happening. So giving a quantitative feedback of here's what's happening in a system that is so large that nobody understands it, Metrics can do that for you. It's that range of view from a single simulation from the view of a single engineer up to tell me how my multi-site distributed geography environment is running efficiently and how can I improve its efficiency. It's that range of bottom to top, through, we'll bring this up again, because it's so important, consistency, an API, in modularity, in reporting, in storage, letting you go and ask questions that you weren't able to ask before.

As long as you keep either original data or some merge of original data years later you can look across the trends of your projects and say, what can we do as we go forward to get better at this. This is what Metrics can provide you, the ability to ask across scope, across time, across project, across environment, the kinds of questions you want to be answering. As long as you have your original data. Either full original data or some level of merged original data you can go back in years, you

can go back in projects, you can track how an IP, or how your SoC projects have been doing over time so you can look at, where do we improve our productivity of verification.

Chapter 5

Main Verification Blocks

In this chapter the main blocks in the verification environment will be illustrated, and in order to get easier understand the role of each block, a simple adder will be taken as a Modul Under Verification. Before dealing in various block for verification, an introduction of SystemVerilog (SV) is necessary because nowadays SV is the host of the most important methodology for verification[14].

5.1 SystemVerilog for Verification

Originally, there was RTL options, Verilog and VHDL, and these 2 languages kind of battled it out as to who would be used the most for RTL. It was never conclusively decided, it had a lot to do with the industry and the geography that there are in as to which you chose. In both cases, SystemVerilog has taken over as the verification language that would work for either of these, and that is because SystemVerilog Is Object Oriented, it includes Functional Coverage, it allows constrained Randomization and it allows to have Methodology Libraries such as the UVM. All of these do not really exist for Verilog or VHDL in the broad industry sense [15].

The difference in philosophy between VHDL and SystemVerilog is that VHDL is more of one of those great big contracts that outlines everything that is going to happen, all the behaviours that are going to happen in a model. And so, in VHDL there are resolution functions, standard library and everything that needs to be defined in order to simulate hardware written explicitly in a language.

SystemVerilog is more of a letter of intent, it says, we know we're using hardware, we know we're going to have 4 states, we know that you understand how the simulator works and so we are going to shorten up our communication in a much shorter and more concise language because we both kind of know what is going on. The difference is that SystemVerilog can be a lot more concise than VHDL but

then you have to know more about what the simulator is doing behind the scenes in order to really see how the language is working.

As most of engineer used VHDL it is important to show the differences between it and SystemVerilog. For this reason, an example of a simple adder is taken to be designed in both languages. The adder has two 8-bit inputs and an 8-bit output with a carry, it is got a clock and a reset. And in output there are the 8-bit for the sum and the signal for curry as it can be seen in the figure 5.1.

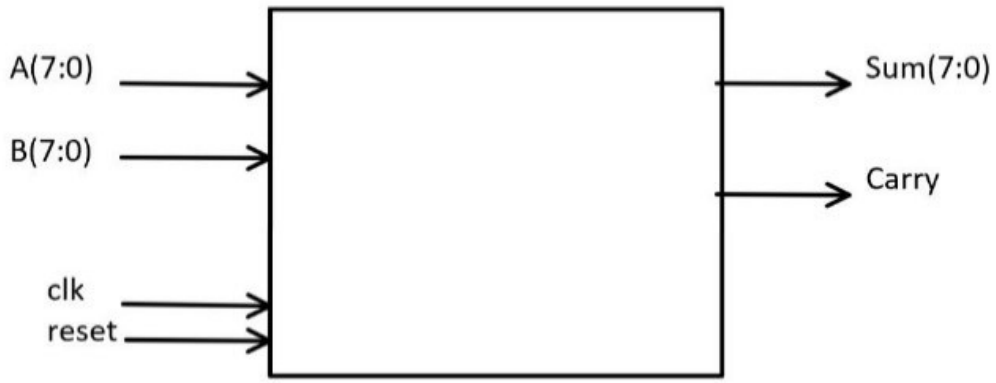

Figure 5.1. Scheme of an 8-bit Adder.

5.2 VHDL Adder

In VHDL a lot of definition must be done. For example, one has to declare ieee library at beginning, and then use the standard logic library, the standard logic arithmetic library and UNSIGNED library. After that the entity for this adder is declared which is to say, here are the inputs and outputs for the adder. Then the implementation of the behaviour is to be done. Signals inside is defined and there that's 9-bits wide, and then within the actual adder logic itself you can see that we have to, for example, concatenate a zero in front of the A and the B to create two 9-bit numbers, add the 9-bit numbers and then pull off from the 9-bit internal number the "carry" and the "sum" and write them out. In the figure 5.2 a code of an adder can be seen.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.STD_LOGIC_UNSIGNED.all;

ENTITY adder IS
    PORT(
        A       :  IN    std_logic_vector ( 7 DOWNTO 0 );
        B       :  IN    std_logic_vector ( 7 DOWNTO 0 );
        clk     :  IN    std_logic;
        reset   :  IN    std_logic;
        carry   :  OUT   std_logic;
            sum     :  OUT     std_logic_vector ( 7 DOWNTO 0 )
        );
    END adder ;

ARCHITECTURE rtl OF adder IS
    signal sum_int : std_logic_vector (8 downto 0);
    signal A8      : std_logic_vector (8 downto 0);
    signal B8      : std_logic_vector (8 downto 0);

BEGIN
        A8  <= "0" & A;
        B8  <= "0" & B;
        sum_int  <= A8 + B8;

    adder: process (clk, reset)
    begin
    if reset = '0' then
        carry <= '0';
        sum   <= "00000000";
    elsif clk'event and clk = '1' then
        carry <= sum_int(8);
        sum   <= sum_int(7 downto 0);
    end if;
    end process adder;
END ARCHITECTURE rtl;
```

Figure 5.2. VHDL code of a simple Adder.

5.3 SystemVerilog Adder

In the figure 5.3 a SystemVerilog code of and an adder is seen. Notice that the interface has the entity architecture combined. 8-bit inputs, an 8-bit output and

a carry signal are defined. And then the math is done, especially at the bottom, A+B gets put into "carry" concatenated with "sum". Two 8-bit numbers get added to create a 9-bit number essentially and that gets put into "carry" and the "sum".

SystemVerilog assumes the engineer knows how the simulator will interpret code, and what is really going on in the code is the + sign always outputs 32 bits, A+B , a 32-bit output is got which has 9 bits of value. That 32 bits of data gets put into the 9-bits that you get when you concatenate the "carry" and the "sum", the top bits get thrown away and the output goes out. All of this is things that are in the simulator that one has to know how the simulator works. You need that information to fully understand how this model works and that is the difference between VHDL and SystemVerilog.

```
module adder
    ( input [7:0] A,
    input [7:0] B,
    input clk,
    input reset,
    output reg [7:0] sum,
    output reg carry);

    always @(posedge clk or negedge reset)
    if (!reset)
        {carry,sum} <= 0;
    else
        {carry,sum} <= A + B;
endmodule
```

Figure 5.3. SystemVerilog code of a simple Adder.

Another thing to note is: SystemVerilog is case sensitive. So Clk is not CLK is not clk. You have to take that into account when you write code. In VHDL if you want to wait for the positive edge of a clock your process has a sensitivity list and that sensitivity list waits for the clock and then it checks to see if this is the rising edge of that clock. SystemVerilog sensitivity list actually has the word "posedge" that says, wait until the positive edge of this clock and then do your thing. And this @ (posedge sig), @ (negedge sig) is unique to SystemVerilog.

SystemVerilog has 4-state values built into it. So, there is no libraries, no standard libraries, we have 1, 0, X, Z. In VHDL usually when you are using standard logic your constants may be a string of 1s and 0s. SystemVerilog does not use strings that way. It has a syntax for defining constants. With that syntax you tell it how

wide is the constant, the radix of the constant and the numerals in that constant. SystemVerilog has 2 kinds of types. It has four state types and two state types. Four state types such as logic and reg, which is the same as logic, integer and time. These hold 0 and 1, X and Z, so these are the 4 states. Two state types such as int (32 bits), shortint (16 bits), longint (64 bits), byte (8 bits) and bit, these only hold 1s and 0s. You can convert between them. Pretty much, if you put four state types. Into two state types, the Xs and Zs become 0s, only the 1s go through as 1s. Naturally, if you go from two state to four state, the 1s and 0s are defined on both sides. Notice that the four state types are unsigned by default, whereas two state types are signed by default, so a byte has an MSB that is the sign bit in a two-state type. You have to say "unsign byte" if you want to get an unsigned 8 bits.

5.4 Main Verification Block

In order to do the verification, there are 2 main things: the design and the verification environment. Our goal is to check the functionality of the device (the adder for example), So, we will stimulate it. To do this we construct various blocks to allow us to verify the model and to be able to adapt the verification environment to other models in the future.

5.4.1 The Model Under Verification (MUV)

This block contains the code of the module to be verified. The design can be described in VHDL, Verilog or SystemVerilog.

5.4.2 The Sequencer or Generator

This block has the task of generating bits sequences or data that can be applied to the MUV. To generate the stimuli, the generator uses the randomize function which is available in SV, to randomize the transaction class. Is to consider the method to transfer the data and send them to the driver, for example mailbox can be used in the case of the adder.

5.4.3 The Driver

Since the sequencer only generates the data without thinking about transmitting it to the MUV, it is necessary to have a block that takes the generated data and sends it to the MUV. The driver then brings the data items to the interface signal which in turn interacts directly with the MUV.
These things are done in the driver:
- Transfer packages from generator to MUV.

- Check the signal enable of input.
- Wait for the end of the operation and it is when the generator stops sending transactions (in the case of UVM this operation is done automatically by the UVM API, so it helps the designer to not think about this detail).

5.4.4 The Monitor

Look at the communication between the MUV and the driver and try to evaluate the response received from the MUV. It also has the task of sampling the inputs and outputs of the MUV. Then it send a prediction of a result that can be accepted to another block called scoreboard.

5.4.5 The Scoreboard

It has the task of receiving the data sampled by the monitor and checking whether they are correct or not. It contains the part responsible for generating the answers accepted through an ideal model. Check if the answer received from the monitor is the same that generates by the ideal model, then if it is the same it indicates that the result is correct otherwise it indicates the error.

5.4.6 Transaction or Data item class

In the verification context the input of the MUV is represented by the transaction. However, in this block it is needed to declare the necessary fields for stimulus to be generated. And in order to create a coverage as high as possible, a SV constraint propriety is exploited to randomize the data items.

5.4.7 The Interface class

Since the development of the MUV is done separately from the development of the testbench, it is necessary to have another block to link these two. So, the interface is used as a connection point between the monitor, the driver, the MUV and the testbench. And in this block, all the MUV signals are grouped.

5.4.8 The Environment class

In this block the Generator, the Driver and the Monitor must be placed and connected to the Scoreboard.

5.4.9 The Test class

In this block the environment is created, the testbench is configured. for example, in the case of an adder, what and how much data item the generator must generate is indicated in this block. Therefore, the stimulus driving is initialized. In the event of a change in the type of data to avoid bringing future modifications in the Generator, driver, monitor and scoreboard, the Generator is connected to the sequence within the test block, so the changing in the code will be only in such block.

5.4.10 TestBench_Top

The testbench includes the MUV, the Test and the interface instances. In this block the MUV is connected to the testbench via the interface. Once all these blocks are built, the simulation can be run. So, the testbench generates inputs in random way. after that, the inputs generated must be sent to the MUV.
The Monitor listen to the communication and try to make a prediction of the correct result. Then the Scoreboard compare the results from the MUV and the prediction of the Monitor and check if the tow results are identical. If they identical the Scoreboard send a message in output in order to indicate the correctness of the functionality of the model otherwise it indicates the error.

5.5 Case of Study

5.5.1 The importance of various blocks in verification

In this section one model will be considered as old project and another one will be considered as a new project. For simplicity simple models will be taken. The first project is an adder and the second one is a multiplexer. The first block that differ between the 2 projects is the MUV. In the figure 5.4 there is the code in SystemVerilog of a simple adder of 8 bit. In figure 5.5 there is the code in VHDL that describe a simple multiplexer of 2 bits. The multiplexer is described in VHDL in order to verify if the verification environment in SV can be used also with the design in VHDL.

In the verification environment there is a little change in the code in order to adopt it to the new project (the mux). The signals of the MUV are grouped in the Interface block, so, it can be changed here as can be shown in the figure 5.6, notice that a signal for selection is added and the size of input and output signal is changed. These changes should be don also in the transaction class.

```
module sommatore(
  input            clk  ,
  input            valid,
  input            reset,
  input  [7:0] a        ,
  input  [7:0] b        ,
  output [8:0] c        );

  reg [8:0] tmp_c;

  //Reset
  always @(posedge reset)
    tmp_c <= 0;

  // Waddition operation
  always @(posedge clk)
    if (valid)    tmp_c <= a + b;

  assign c = tmp_c;

endmodule
```

Figure 5.4. 8-bit Adder in SystemVerilog.

Next changes should be in the scoreboard in order to describe the behaviour of the ideal model, as can be seen in the figure 5.7 and 5.8.

However, in the Test-bench class there are the MUV instances, and the interface signals are connected to the MUV ports, so, the modification is to be ported into the Test-bench. The same procedure can be prosecuted with other projects like multipliers, register or projects more complex.

5.5.2 Running the code

The simulator used to do the verification is Questa® which support several languages such VHDL, SystemVerilog and UVM. High performance and capacity simulation can be combined by Questa with unified advanced debug and functional coverage capabilities. The first step to be done is compiling the code and here the first bug may be can appear, after having the code compiled correctly the next step is the simulation. Between these 2 steps may be some setting have to be done such as enabling the code coverage. At the end of simulation, the code can be run. In the figure 5.9 a piece of result that appear in the screen when running the code for the Adder and multiplexer is illustrated. The driver prints the result from

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux is
  Generic (N :integer := 2);
port (
  clk :in std_logic;
reset :in std_logic;
valid :in std_logic;

  a : In std_logic_vector(N-1 downto 0);
  b : In std_logic_vector(N-1 downto 0);
  s : In std_logic;

  c: OUT std_logic_vector(N-1 downto 0));
END ENTITY;

architecture beh of mux is
signal x: std_logic;
signal z: std_logic;
signal m: std_logic;
signal tmp_c : std_logic_vector(N-1 downto 0);
begin
p1: process(clk,reset)
begin
x <= valid and s ;
m <= not s;
z <= valid and m;
 if reset= '1' then
     tmp_c <= (others => '0');

 elsif rising_edge(clk) then
    if valid = '1' and s = '1' then
      tmp_c <= b;
    elsif  valid = '1' and s = '0' then
      tmp_c <= a;

end if;
end if;
end process;
 c <= tmp_c;
end beh;
```

Figure 5.5. 2-bit Multiplexer in VHDL.

the model under verification and the monitor print the prediction of the expected result, the scoreboard compare the 2 results and in these 2 cases it was identical, so, scoreboard print the message of the correctness of result.

The correctness of the functionality of the model can be seen on the screen in the Transcript section without going to look at the waves. This method consist to save time, in the case of study 2 variables were added in order to indicate the number of stimulus generated and the number of errors founded, notice that it is possible to look at the last packet generated to see if there were errors.

```
interface intf(input logic clk,reset);

  //declaring the signals
  logic       valid;
  logic [7:0] a;
  logic [7:0] b;
  logic [8:0] c;

endinterface
```

```
interface intf(input logic clk,reset);

  //declaring the signals
  logic       valid;
logic       s;
logic [1:0] a;
  logic [1:0] b;
  logic [1:0] c;

endinterface
```

Figure 5.6. The Interface class of the Adder and the Interface class of the Multiplexer.

```
//Compares the Actual result with the expected result
task main;
  transaction trans;
  forever begin
    mon2scb.get(trans);
      if((trans.a+trans.b) == trans.c)
        $display("Result is as Expected and errors = %0d, packet is %0d, ",x,y++);
      else
        $error("Wrong Result.\n\tExpeced: %0d Actual: %0d and x = %0d",(trans.a+trans.b),trans.c,x++);
      no_transactions++;
    trans.display("[ Scoreboard ]");
  end
endtask
```

Figure 5.7. The part in the Adder Scoreboard that compare the result of Monitor and the MUV.

```
 //Compares the Actual result with the expected result
 task main;
   transaction trans;
   forever begin
     mon2scb.get(trans);

if((trans.s ? trans.b : trans.a)  == trans.c)
        $display("Result is as Expected and x = %0d, count is %0d, ",x,y++);
      else
        $error("Wrong Result.\n\tExpeced: %0d Actual: %0d and x = %0d ",(trans.s ? trans.b : trans.a),trans.c,x++);
      no_transactions++;
    trans.display("[ Scoreboard ]");
  end
 endtask
```

Figure 5.8. The part in the Multiplexer Scoreboard that compare the result of Monitor and the MUV.

```
# ------------------------
# ------------------------
# - [ Driver ]
# ------------------------
# - a = 219, b = 39
# - c = 258
# ------------------------
# ------------------------
# - [ Monitor ]
# ------------------------
# - a = 219, b = 39
# - c = 258
# ------------------------
# Result is as Expected and errors = 0, packet is 255,
# ------------------------
# - [ Scoreboard ]
# ------------------------
# - a = 219, b = 39
# - c = 258
```

```
# - [ Driver ]
# ------------------------
# - a = 3, b = 2, s = 1
# - c = 2
# ------------------------
# ------------------------
# - [ Monitor ]
# ------------------------
# - a = 3, b = 2, s = 1
# - c = 2
# ------------------------
# Result is as Expected and errors = 0, packet is 15,
# ------------------------
# - [ Scoreboard ]
# ------------------------
# - a = 3, b = 2, s = 1
# - c = 2
# ------------------------
```

Figure 5.9. A piece of result after running an Adder (on the left side) and a Multiplexer (on the right side).

Chapter 6

Verification By UVM

6.1 Introduction to UVM

In this chapter a brief overview of the architecture of a UVM testbench will be given and then an introduction to some of the concepts of UVM. After that a technical introduction technical introduction to the details of UVM coding will be given step by step.

UVM is the Universal Verification Methodology, that is what UVM stands for. It is a mechanism for describing testbenches in SystemVerilog, for designs that are either in SystemVerilog or Verilog, or even VHDL or SystemC designs. It is an Accellera standard, it is based on the work that Mentor did along with Cadence to develop the OVM, the Open Verification Methodology [16]. The success of OVM caused a groundswell of support in the industry and we brought other vendors and users together under Accellera to develop the UVM. The UVM is a SystemVerilog base class library, this is the first standard that ships source code along with the documentation. So that source code is shipped in open source under the Apache license, and it is nearly backward compatible with OVM. So, if one are familiar with OVM, a lot of the concepts will seem familiar.

Some of the highlights of UVM is it enables constrained random, coverage-driven verification, that is kind of its reason for being. It allows to put together configurable and flexible testbenches and it is really focused on Verification IP reuse. SystemVerilog is a large language, there are lots of different ways of doing certain things and the aim of using UVM is to focus the efforts and create freedom from choice.

If everybody does things the same way, that is easy to take one piece of verification IP and replace it with another. The idea of this goes along with the separation of concern, so the notion of a test from a testbench is separated, transaction-level

communication is used, so that are talking between components at a very high level of abstraction and it removes some of the details that make it harder to read these components from one environment to another. The stimulus itself is sequential and randomizable, but it is also layered, so one can have sequences of sequences and a way of coordinating the operation across different interfaces.

There is a standardized messaging system and there is also in UVM a register layer, a way of specifying stimulus and response at the register transaction layer and also for modeling the registers that are in the design, so it is possible to make sure that the values of the registers that can be expected to be there are actually there.

The aim of this chapter basically is put together a system that allows to build a verification environment without having to reinvent the wheel every time. So, one can be able to reuse components from project to project, it is desired to be able to reuse pieces of the environment as going from the block level up to the system level, and all of the things in UVM are geared towards that goal.

Talking about constrained random verification. The idea of constrained random is twofold - the constraints are there to make sure the randomization allows to find unexpected bugs. Particularly when there are stimulus coming in on multiple interfaces, the randomness of the behavior across those two interfaces is more likely to uncover things in the design that one had not initially thought about.

If we thought about them, we would be able to code around them or fix them initially, so the value of having random stimulus, particularly when you have multiple interacting interfaces, allows us to find those unexpected bugs. It also allows us to automate the stimulus generation, so by specifying it in a constrained random manner you write one description of your stimulus and the tools allow you then to take advantage of automation to generate multiple scenarios from that. So, when you rerun your simulation with a different random seed, you will get a different set of random values still subject to the constraints so they will still be legal, but it will create different sets of transactions that go into your design and particularly different interactions between the stimulus occurring on different interfaces.

So, we send the stimulus into the model under verification, the model under verification operates on it in some way and sends the values out and now we need to understand exactly what happened, because remember we do not know exactly what the input stimulus is, therefore we do not know exactly what the result is, so we have to build an environment that will be self-checking. So, we create checkers that look at the inputs and the outputs and have some notion of a reference model or some other way of determining correctness, because that question, "does

it work?", is one of the most important questions that we need to answer in verification as seen in chapter 4.

The other question we have to understand though, is "are we done?" If we are creating random stimulus, we need some notion that we have actually covered all of the different aspects of the design that need to be covered in order to say that we have actually tested everything.

So, we start with our verification plan, from that we develop a functional coverage model, a set of checkers, a set of cover points in SystemVerilog, assertions, whatever you are more comfortable with and we track that to make sure that all of the randomness that we have created actually has reached all of the specific points in the model under verification that we want to be able to cover, everything from "have I actually exercised all of the aspects of my protocol on the bus?", to "have I actually hit every transition in my state machine?", UVM allows you to build an environment that will take advantage of constrained random to uncover things that you had not necessarily predicted but also to be able to answer these two questions. Once we understand what our coverage is, we can then modify our constraints to increase our coverage.

So, if we have packets coming in with headers and payloads and checksums we may initially start with a small sized payload or something and once we have reached coverage on all of those, then we can modify our constraints, so the payloads will start becoming larger. We may expand the range of values allowable in our header. So, as we modify those constraints it allows us then to increase our coverage and target more aspects of the problem that we want to look at that we defined in our verification plan. So, one of the ways that we do this is we separate this idea of a test from a testbench.

The test is responsible for defining exactly what is going to happen for this particular simulation run. The reusable verification environment is there to define all the components needed to interact with the MUV through the interfaces. Once the reusable verification environment is defined, the test's job is to specify individual things, everything from configuration values to how many times a particular set of stimuli should run, to what specific sequence will going to be run, to what version of a particular component may there is in the environment, those kinds of things.

The environment itself is highly configurable, highly reusable and then the tests are there to define the differences from one run to the next of what is done in that verification environment. So, it is possible to have multiple tests all using the same verification environment, all setting up different things in that verification environment. The most basic things that one is going to change are which sequences may

run to generate different sets of stimulus and perhaps what coverage information must be collected that corresponds to that sequence of what one is going to look for.

Taking advantage of the randomization in there it is possible to look at things where those interfaces interact in the MUV and uncover things that one had not necessarily thought about. So, the tests themselves become relatively straightforward and very targeted at just the specific things which needed to modify in that environment to get the appropriate things to happen for this individual simulation run.

The stimulus itself is there to drive transactions into the MUV. So, a transaction is an encapsulation of whatever information needed to communicate from one device to another. So, in a bus-based system a transaction may be address data read/write, in a network system it might be a packet with header, payload, whatever. So that transaction gets sent to the driver, the driver is a component in UVM whose job it is to talk to the MUV at the signal level. So, the transaction is taken, which is the way to think about the problem, a packet must be sent into the MUV, the driver then takes that transaction information and turns it into pin wiggles at the MUV. Those transactions themselves are defined as sequences, so a sequence is a specification of a chain of transactions, and that can be a reactive chain as well based on what the response is from the previous transaction, it could enable you to create a different transaction the next time.

Sequences themselves can be nested or layered, so one can have a sequence of sequences, we refer to this as a virtual sequence whose job it is to coordinate the operation of other sequences, and those sequences can be run in parallel or sequentially. It allows to create as complex a set of scenarios as you may need and the randomization again of what is going on takes care of figuring out the next transaction to generate based on the current state of the system and we can ensure maximum flexibility in the set of transactions that we generate. So, if we look at the big picture of what a verification environment is, we have the MUV, a set of verification components whose job it is to communicate with the MUV, these verification components are called agents and inside an agent there are three specific components.

There is a sequencer whose job it is to execute and arbitrate across multiple sequences that may want to communicate to the MUV. That communication happens again through the driver, so the transactions from the sequencer go to the driver, the driver communicates at the pin level to the MUV. And a monitor looks at that same pin level interface, recognizes the pin level activity as transactions and communicates those transactions out to the rest of the environment.

In the environment, in addition to these agents there will be things like scoreboards, coverage collectors, or other analysis components, perhaps other verification components, other agents that are talking to the MUV. And then to coordinate the activity of all these things we may have a virtual sequence that is part of the test. So, the test's job is to configure that environment to specify what the coverage model may be, what sequences we want to run, what virtual sequence we might want to run to coordinate the activity of other individual sequences that might be running in the agents. And there can be multiple tests, so you can have a library of tests that are all dedicated to a particular reusable verification environment and we can modify the behavior of that environment through a configuration database.

So, there is a piece of UVM called the config database that has a set of name-value pairs, things like 'this sequence to be of this particular type', 'I want to configure that particular sequence to run a certain number of repetitions', 'I may want to configure my driver to inject errors with a certain frequency', things like that. So, there is a mechanism built into UVM to allow that test to configure the environment is set up to be configured in such a way that it is flexible enough for the test to be able to tweak it as you need to specify whatever specific scenarios you want to have happen for a given simulation run. The UVM is shipped not only as a set of source code but also as HTML documentation.

This is actually the official standard. This is the documentation that says these are all of the classes and these are their interfaces. So, theoretically anybody could come up with another implementation of UVM. As long as it met this interface specification it would be consistent with UVM.

UVM is based on SystemVerilog. It is a base class library which takes advantage of the object-oriented programming and TLM capabilities in SystemVerilog, so it is necessary to understand how classes work in SystemVerilog.

6.2 Case of Study

In this section there is an actual runnable example of an adder in UVM to provide confidence moving forward that we kind of know what is going on.

Starting out with a MUV which is a module. The UVM verification environment consists of a fixed part and a variable part. The fixed part is the UVM environment itself which includes all the components needed for communicating with the MUV, and the variable part is the test that configures the environment to do whatever it is needed for this particular case.

The UVM environment is all class-based, it is implemented through SystemVerilog classes. The communication is through a SystemVerilog interface, so the interface and the MUV module themselves are structural, and all of this is instantiated in a top-level module. Once we have this understanding, let's move forward and look at what each of these individual pieces actually looks like.

The interface is a standard SystemVerilog interface, we declare it interface dut_if() and it includes in it whatever signals are necessary for actually communicating with the MUV. The MUV itself is a module and the port connection to that module is the MUV interface itself. And then in the top-level module, we instantiate the MUV interface and the MUV, and we connect up the MUV interface to the MUV. In the environment, we extend the environment from a base class called uvm_env and we use this macro, the uvm_components_utils macro to register our environment class with the UVM infrastructure.

Whenever you declare a component you use the uvm_component_utils macro and notice there is no semicolon at the end of that line. We declare the constructor for the environment and constructors in UVM for every component have two arguments: there is a string name argument and a parent argument that is itself a UVM component. And the only thing we do in the constructor is we call super.new and pass in the name and the parent. This is consistent across all UVM components, and it is just something that you will get used to. Then we have the build_phase method, so this is a function, the argument is something we call a phase of type uvm_phase and in the build_phase method this is where we instantiate all the components that are in our environment: our agents, our scoreboards, our coverage collectors.

And then in the environment there is a run_phase; The run_phase is the only task based phase in UVM. All of the phase methods, whether they are functions or tasks, take this uvm_phase argument so we know which phase we are in and inside of the run_phase is where we do the interesting things.

So, in order to control the test, to be able to start and stop things, we use what we call objections. We have to raise at least one objection at time zero, so in the run_phase we will raise the objection and raise_objection is a method of the phase argument, so we are going to raise an objection in the run_phase here, and then we do whatever it is we need to do, in this case we are just going to wait for 10 time unit, and then we drop the objection when we are done. So, when all of the objections that were initially raised are dropped, that is when the test ends. In the test itself, there is a component, it is extended from the uvm_test base class so we register it using the uvm_component_utils macro, and then we declare the environment. So, we use the _h notation to indicate a handle to the environment.

And then, just like any other component in the test class, we declare the constructor with the name and the parent argument, and there is a build_phase and in that build_phase is where we actually instantiate the environment. We do that through a method called create. We assign to the environment handle the result of this create call.

The double colons and type ID and things. This is part of the UVM infrastructure, and what it allows us to do is to create an instance of the environment type, but to do it in such a way that we have flexibility to override the type later if we want to. So, we use the create method, so instead of calling the constructor we call the create method which will in turn call the constructor. We call this a wrapper pattern in object oriented programming.

You just need to understand the first piece is the type of the component, type_id is an internal element of that component, and then the create is a static method of the type_id type. We say create and then we give it the two arguments, the name and the parent. Then we can put all of this stuff in a package, so in my_pkg we need to include the uvm_macros.svh file, this is part of the UVM distribution. We import the uvm_pkg, using the star notation and then we can include other files.

This gives us the ability now to just compile this package and we have included it and imported all the UVM stuff that will then be recognized by the environment and the test. To instantiate this in a top-level module, we need to import the uvm_pkg and we also import our pkg. This now gives us access to everything that we have declared. And then all we need to do in the top-level module is in an initial block we call run_test and we give it as an argument the string name of the test type.

The run_test method will actually instantiate the test component and start the phases executing which will cause the test to run. When we run the simulation, we call vlog on the file that does all of the package imports and all that. The top level module is called top, so we invoke vsim on that. And then when we do that, it loads all of the packages including uvm_pkg and our pkg and the top level module, and the interface and the MUV. And then it runs, and you see a standard header from the UVM package, and then what happens is we get some information messages printed out. So, the first thing that we will get printed out at time 0 is from the reporter, which is part of the UVM infrastructure, and it just indicates that it is running our test.

When that test is complete at time 1800, we issue a message that says, from this file uvm_objection.svh at line 1267, at time 1800, the reporter is issuing a test done message, which says the run phase is ready to proceed to the extract phase

which means that we are done with the test.

And then at the end of the simulation we issue a report summary, so it shows how many different types of messages were issued. In the figure 6.1 we can see the result after running the simulation of the adder.

```
# ----------------------------------------------------------------
# UVM-1.1d
# (C) 2007-2013 Mentor Graphics Corporation
# (C) 2007-2013 Cadence Design Systems, Inc.
# (C) 2006-2013 Synopsys, Inc.
# (C) 2011-2013 Cypress Semiconductor Corp.
# ----------------------------------------------------------------
#
#
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(215) @ 0: reporter [Questa UVM] QUESTA_UVM-1.2.3
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(216) @ 0: reporter [Questa UVM]  questa_uvm::init(+struct)
# UVM_INFO @ 0: reporter [RNTST] Running test simpleadder_test...
# UVM_INFO /home/thesis/SystemVerilog/UVM_adder/simpleadder_scoreboard.sv(51) @ 140: uvm_test_top.sa_env.sa_sb [compare] Test: Correct!
# UVM_INFO /home/thesis/SystemVerilog/UVM_adder/simpleadder_scoreboard.sv(51) @ 260: uvm_test_top.sa_env.sa_sb [compare] Test: Correct!
# UVM_INFO /home/thesis/SystemVerilog/UVM_adder/simpleadder_scoreboard.sv(51) @ 380: uvm_test_top.sa_env.sa_sb [compare] Test: Correct!
# UVM_INFO /home/thesis/SystemVerilog/UVM_adder/simpleadder_scoreboard.sv(51) @ 500: uvm_test_top.sa_env.sa_sb [compare] Test: Correct!
# UVM_INFO /home/thesis/SystemVerilog/UVM_adder/simpleadder_scoreboard.sv(51) @ 620: uvm_test_top.sa_env.sa_sb [compare] Test: Correct!
# UVM_INFO /home/thesis/SystemVerilog/UVM_adder/simpleadder_scoreboard.sv(51) @ 740: uvm_test_top.sa_env.sa_sb [compare] Test: Correct!
# UVM_INFO /home/thesis/SystemVerilog/UVM_adder/simpleadder_scoreboard.sv(51) @ 860: uvm_test_top.sa_env.sa_sb [compare] Test: Correct!
# UVM_INFO /home/thesis/SystemVerilog/UVM_adder/simpleadder_scoreboard.sv(51) @ 980: uvm_test_top.sa_env.sa_sb [compare] Test: Correct!
# UVM_INFO /home/thesis/SystemVerilog/UVM_adder/simpleadder_scoreboard.sv(51) @ 1100: uvm_test_top.sa_env.sa_sb [compare] Test: Correct!
# UVM_INFO /home/thesis/SystemVerilog/UVM_adder/simpleadder_scoreboard.sv(51) @ 1220: uvm_test_top.sa_env.sa_sb [compare] Test: Correct!
# UVM_INFO /home/thesis/SystemVerilog/UVM_adder/simpleadder_scoreboard.sv(51) @ 1340: uvm_test_top.sa_env.sa_sb [compare] Test: Correct!
# UVM_INFO /home/thesis/SystemVerilog/UVM_adder/simpleadder_scoreboard.sv(51) @ 1460: uvm_test_top.sa_env.sa_sb [compare] Test: Correct!
# UVM_INFO /home/thesis/SystemVerilog/UVM_adder/simpleadder_scoreboard.sv(51) @ 1580: uvm_test_top.sa_env.sa_sb [compare] Test: Correct!
# UVM_INFO /home/thesis/SystemVerilog/UVM_adder/simpleadder_scoreboard.sv(51) @ 1700: uvm_test_top.sa_env.sa_sb [compare] Test: Correct!
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 1800: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   18
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [compare]    14
# ** Note: $finish    : /software/europractice-release-2019/mentor/questa10.7c/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 1800 ns  Iteration: 61  Instance: /simpleadder_tb_top
# 1
# Break in Task uvm_pkg/uvm_root::run_test at /software/europractice-release-2019/mentor/questa10.7c/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh line 430
```

Figure 6.1. The Output of The Simulation of the Adder.

6.2.1 Connecting the Environment to the MUV

The MUV is a module, is the structural part, that is connected to our class-based environment made up of the uvm_test which is the variable part, its job is to configure the fixed part, the UVM environment. This are both classes and the environment include in it all the components that we need to communicate with the MUV.

UVM agent is a protocol specific component that is responsible for connecting through the interface, to the MUV and communicating with it at the signal level. All of this is included in the top-level module. The MUV itself has a port connection of type MUV_if. Inside of the MUV it will refer to signals inside of the interface. The MUV may just be a wrapper, that uses the interface as the port and connects to another MUV inside of it that has signal level port connections.

The way that we connect the classes in the environment to the MUV is though this interface. We use what is called a virtual interface, in SystemVerilog a virtual interface is a pointer to an actual interface object. The way that we communicate it is using the configuration database which is a series of name value pairs, so, from the top-level module, we pass that virtual interface into the configuration database with a name, the test will then get that virtual interface out of the config database, put it in what we refer to as a configuration object, pass that configuration object down to the UVM environment which will extract that virtual interface and any other configuration information it may need, and it will pass the virtual interface down to the agent component that is responsible for actually communicating at the signal level, to the MUV, through the actual MUV interface in the top-level module.

In our top-level module we instantiate the interface and in the initial block we issue a command called uvm_config_db::set, so set is a static method of the uvm_config_db object, the config database is parameterized by the type of the value that we are going to be configuring, in this case a virtual interface of type dut_if. The arguments to the set call are a prefix and a path, so we put this together to create a hierarchical path to the object that is going to be getting the value out, in this case it becomes uvm_test_top which is always the name of the top-level test in UVM. Then we specify the name value pair, so the name in this case is called dut_vi when the test looks for something called dut_vi it will get the value, or in this case a pointer to dut_if1 in our top-level module. After that we call run_test which will create an instance of the our test class and inside of the test it will go and get this value from the configuration database and then execute the test.

In that test we extend from the uvm_test base class we have a configuration object of type my_dut_config we will call it dut_config_0, and inside the build phase of the our test class we construct an instance of dut_config_0 and then we call the get method of the uvm_config database. This is a static method of the config database, it is parameterized by the type of the value we are looking for, in this case virtual dut_if, and the arguments to the get method are a prefix and a path, so the prefix in this case is "this" which is UVM test top, because this is the top level test and then there is nothing after that, so we put the prefix and the path together and those specify a scope that we match against the prefix and the path of the set call, then we look for the field name and when we get that value we will put it.

We do other MUV configuration settings here and then from the test, we are going to call uvm_config_db::set. The parameterization is of a our dut_config object and so we are going to pass into it the dut_config_0 we use the star *, so any element that is looking for something called dut_config will see the value of dut_config_0 which includes in it the dut_vi pointer and any other configuration

settings that we may have done.

Now, inside of the agent, there is a component we call a driver. This is the thing that connects specifically to the MUV, it has in it a virtual dut_if that we will call dut_vi and we will just use _vi to identificate that as a virtual interface. This is a component in UVM, it has the same constructor and the same build_phase. In the run_phase, we will raise an objection and then we can access the pins through the virtual interface, so dut_vi.data allows us to set the values for that, and then when we are done with that, we will drop the objection to end the test.

So, when we set and get configuration values, the confid_db is parameterized by the type of value that we are setting and all the values inside of the config database are grouped by scope. The scope is made up of the path, that is a combination of this pointer and the path name that we specified. On the set we use this and some path that we are going to specify, which is typically a relative path from whatever component is doing the set, and then on the get, the scope is made up of this, which is the hierarchical path to that component, and the path is typically a null string, because we don't want to go anywhere below us.

You can create a longer path using this and some other string, if you want to actually configure something below you in the hierarchy, we don't recommend doing that. When you call set, you use this and whatever the relative path is from the component that is doing the set. And on get you use this and the null string to specify that you are getting it. And then the name and the value pairs match up. The path and field names can contain wildcards using glob pattern matching.

In any particular scope we look for a specific name-value pair and we set and get the values. Config information in the config_db actually flows top-down, so the higher level will override the lower level. At the uvm_env level we do a set, using the star * so anybody that is looking for data is going to get object b. And then in the test we are doing a set, again using the star * to anybody that is looking for data, and we are passing in the a object. Down in the uvm_component, when it does the get, because the test overrides the environment, the value that it will actually get is a for that object.

To summarize, we have an interface that we use at the top-level module to connect to our MUV. We create a virtual interface, that is a pointer to that interface object, pass it into the configuration database from the top level module and then the test gets that virtual interface object from the config database, passes it down to the environment, the environment passes it to the agent and the agent uses that pointer to the MUV interface to communicate at the signal level to the MUV.

6.2.2 Connecting Components

We look at the UVM class hierarchy, we start with uvm_object, uvm_report_object is extended from uvm_object and it includes infrastructure which allows us to report messages out of UVM. uvm_component extends from uvm_report_object and that is the base component type used. There are several specific components that are extended from uvm_component that allow us to communicate what role this particular component is intended to play in our setup. Whether it is a uvm_env, a uvm_test, an agent.

Some of these have some distinguishing features in them, some of them are just shells that extend from uvm_component to communicate that intent to you but they are still useful. A component extended from uvm_agent will be created, we use the uvm_component_utils macro to register the our agent type with the UVM infrastructure.

Then we have to declare the pointers to the subcomponents, like a sequencer, a driver and a monitor. Inside of the agent we have the constructor, which every component in UVM uses a string name argument and a uvm_component parent argument and simply calls super.new. And then we have additional methods build_phase and connect_phase and these are the ones that we use to instantiate some components in build phase and connect them in the connect phase. So, in the build_phase rather than constructing directly the sequencer, driver and monitor, we use what we call factory methods.

So, for every component that we create we have an instance name and a parent, that allows us to build a hierarchical name for every instance of every component in the system. The reason that we use the factory methods is it allows the type of object that is actually created to be overwritten in the tests. When it comes to connecting the components together, we use the connect_phase for doing that. The connect method connects together these child components by calling the connect method in the port and export of the components we want to connect. Our driver has something called the seq_item_port, and our sequencer has something called the seq_item_export and the connect method hooks those two things together.

You can think of this as kind of a fancy version of ports in Verilog and VHDL, but the effect is the same. So we have the driver connected to the sequencer, and they will communicate through this seq_item_port. Then we have the phase methods. Then there is the build_phase method which is called top-down so as every component gets created its build_phase gets called. And the build_phase is used to create subcomponents and then once those are created in build_phase then their build_phase methods are called, so we call build_phase in a top-down

manner. Then we have the connect_phase where we connect together the ports and exports of all of these components.

Then we have a function called start_of_simulation which allows us to do some extra stuff before simulation starts, things like opening files or printing up the topology of what you have in your environment. And then lastly, we have run_phase which is the only task that we have in UVM. There are subphases of run_phase. Lastly, we have the report_phase so at the end of run_phase we can go and gather information about the system and report it out, so that you can see actually what happened.

In any UVM component, you have some basic things in it, that every component is going to have, you extend from uvm_component or one of the other uvm_component extensions. You register with the factory using the component_utils macro, then you have potentially some external interfaces, you have some internal component handles that you have to create, and then you have the standard phase methods, so there is the constructor, there is the build_phase where you instantiate components, the connect_phase where you actually connect the components together, start_of_simulation allows you to double check things, open files or whatever before you actually start your simulation, run_phase is kind of the guts of the simulation, and then at the end of run_phase you have check_phase and report_phase. There is another one called extract_phase so it is extract, check and report. And you can use those to go around and gather information about what happened, the current state of your system, whether FIFOs are empty or whatever. Check results and then report out what happened.

In summary, we have in our environment, which is itself a UVM component, we have other kinds of components. The agent is a typical grouping of sequencer, driver and monitor which are other types of components. In the build_phase we instantiate child components. In the connect_phase we connect them, and then in run_phase that is where everything interesting happens in the test. It is possible also having multiple instances of these agents. Agents typically are protocol specific and there is one agent for each interface to your MUV.

6.2.3 Producing the Transactions from a Sequencer and consume them in a Driver

In the agent setup there are a sequencer and a driver. The driver's job is to communicate at the pin level to the MUV, so the driver wiggles the pins, but what we really want in UVM is to communicate things at the transaction level. So, we use transactions as kind of abstract packets of information.

There is just a bunch of methods calls in a transaction that allow us to do this communication and what we are doing is abstracting the information that the driver needs to control the pin wiggles so that we can focus on the problem at the level that we are really thinking about it in terms of read and write transactions or whatever.

In our agent we have our sequencer and our driver, the driver talks through pin wiggles to the MUV and the sequencer's job is to allow the sequence to generate the transactions and communicate those transactions to the driver. So, we have transaction level communication from the sequencer to the driver and then the driver turns that into pin level communication to the MUV.

6.2.4 UVM Reporting

UVM includes a built-in messaging system to allow you to specify messages of a particular severity so that you can see better what's going on inside of your system. So, in its simplest form it is simply a call to a function to display a message. In this particular case we are using a macro, as we will see the macros provide some additional information about where the function was actually called. So, the severity is specified by the macro that we call, in this case 'info' for an informational message.

There is also warning, error, and fatal. The first argument to the messaging macros is the originator or the ID. This could be the name of the company that's supplying the IP that's providing the message, it could be the initials of the engineer who created it, it could be whatever it is that you need to indicate some useful information about how this message was created.

The message itself is simply a string. It can either be a hard-coded string or it can be the result of a sformat call or some other function that produces a string, so, you can embed variable information in it as well. And then for info messages, there is also a verbosity which allows you to filter certain messages based on how important they are to you. When you call UVM_INFO from within your code, the simulation will produce a message where the first piece of information is the severity, then it will show you the file and the line number, and this is information that the use of the macro provides, the time at which it was called, the hierarchical path to the component that actually called it, and then we replicate the ID so you can actually see where the message came from and also the message itself.

So, this gives us a standard mechanism for getting useful information out of the system in a regular way so that we can then go look at these messages and understand what they mean.

The macros themselves consist of uvm_info, warning, error and fatal. These are pretty standard severity levels. The uvm_info macro, in addition to the id and the message which they all take as arguments, has the verbosity argument. These can all be called either from UVM components, from UVM sequences or also from SystemVerilog modules. They will all create messages that indicate to you what the information is and where in the system it was called from. So, the verbosity is controlled for uvm_info messages by specifying the third argument, and the value is specified using an enum.

UVM_NONE means that the message will always appear, and then you can add filtering information, so UVM_LOW, MEDIUM, HIGH and FULL. So the way that it works is from the command line you specify the verbosity, in this case UVM_LOW, and the message will be output if the verbosity is less than or equal to the verbosity level you set on the command line. You can also specify this in your code, but then to change the verbosity you have to recompile.

The messages themselves have the verbosity level associated with them, so, you can control the filtering. If you have information that is really necessary only in very isolated cases when there is something really complicated going on, you can specify UVM_FULL as the verbosity for that particular message and then if you need to see it you can run a simulation where you set +UVM_VERBOSITY to UVM_FULL and then you'll see all of the messages that could be printed out, and you can filter those out by setting verbosity to be somewhat lower.

Typically we use UVM_LOW for regular messages, and then if you really want to just run with nothing, you can use UVM_NONE on the command line and then no messages will be printed out. Errors, warnings and fatals can't be filtered out in this way. In addition to setting the verbosity, there are other actions that you can take. From uvm_top you can set report severity action, so that means that for a given severity you will take a particular action, and 'hier' suffix means that this will apply to everything in the hierarchy from uvm_top on down, it means everything in your UVM system. So, what is this saying is that for UVM info messages, the action that we will take is UVM_NO_ACTION which means to suppress all of the messages.

So, all of the messages that get created get passed into what we call the report handler in UVM and these methods actually tell the report handler how to process those messages, in this case to just suppress the message and not pass it on. You can also control the action based on the ID. So, if we use the set_report_id_action_hier we can specify the ID, and in this case the action UVM_NO_ACTION will suppress all messages with the ID equal to "mg". Some common actions that you can take:

UVM_NO_ACTION will do nothing, i.e. suppress the message. UVM_DISPLAY will send the report to standard output, that's the default. UVM_LOG will send the report to a file. UVM_COUNT will stop the simulation when the max count is reached. This is the default for a UVM error along with displaying the message.

UVM_EXIT will finish the simulation immediately, which is the default for a fatal, and UVM_STOP will call stop. So, to redirect reports to a file we can open a file using the fopen call, so we will specify a writable file called "my.log". Then from uvm_top we will set_report_default_file_hier to use that file. So that means that's a hierarchical call so everything from uvm_top on down will use that file for its default messaging. Then we will set_report_severity_action_hier to say that every UVM_INFO will display to standard out using UVM_DISPLAY, and will also output to the log file.

Each of the actions is represented by a bit mask, so, you can bitwise order them together and in this case UVM_DISPLAY and UVM_LOG will both happen, it will display to standard out and put it in the log file that we just created. The hierarchical methods, because they rely on the hierarchy must be called after the build phase, so doing them in start of simulation is a good idea.

Chapter 7

Conclusion

UVM is targeted to allow you to create reusable verification environments and reusable verification IP where components communicate at the transaction level to allow you to communicate with your MUV and send stimulus in, and get responses back.

By separating the tests from the testbench and making the testbench configurable, the tests become targeted to configure your testbench to specify the differences from one simulation to another, what sequences you are going to generate, what coverage you might want to collect, what specific components you might want to override, things like that.

So, you can have a library of tests that all use the same reusable verification environment where that verification environment is made up of a series of components from a package or library where you pick out the specific components that you want for your particular test and then run it.

If we take a closer look at a typical environment, you'll see that inside of the agent, we have a sequencer communicating to a driver. The sequencer is responsible for generating the stimulus, sending it to the driver, the driver converts that into pin wiggles to the MUV, the monitor recognizes those transactions and communicates them out through the analysis port up through the agent to the rest of the environment.

So, the subscribers are things like scoreboards or coverage collectors, most of the communication is at the transaction level in the environment, and the provides a useful level of abstraction, so we can interact with components at the level that we think about the problem.

We can think about reads and writes, we can think about sending packets. Whatever the typical transaction level communication is for our particular application, that's represented by the sequence items and the transactions that we've

talked about.

All these components are reusable because they all have similar interfaces, everything from the constructor arguments to the use model for connecting them to other similar components, and that gives us additional flexibility. The stimulus itself can be layered, so you can have sequences that generate transactions and you can have sequences of sequences, and you can also have what we call virtual sequences which will enable you to control the activity of multiple sequences in your simulation.

Bibliography

[1] BDTi staff (2010) BDTi certified™ results for the AutoESL autopilot high-level synthesis tool. http://www.bdti.com/Resources/BenchmarkResults/HLSTCP/AutoPilot. Accessed 30 March 2011

[2] Kumar, J.; Pixley, C., "Logic and functional verification in a commercial semiconductor environment", Int. Conference on Application of Concurrency to System Design, pp. 8–15, 1998

[3] Ernesto Sánchez, Matteo Sonza Reorda, and Giovanni Squillero. Efficient techniques for automatic verification-oriented test set optimization. volume 34, pages 93–109, Mar 2006

[4] F. Corno, E. Sanchez, M. S. Reorda, and G. Squillero. Automatic test program generation: a case study. volume 21, pages 102–109, Mar 2004

[5] https://it.wikipedia.org/wiki/Verifica_formale

[6] VSI Alliance, "Taxonomy of Functional Verification for Virtual Component", www.vsi.org, Jan. 2001

[7] www.verplex.com/product.html, 2001

[8] Glenn, Scott et al, "Functional Design Verification for the PowerPC 601 microprocessor", 12th IEEE VLSI Test Symposium, pp. 334-339, 1994

[9] J. Bergeron, Writing Testbenches: Functional Verification of HDL Models. Kluwer Academic Publishers, January 2000

[10] H. Foster, Applied Assertion-Based Verification: An Industry Perspective, Foundations and Trends® in Electron Design Automation, vol 3, no 1, pp 1-95, 2009

[11] Dr F. Brooks, Essence and Accidents of Software Engineering, Information Processing. 1986

[12] https://blogs.mentor.com/verificationhorizons/blog/tag/debug/

[13] Evans, James R; Lindsay, William M (1993), "5: Total Quality Management", The Management and Control of Quality (4 ed.)

[14] IEEE, Standard 1800-2012 for SystemVerilog Hardware Design and Verification Language, New York, NJ: IEEE, 2012

[15] P. Flake, "Why SystemVerilog?," in FDL, Paris, 2013

[16] https://www.mentor.com/products/fv/uvm