Azrieli College of Engineering Jerusalem | עזריאלי מכללה אקדמית להנדסה ירושלים | Valens First in HDBaseT | Verisense בס"ד

Faculty of Software Engineering

Final Project 2016/17

Functional Verification Environment

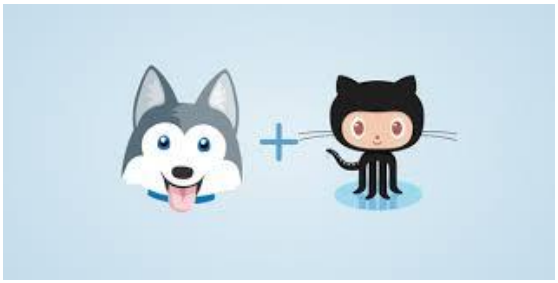
Written By:
Batzion Nissenboim

Academic tutor: **Dr. Miriam Allalouf**	Approval:	Date:
Industrial supervisor: **Zvi Marks**	Approval:	Date:
Projects coordinator: **Dr. Miriam Allalouf**		
Dr. Reuven Yagel	Approval:	Date:

Managing project system

#	System	Location
1	Code	Due to a confidentiality agreement I cannot add a link to my code so I attached appendix with snippets of code and explanations at the code appendix.
2	Diary	https://trello.com/b/8eX2deFS/final-project-management-board#

*I cannot attach a link to the whole code of my project because I'm doing my project at work.



Glossary

Term	Definition
HDBT	High Definition Base Twisted pair
HDMI	High Definition Multimedia Interface
KVM	**K**eyboard, **V**ideo and **M**ouse
HW	Hardware
SW	Software
RTL	**R**egister **T**ransfer **L**evel
DUT	**D**esign **U**nder **T**est
EVC	**E**nvironment **V**erification **C**omponent
BFM	**B**us **F**unctional **M**odel

Contents

1. Introduction

This project presents a verification environment for a part of a chip developed by Valens, the company where I am working as a contractor on behalf of Verisense.

Valens are the inventor of HDBaseT Technology, the world leaders in HDBaseT technology and a leading providers of semiconductor products for the distribution of an uncompressed HD multimedia content.

Let me explain about the HDBaseT technology.

1.1 HDBaseT

We have so many cables, just cables, cables, cables everywhere.

Figure 1: Cables

The chip of Valens developed the HDBaseT technology, the new standard of digital connectivity, all in one ultimate cable!

Figure 2: One cable

The HDBaseT technology transmits and receives uncompressed HD (High definition) video, audio, Ethernet, power and control signals through a single 100m/328ft LAN cable for point-to-point connectivity, for providing home networking, digital signage, video cameras and hospitality applications.

That is to say - 5 play over a single cable, as shown in the figure 3.

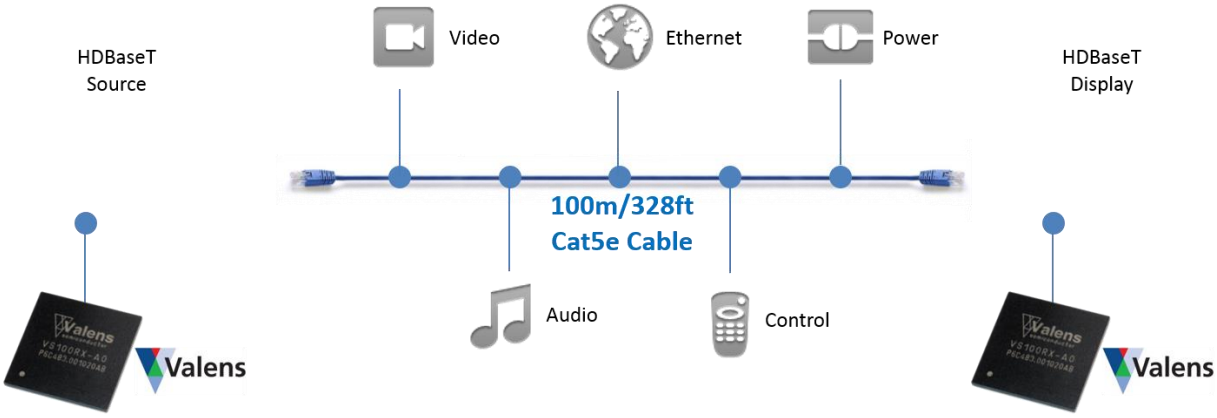


Figure 3: 5 play on a single cable

1.2 Current Valens development

Currently Valens are developing the VS3000 – First Chip to support the new HDBT standard 3.0, with improvements of the algorithm and new components.

1.2.1 VS3000 – Uses

The following figures present some of the use cases for the VS3000 chip.

- Ordinary HDBaseT link with the extenders – USB, HDMI, and KVM:

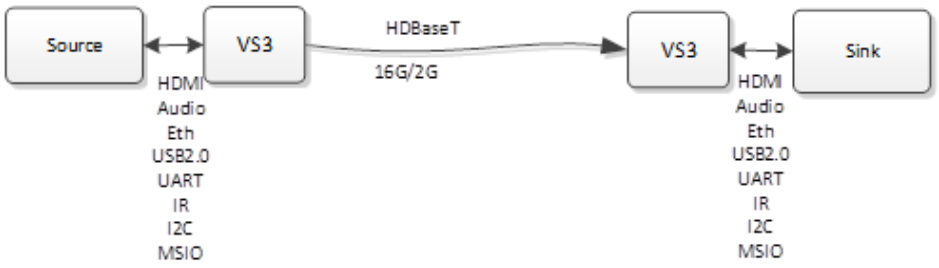


Figure 4: HDBaseT link

KVM is a protocol to transfer the keyboard, video and the mouse controls over long distance.
For example: The ordinary link allowing KVM transfer from point to point, and all other devices need their own cables.
When using it over HDBaseT link in a meeting room 50 meters in length, a computer located in one corner can be connected with the video projector or the mouse on the table at the center of the room with only one cable!

- Daisy Chain – On VS3000 there is a component calls DHDI that enable data transfer between two directly connected VS3000s.

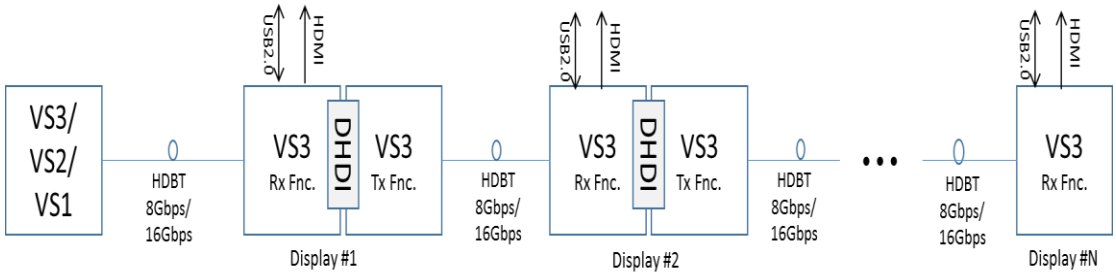


Figure 5: Daisy chain link

An example for the using the daisy chain link is shown here.

- Distributing Audio and Video link:

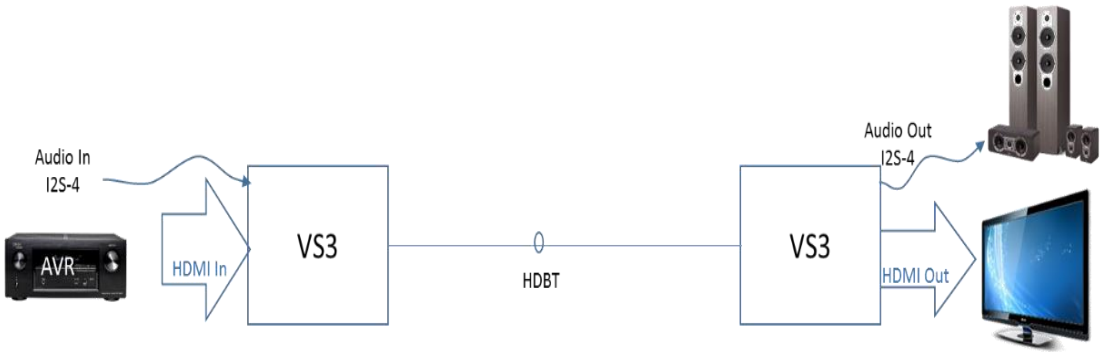


Figure 6: Audio/Video link

- KVM for long distances:

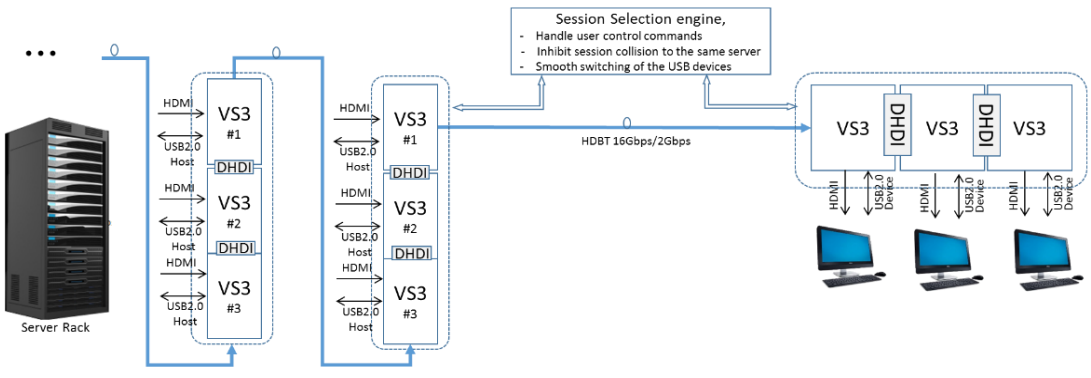
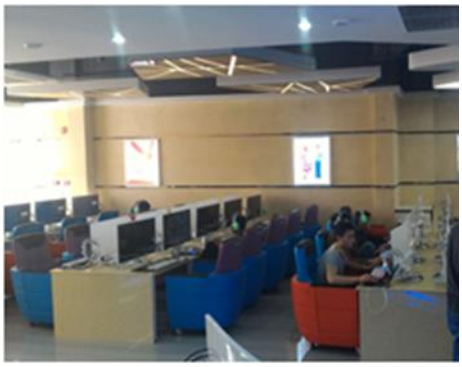


Figure 7: KVM link

Here is a different aspect of the KVM possibility. This is aspect is about networking. This network-like connectivity allows the communication between racks of servers located in a back room with **only** screens, mousses and keyboards located in front office by using only one cable!
It also allows networking between two users in the same room like in gaming, internet café (see below).



- Virtual Reality:

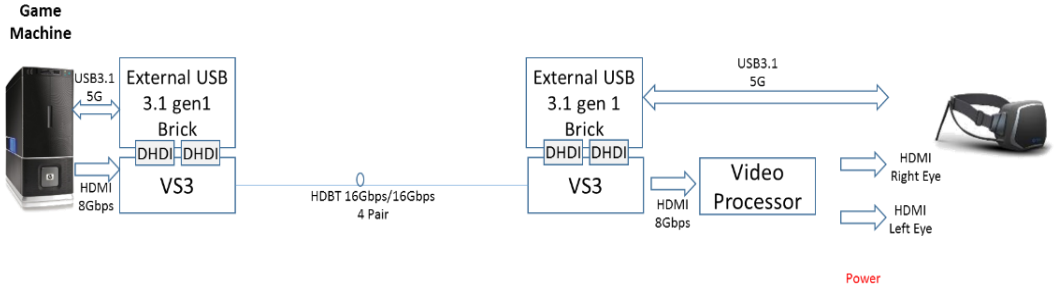


Figure 8: VR link

An example for the using the VR link is GEAR VR glasses for gaming, which became so popular recently.

- Regular network link:

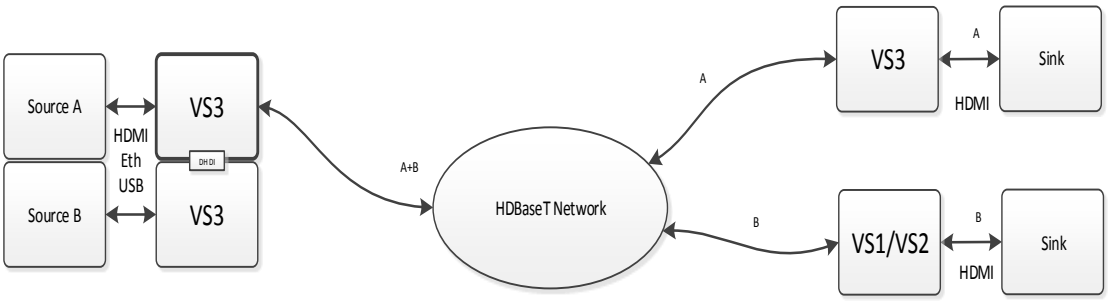


Figure 9: Network link

1.2.2 VS3000 and this project

The VS3000 chip was first planned as a two chip solutions name: Hermon and Snir, and recently merged to one.

My project is to develop a verification environment for a component, called **CpuPkt_Tadp** which is located in the Hermon chips.

The goal of this component (or block as it is called in hardware terms) in the chip is to provide another option to transfer packets of general information between chips rather than the regular traffic path. This block comes as a backup in cases of programming errors, missed holes or bugs that cause failures or uncertain behavior through the regular path in the chip.

The goal of my project is to build the CpuPkt_Tadp environment from scratch, on **block level**, which means that the project will treat CpuPkt_Tadp as a standalone component with no relevance to the whole chip.

Block Diagram

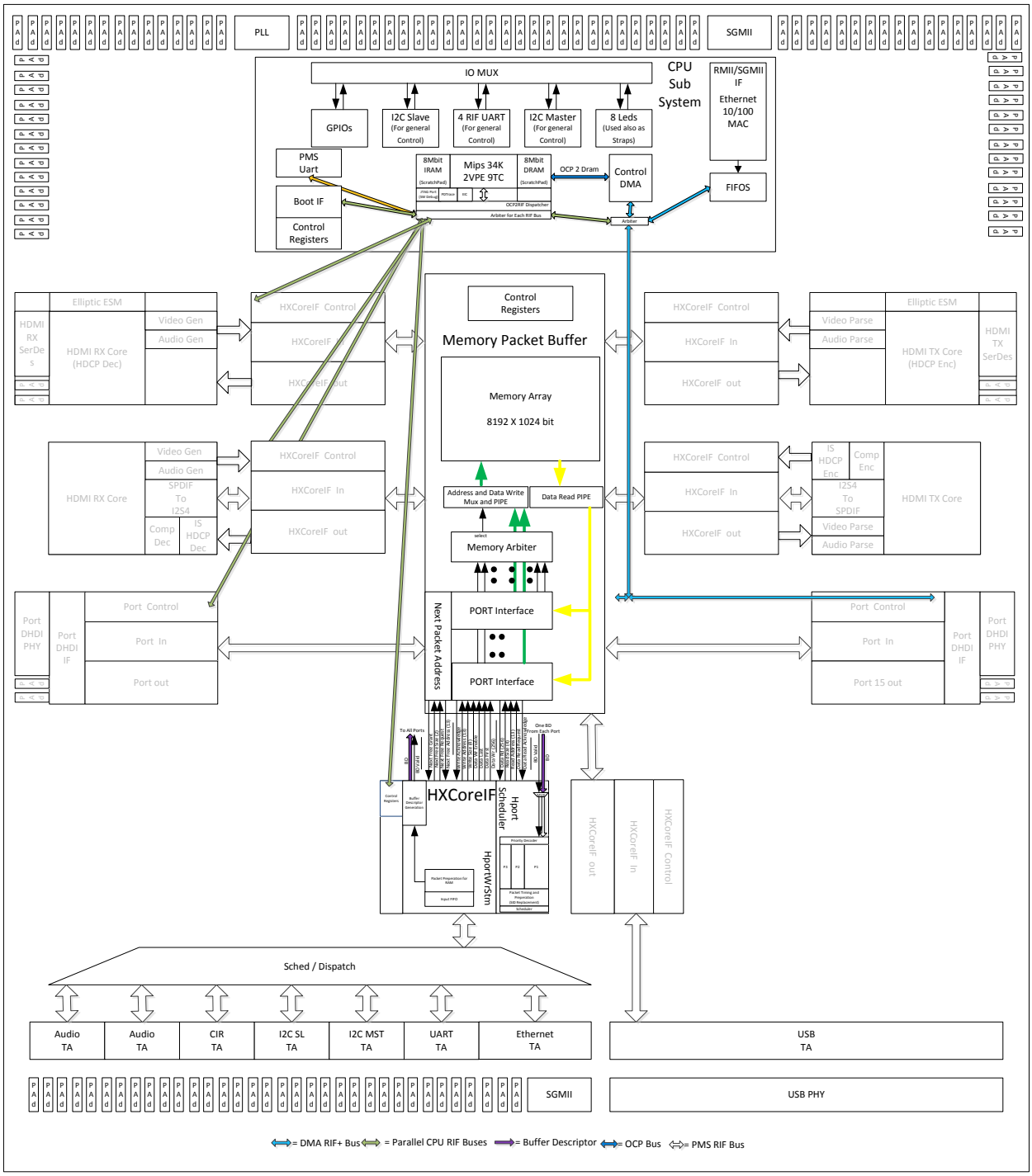


Figure 10: VS3000 block diagram

2. Problem Description

My project will provide functional verification environment for the CpuPkt_Tadp block.
What is a functional verification, and why is it necessary?

2.1 Functional Verification

A functional verification is a process used to demonstrate, prove and simulate the functional correctness of the implementation of hardware's design, namely, a design of a chip.

Another definition of functional verification is - Ensure the design meets its functional intent (a pre-silicon stage), or more simply, as **Harry Foster, Chief Scientist Verification of Mentor Graphics (The Verification Academy)** defines it: *"Verification is trying to answer the question: ' Did we build the system right ' "* [1].

There are two kinds of chips: ASIC (**Application-specific integrated circuit**) and FPGA (**Field-programmable gate arrays**), each requires a verification process.

The process of designing an ASIC or FPGA chip is composed of several stages:

1. Concept
   The need or the new idea for developing the current chip.

2. Specification
   A document written by architects that specified the required chip functionality.

3. Behavioral Description
   Architectural Synthesis.

3. RTL Design
   Logical Synthesis (Code written on VHDL or VERILOG).

4. Gate Level Description
   Physical Synthesis.

5. Layout and Tape-out
   Last production to print on silicon.

6. Silicon
   The final chip.

Chip design and implementation is very complicated as mentioned in the IEEE article by **Ronny Morad, IBM Haifa Research Labs (Page 3) [2]**:

"Hardware-only system-level verification is definitely a growing challenge. SoCs today have more logic than ever before, encompassing functionality that spans from CPUs, to IO interfaces, message passing bridges, special purpose accelerators, and more. All this functionality needs to operate together correctly.
For example, a CPU must be able to handle the sending of a message while being interrupted because an accelerator completed its work.
Therefore, the number of possible interactions between all components in a SoC is huge."

Thus, getting the design right at the first time is an art.

In this matter Miron Abramovici and Paul Bradley wrote on September 2007 [3]:

"Even the most sophisticated SoC design methodology cannot fully account for all the parameters that impact silicon behavior, or for all logic "corner cases" that occur in the real life of a chip working at speed and in system."

Finding a bug after customer delivery cost millions, therefor post-silicon verification has become an essential step in the design implementation methodology.

As Ronny Morad goes on and say (Page 3) [2]:

"*What exacerbates this situation is that a bug in a single component in a SoC may require **another tape-out**, unless it can be easily bypassed by software.*
*This, in turn, creates **schedule delays** and **additional costs**.*
Due to all of the above, system-integration has to be verified. "

Figure 11 depicts the Cost of Bugs over Time:

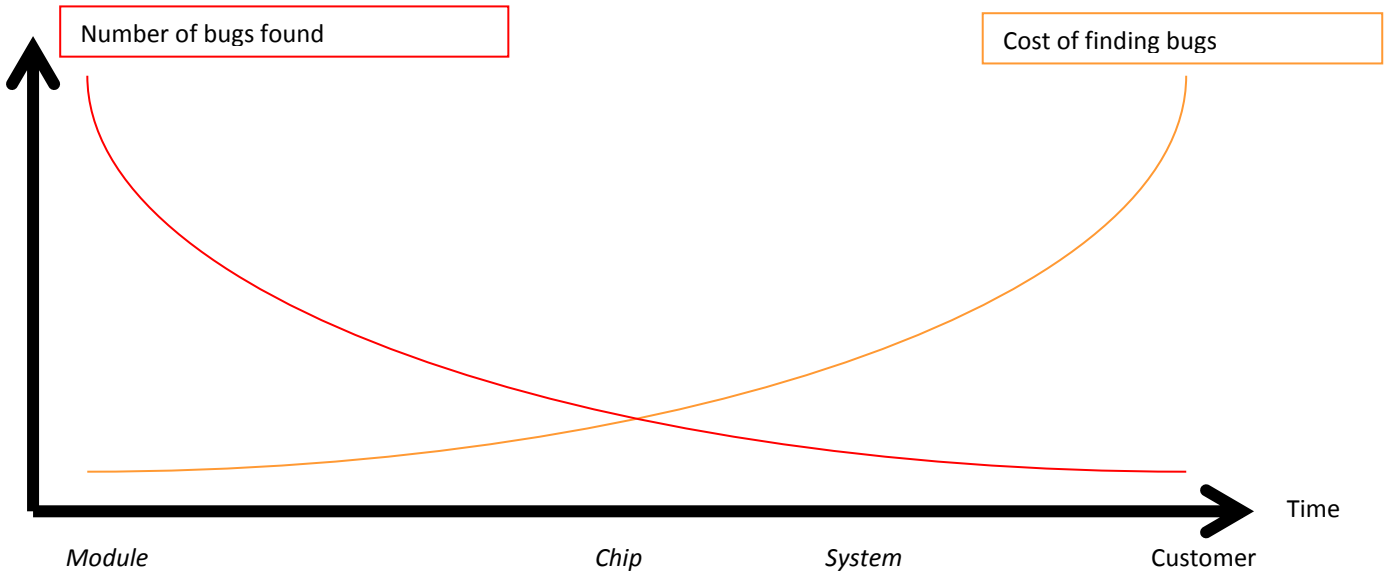


Figure 11: Cost bugs

Some more facts about Verification process [4]:

- 70% of the effort invested in SoC design goes to verification.
- 80% of written code is for the verification environment.

In order to fully verify chip design, you must show that the logic works correctly for all combinations of inputs, which means: driving all permutations on the input lines of the simulated chip and checking for proper results in all cases.

Full verification is not realistic as mentioned in an article written by Rangarajan (Sri) Purisai on the EE|TIMES online magazine:

"Even testing for known test-space can take a long time. For example, to verify the test-space of a simple adder that adds two 32-bit operands will take 232x232 clocks! When the logic gets more complex, the verification space increases. This brings about random dynamic simulation, which provides random stimulus to the design in an effort to maximize the functional space that can be covered" [5].

The common process for the verification is called Coverage.

2.2 Coverage

The coverage method conducts tests over simulated design where its input values are selected randomly and monitored continuously to make sure that the edge cases as well as the regular cases were tested. When driving random values the tests get to cases that a human couldn't even think of, and discover bugs that could show up only after producing the silicon chip and then create serious and expensive problem.

Randomization can be controlled, of course, because we would like to constraint the random values to the relevant values in the defined range of each parameter.

To complete the process, the chosen input values are collected during the test runs. Some parameters have specific values and have to be covered by these values accurately.
Other parameters require to be covered by ranges – one value in each range is enough for 100% coverage, with no need to test all the values.

Figure 12, lists on the left side the required ranges for a specific parameter, on the right side marked the percent of cover each range reached (usually one is enough).

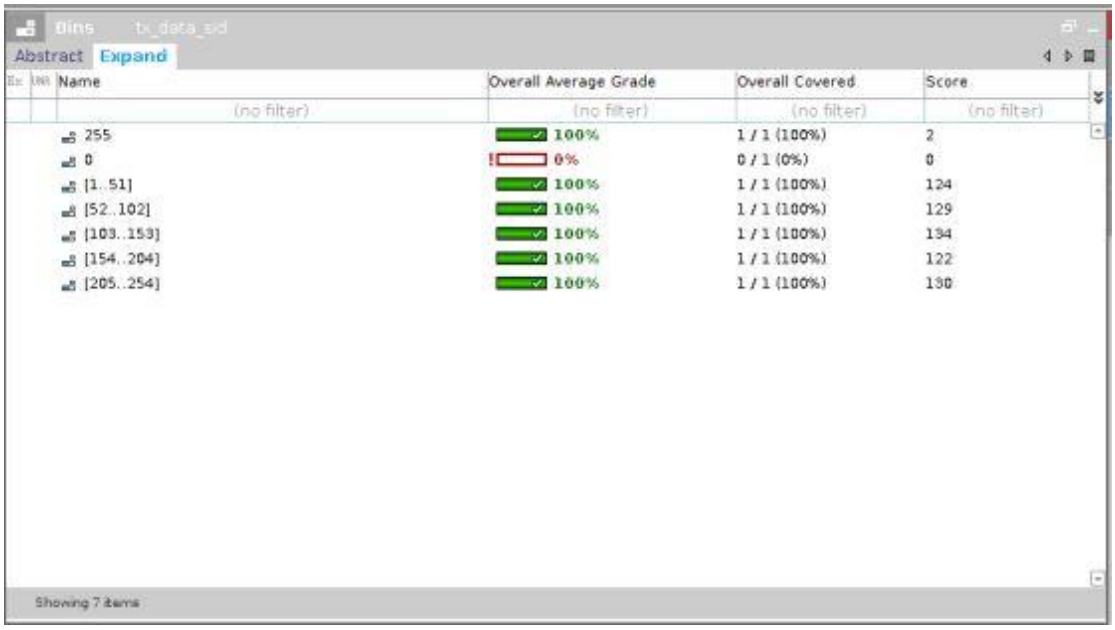


Figure 12: Values by ranges

A coverage has two kinds of checks – items and crosses.
Item-type coverage is checking only one parameter while
Cross-type check is a combination of situations the chip needs to reach.

The coverage method is accompanied with a well-defined document that includes all the simulation parameters.

In figure 13, there is a coverage document after a unit running, showing all the parameters to cover and the percentage of coverage they have. On the left side there is the list of parameters to cover each has an extension "_cov_e", on the right side there is marked which cover every parameter reached.

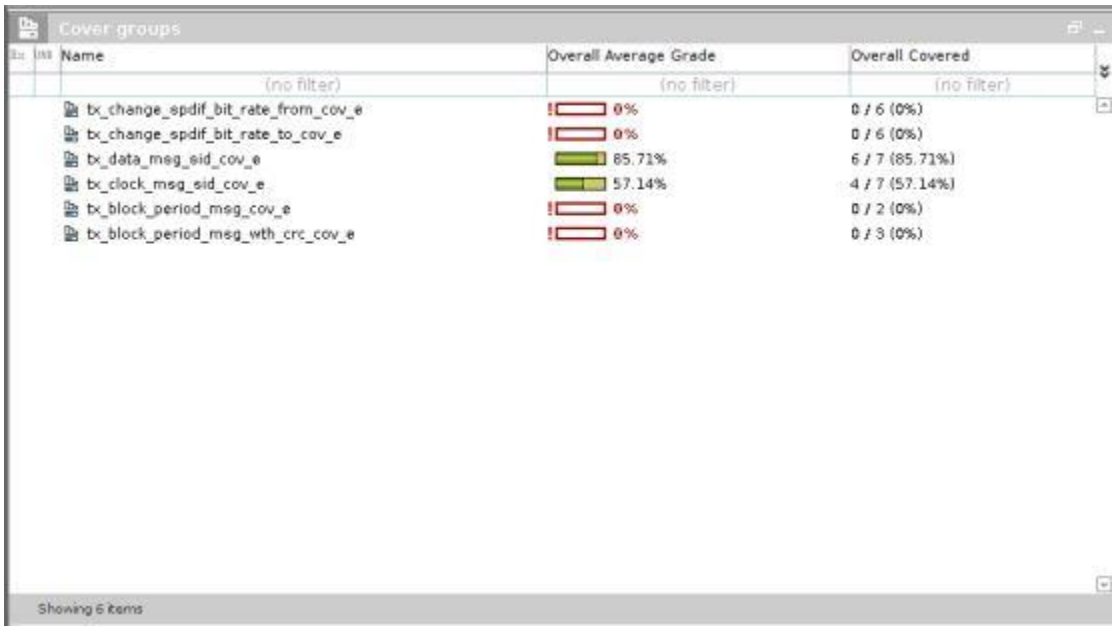


Figure 13: Coverage document

The goal is to get to 100% coverage.

So how actually is it done?

2.3 Verification in a practical manner

In general the verification process goes like this:
The verification engineer writes a software code that performs the same functionality that the hardware supposed to perform according to the specification of the chip.

In addition structure definitions or **items** are defined for representing the output and input values in the software according to the chip's protocols required parameters.
When testing the chip the software gets the same input as the hardware, processes the data and outputs an item.
At the same time the hardware processes the same input, and extract output - the results.
The software creates an item from that result without changing it, and compares the software item with the hardware item.

The point of this process is to conclude the following: if the resulting items are not completely equal, there is a mistake whether in the software code and whether in the hardware code.
Now the engineers just need to figure what is wrong and where is the error by debating about the specification and reach common conclusions.
Now if we assume that if the two engineers code the same functionality in different languages and without planning anything together, they will probably **not** do the same mistakes.
Comparing their results will hopefully have revealed all the bugs...

2.4 In terms of software engineering …

The main difference between hardware and software development is the time.
Software do not require time consideration.
Or as the hardware designers call it – software runs in 'zero time'.

As the HW and the SW are taking turns when running a verification environment, when the HW runs it takes time because it runs according to a particular clock with a specific rate like a clock of 125MHz or 500MHz, and when the SW runs it runs literally in zero time compare to the time that the HW required.
When developing a verification environment, the challenges are the HW time's requirements and synchronization at the software code so that it would be able to be compared against the hardware in the right way.

For example:
There is a chip that gets an asynchronous signal, and need to output it according to a particular clock.
One of the software requirements will be to check that the design output the signal at the right rate…
So when the SW gets an item of let's say a frame of this signal the environment will need to wait until the HW will finish toggling all the frame into the design, process the frame, and then check that bit by bit are coming out according to the right rate.
And this is one complicated thing to do…

2.5 The CpuPkt_Tadp block

The verification environment I am going to develop comes to verify the CpuPkt_Tadp block.
The goal of this block in the VS3000 is to provide another option to transfer packets between chips rather than the regular traffic path.
This block comes as a backup in cases of devices that work with a protocol that is different from the standard and need to change the configuration of the chips on both sides of the link, or on cases of programming errors, missed holes or bugs that cause failures or uncertain behavior.

In this cases there is a need to pass a packet with information than the regular path in the chip but an information to the chip itself, from chip to chip.

This can be made by the CpuPkt_Tadp.

Figure 14 provides an abstract view of the chip with traffic interface with a variety of TADPs and one **CpuPkt_Tadp**:

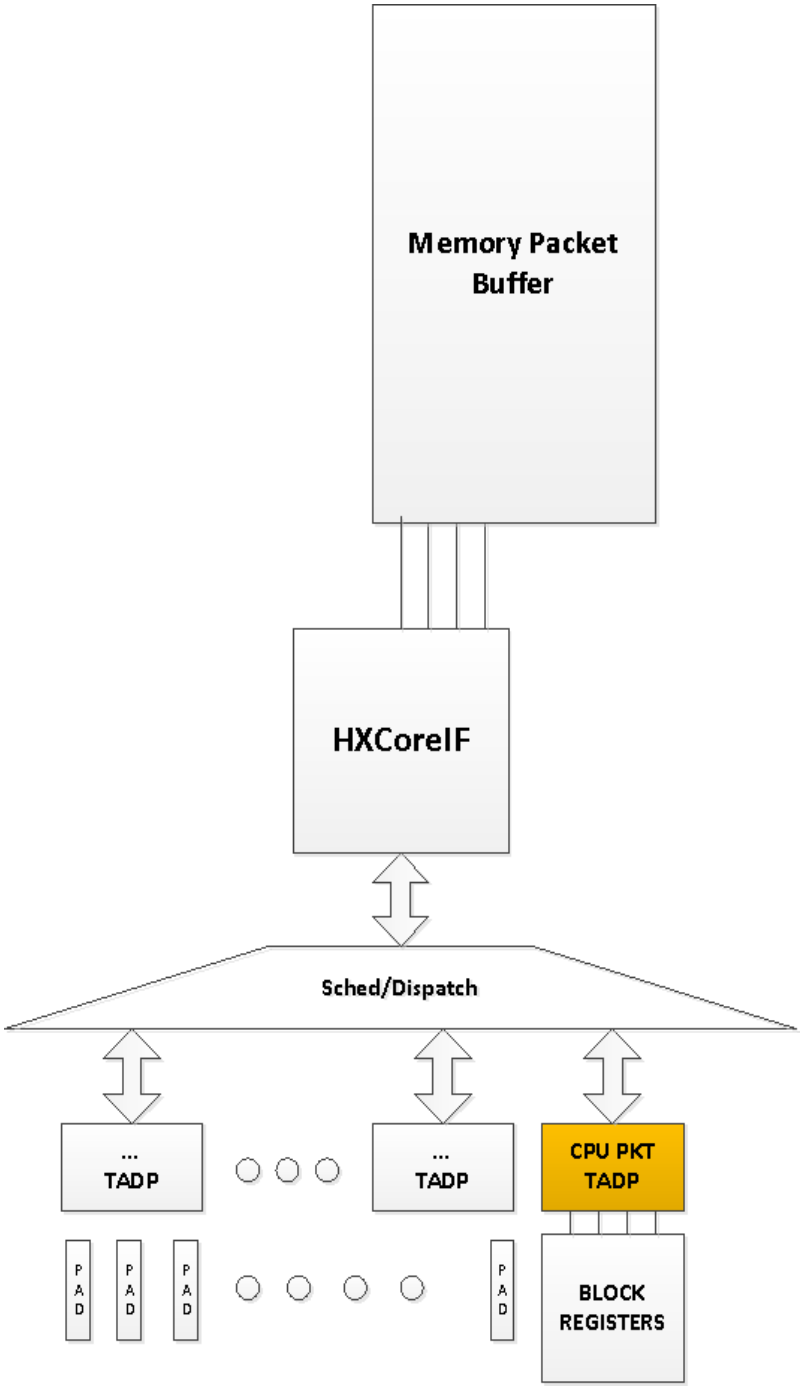


Figure 14: CpuPkt_Tadp Diagram

Each T-Adaptor block on the chip has defined types of tokens that it can get and process. The block can get any type of token and data, of any protocol that tokens can present like tokens of SPDIF, tokens of CIR, tokens of UART and also general tokens of clocks information and more.

Diagrams 15 shows the inside structure of the CpuPkt_Tadp:

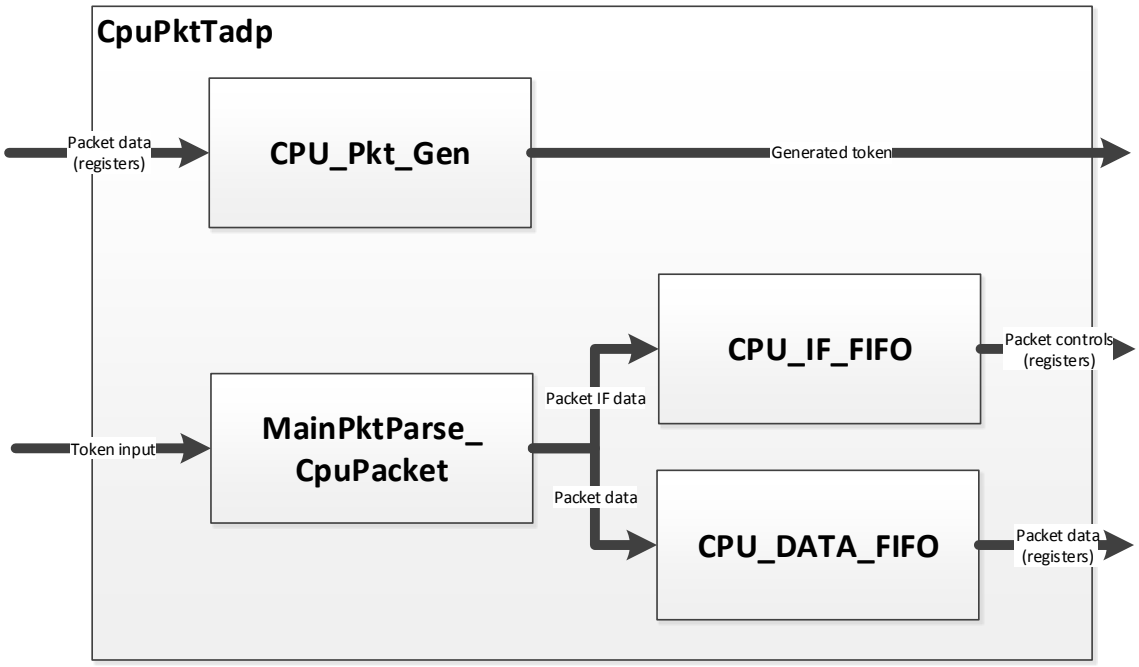


Figure 15: CpuPkt_Tadp Detailed Diagram

CpuPkt_Tadp block functionality:

- The CpuPkt_Tadp block acts as an adaptor and bridges the format difference between the CPU packets that are composed from native data (series of bits reflect in the content of each register) and the T-packets of the HDBaseT protocol.
- The CpuPkt_Tadp generates data into the system.
- The CpuPkt_Tadp reads/write data from/to registers.
- The packets that are emitted and received by the CpuPkt_Tadp should be merged with the regular HDBaseT data traffic. HDBaseT data traffic composed of tokens or t-packets that having a specific structure and order of bits.
- In the scenario there are two chips, the source and the destinations, both of them have two paths:
- RX – The CpuPkt_Tadp block reads data from registers that hold an information inserted by the SW, generate a token from all the data and send it out to the other chip.
- TX – The CpuPkt_Tadp block gets tokens arriving from the source chip, convert them to native data and writes the data to registers.

I am developing the verification environment and will use it to verify the behavior of the block - that it meets the functionality requirements as specified.

For this purpose I have wrote a verification plan that can be seen here.
The Checkers section (page 13, 14) includes the list of all the checkers and coverage that are required for this block.

3. Solution Description

In my project I develop a verification environment for the CpuPkt_Tadp. The verification environment is based on the **UVM** - **U**niversal **V**erification **M**ethodology.

Due to a confidentiality agreement I cannot add a link to my code so I attached appendix with snippets of code and explanations at the code appendix.

UVM Definition

"UVM is a methodology for building class-based verification environments in SystemVerilog, taking advantage of object-oriented programming techniques to help with code reuse. Reuse is right at the heart of UVM: if you are not planning to reuse verification code right from the start, then you may have missed the point of UVM."[8]

3.1 The CpuPkt_Tadp Verification Process

In this section I will present the structure of the software CpuPkt_Tadp verification environment that I developed.

As I explained the verification environment is a software that interfaces with the hardware – a design of a chip, and test it to verify the correctness of the chip operation.

The design developed by hardware designer in Verilog or VHDL, while the environment I – a verification engineer – develop in Specman.

My verification environment consists of the following software components each one called an **EVC** - '**E**nvironment **V**erification **C**omponent':
Agents, **sequence drivers**, **BFMs** and **monitors** that I already developed and I'm going to describe them in this section, and **reference models** and **score boards** that I will develop and will describe them in the next section.

The **EVC's** - **agents**, **sequence drivers**, **BFMs** and **monitors** are the part that responsible for interfacing and contacting with the hardware, while the **reference models** and **score boards** do all the checking and comparing operations, which are the core functionality of the verification.

The last part of my project is to develop the **tests**.
The **test** is the manager of the verification process, it starts the process by **do sequence**, and end the process by **stop_run ()**.
I have not developed the **tests** yet, and I will present them in the next section too.
I will run the tests after the whole environment will be ready in individual and then

regression of tests so that the process will meet the goal – to verify the CpuPkt_Tadp block.

3.1.1 The CpuPkt_Tadp SW Environment Architecture

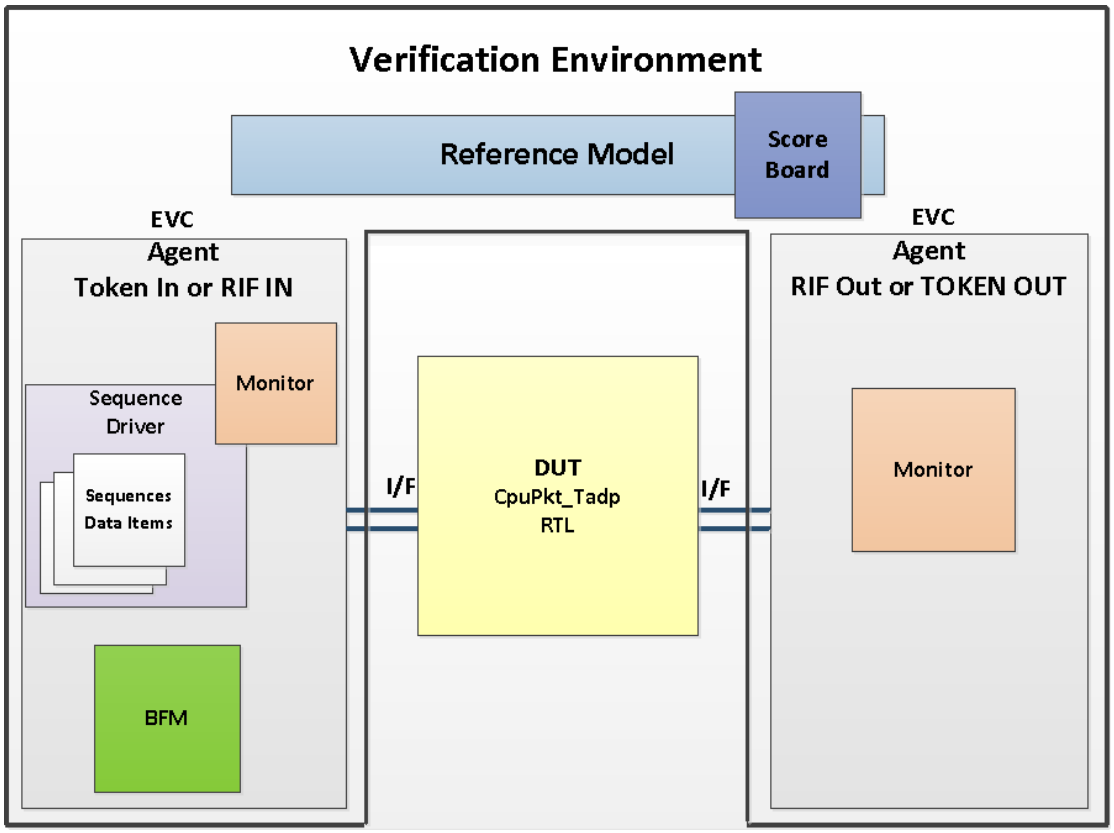


Figure 16: Verification environment RX and TX path

Figure 16 describe the CpuPkt_Tadp verification environment where the design under test, the **DUT** (in the middle) represents the tested HW in our case the CpuPkt_Tadp.
The code of the DUT is called the **R**egister **T**ransfer **L**evel code - the **RTL**.

3.1.2 Agent

The environment consists of two **EVC's** called **agents** (the grey software containers in Figure 16).

I have programmed a generic agent wrapper module that is described below.
Each agent has other internal EVCs placed inside it.

The **agents** drives and/or receive data items into/from the testing environment (DUT).

The Agents in the CpuPkt_Tadp environment are **token_in**, **token_out**, and **RIF's** agents (RIF is a protocol of write and read register operations).

On the CpuPkt_Tadp there are **RX** (Receiving) path and **TX** (Transmitting) pass.
The **token_in** agent drives transactions of tokens into the DUT (using the TX pass).
The **token_out** agent receives tokens from the DUT (using the RX pass).
The **RIF's** agent drives data into the DUT via write operation, and receives data from the DUT via read operation.

When agent is **ACTIVE**, it has a **sequence driver** and **BFM** and it drives sequences into the DUT, and a **monito**. When agent is **PASSIVE,** means – it has only **monitor**.

For example: The "in" agents like **token_in** in our case will be ACTIVE, while the out agents like **token_out** will be PASSIVE.

3.1.3 Sequence driver

I have programmed a specific sequence driver for the CpuPkt_Tadp as follows. The **sequence driver** gets sequences from the test by **try_next_item ()** function call. The sequences are required upon **BFM** demand.
When the **BFM** calls the base method of the sequence driver – **get_next ()**, the sequence driver request for the next item and pass it to the **BFM.**

The sequences composed of items that fit the specific kind agent that this sequence driver located in.

For example: The token_in sequence driver gets sequences of tokens from the test, and the values of the tokens are according to the specifications of the chip.

And what does the **BFM**?

3.1.4 BFM

The BFM – **B**us **F**unctional **M**odel - is the one that actually drives the input values into the **DUT.**
The **BFM** software component I have developed is connecting to the DUT by an interface of ports.
It gets sequences and data items from the sequence driver, and drives each value to the appropriate port.

For example: The sequence driver of the **RIF's** agent gets read and write operations and transfer them to the **RIF's BFM**.
The **RIF's BFM** makes a respective stream of bits from the programming scenario and actually injects that stream into the DUT.

My code had to make sure that before the **BFM** drives values into the **DUT** it needs to check that the **DUT** is ready to get input, and not busy processing anther data.

The **BFM** knows when the **DUT** is ready by checking a physical signal that the **DUT** raises when it is ready to get inputs.

When the **BFM** done with driving an item it calls the sequence drive's **get_next ()** method to go on.

3.1.5 Monitor

I have developed the monitor module as follows. While the **BFM** pushes the data into the DUT, the **monitor** collects the data from the input interface's lines and create a data item according to the protocol the agent that it located in.

For example: **RIF monitor** will create RIF item of read or write operation, while **Token monitor** will create a token item.

This monitor is located in the left agent of figure 15 and **monitors both** agents: the left and the right**.**
This **monitor** collects the out data from the output interface's lines.

In the **DUT** the data goes through a processing and then goes out on the output interface ports.
The **monitor out** collects the data and create fitting data items just as the **monitor in.**

In this case of the CpuPkt_Tadp on the RX path will be a **RIF monitor in** and a **token monitor out,** and on the TX path will be a **token monitor in** and a **RIF monitor out**.

3.1.6 Data Item

The **data item** is not an **EVC**.

It is a **struct** that gives uniformity and order to the data that is going to be driven into the **DUT**.

The structure that was designed by me comes to reflect patterns of the RIF's operations.

For example:
Some of the ports of the RIF are –

- rd, wr – Indication if it is write transaction or read transaction.
- data_rd, data_wr – The data of the transaction.
- addr – The register's address where the RIF read/write from/to.

The item's fields that are conform to the ports:

- **addr** – The address of the register where you read from/write to.
- **data** - The data you read/write.

This **data item** is an important building block of a sequence and allows organized libraries of sequences.

Usually the **data item** is built according to the interface's ports of the protocol that it present.

Here, in the CpuPkt_Tadp environment there are two data items: token transaction, and RIF's transaction.

3.1.7 The Verification Process

In the next figure you can see a summary of the verification process over time:

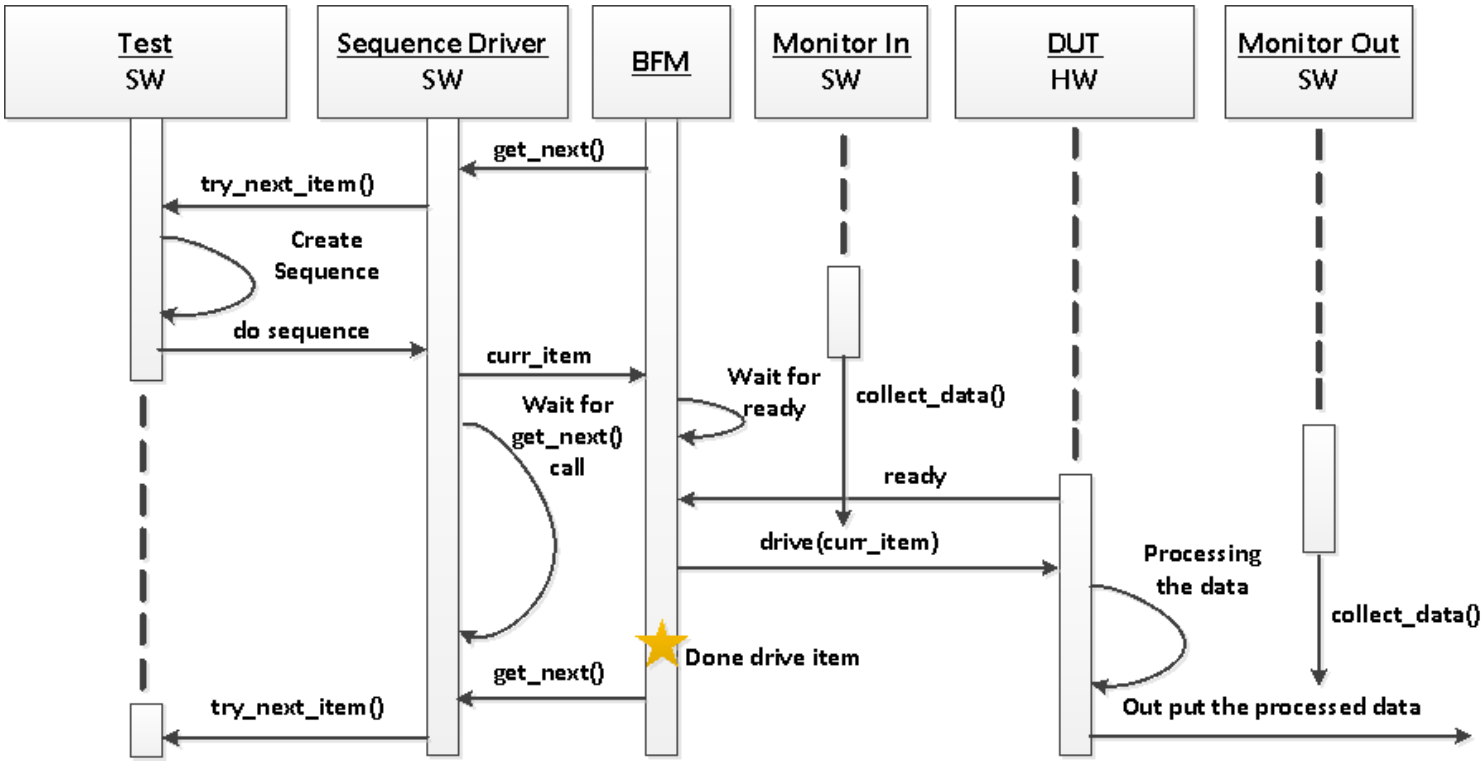


Figure 17: Sequence Diagram of the Verification Process

The **BFM** asks for items from the **sequence driver** that tries to get an item from the **test**.
The **test** creates a sequence and transfers the sequence to the **sequence driver** that transfers a **data item** to the **BFM** on its turn, and waits for the next call for next item from the **BFM**.
The **BFM** waits for the **DUT** to be ready to get data and drives the data into it.
While the **BFM** drive data, the **monitor in** collects the data from the input ports exactly as it goes into the **DUT**.
The **DUT** processes the data, and output it, and the **monitor out** watches the output ports and collects the data from them, exactly as it comes out from the **DUT**.

Up until here I present the process of the software components that I already developed.
In the next step I will develop the components to check the data and compare it to conclude whether the DUT works well or has some bugs to fix…

3.2 Testing Program

In the verification process as much as the software verify the design, the design verifies the software.

When writing the code I will follow a list of checkers and testing that the design – the **RTL** should be checked. This list can be seen in the 'Checkers' appendix.

Then, during the verification process I will run regressions of tests over my CpuPkt_Tadp environment and the CpuPkt_Tadp design.

Test fails due to HW bug or SW bug.

First of all, I will check my code according to the failure, and after I will be sure that my code is correct, I will pass the handling of the bug to the designer.

When a regression will be over with no fail then I'll know that my code has no mistakes, as much as the **RTL**.

3.3 Language Tools and Environments

In this project I am using the following:
Operating system: Linux.
Language: E with the Specman tool.

One word about Specman E….
Although SystemVerilog is the language associated with UVM methodology, the Specman E language fully supports the UVM points because it uses an aspect-oriented programming (AOP) approach.
So the project will be developed with Specman E in the UVM methodology.

I will work on remote servers of Valens with the VNC viewer.

The simulator will be the Cadence Enterprise Incisive that is a very power full and convenient simulating and debugging simulator.
The simulations run on a remote machines because a regular computer cannot run a simulation of FPGA chip.

4. Similar works in Literature and Comparison

As mentioned before there some other verification methodologies.

Some of the other verification Methodologies are VMM and OVM.

Aldec, the Design Verification Company introduce [6]:

Verification Methodology Manual *(VMM) was the first successful and widely implemented set of practices for creation of reusable verification environments in SystemVerilog. VMM contribution was an important factor in creation of UVM.*

Open Verification Methodology *(OVM) is the library of objects and procedures for stimulus generation, data collection and control of verification process. As the first SystemVerilog-based verification library available on multiple simulators, OVM contributed significantly to the development of its successor, UVM.*

Besides the various methodologies there are different verification techniques:

Formal verification, random, directed, constrained random, assertions, property checking and more.
Each of these techniques presents a different way of creating sequences and set values to the different parameters.

Those techniques and methodologies can be developed in some different languages, i.e. SystemC, C/C++, SystemVerilog, OpenVera, e.

With many methodologies and a lot of techniques the question surfaces:
In which way to choose for verifying a chip design???

The answer is not as simple as figured from an article in the EE|TIMES online magazine of the global electronics community written by **Rangarajan Purisai** [7]:

"It is a known fact that functional verification takes the lion's share of the design cycle. With so many new techniques available today to alleviate this problem, which techniques should we really use? The answer, it so happens, is not straightforward and is often confusing and costly!

The tools and techniques to be used in a project have to be decided upon early in the design cycle to get the best value for these new verification methods. Companies often end up making costly mistakes by underestimating or sometimes overestimating the complexity of the design and skill set required to run these new tools and techniques."

But as you can see the UVM is the most common today.

Ramdas Mozhikunnath, Experienced Verification Engineer, Intel Alumni, Author and Online teacher:

"UVM (Universal Verification Methodology) is a SystemVerilog language based Verification methodology which is getting more and more popularity and adoption in the VLSI Verification industry. The methodology is currently in the IEEE working group1800.2 and is expected to be an IEEE standard shortly. "[8]

On the next figure there is a time line of the verification methodologies evolution, up until 2010, since then everyone use the UVM.

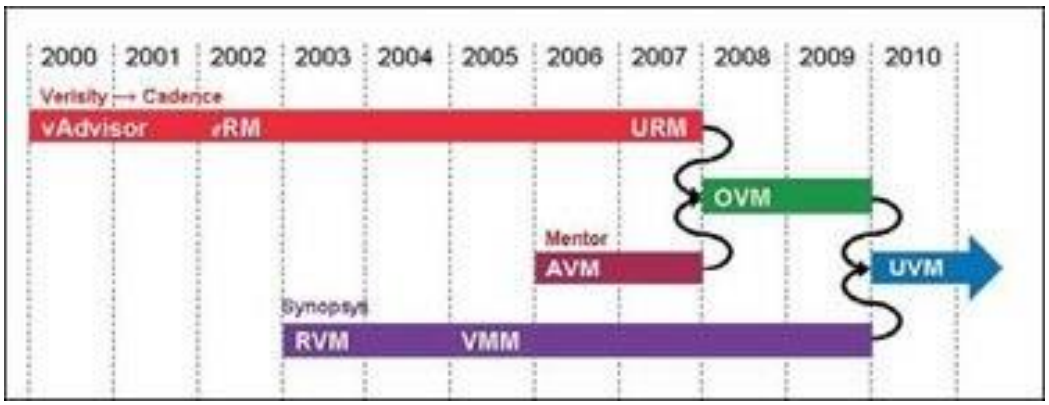


Figure 15: Time line verification methodologies

And what about the future?

"We believe that in the next ten years formal will become the default verification strategy in the verification flow, routinely used to sign-off at the unit level. This is already possible today."
Writes **Pippa Slayton Marketing and Business Development Manager [9].**

Just for your knowledge (Section 1):

"The main idea of formal hardware verification is to prove the functional correctness of a design instead of simulating some vectors. For the proof process different techniques have been proposed."[10]

And it is expected for 2025…

5. Appendix

5.1 Bibliography

[1] http://www.youtube.com/watch?v=GuWertTgBpE&t=0m34s

[2] http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6800586

[3] http://www.design-reuse.com/articles/16730/a-new-approach-to-in-system-silicon-validation-and-debug.html

[4] IBM.ppt

[5] http://www.eetimes.com/document.asp?doc_id=1217826

[6] https://www.aldec.com/en/solutions/functional_verification/uvm_ovm_vmm

[7] http://www.eetimes.com/document.asp?doc_id=1217826

[8] https://www.quora.com/What-is-Universal-Verification-Methodology-UVM-in-laymans-terms/answer/Ramdas-Mozhikunnath?srid=kd5Y&share=cf132a53

[9] http://www.oskitechnology.com/blog/formal-2025-its-back-to-the-future.html

[10] http://www.informatik.uni-bremen.de/agra/bookdetail/download/intro_afv.pdf

5.2 Risks Table

Risk	Severity	Possible solution
Chip architecture changes	High	No
Removing a component from the chip due to excessive complexity Of the chip.	High	No
Design changes : • Add new feature for additional functionality. • Specification changes following laboratory falls.	High	Start code the design as soon as possible, write the verification code as generic as possible…

Design bugs	Medium	This is a risk but the whole idea of verification… Nothing to do! ☹
Verification code bugs	Medium	Understanding the specification of the chip as much as possible before start coding.
Unexpected changes in the company: • A staff member leaves. • Assignment gets low priority.	Depends	No

5.3 User Requirement Table

Requirement number	Description
1	Vplan – I Wrote a full description and well planned verification document. (Already done and attached here.)
2	Checkers – Writes checks for all the block features. (The full checkers list required is on the attaches Vplan, part 4.6)
3	100% coverage – Detailed in specific parameters and values in the Vplan of the specific block. (The full coverage values required is on the attaches Vplan, part 4.8)
4	Neat and readable and well documented code.
5	Versioning – Once in a while deposit the code in the main reservoir calls $DBASE.
6	Full generic source code in accordance with the company existing models in the main reservoir.
7	Preparing code to full chip integration, when integration all the block in the chip and simulate all of them simultaneously.

5.4 Checkers

5.4.1 General checkers list

The checker name	Description	Location	priority
Data correct	The RM is compared the data native data vs data Token according the configurations: -Data: All registers values. -Token: All options of token configurations.	RM	High
Idle mode/got reset	Token Ready out signal is on '1' The status registers are on default values.	RM	High
Loopback mode	Check that on loopback mode data does not go out but going back.		Low

5.4.2 Token checker list

The checker name	Description	Location
Generate token	Check that Tokens are not coming out before go command.	Monitor
CRC error pkt	• When the ignore CRC bit is on, ignore bad CRC indication use the data pkts. • Else ignore the packet.	RM
Token space transaction	Check default space.	RM
Check the Token trans values	Check according to the configurations.	RM
Token_out back pressure mode	Check that the RTL does not stuck (Data can be lost).	General

5.4.3 Native Data checker list

The checker name	Description	Location
FIFO validation	Check that data is not coming out from FIFO without valid.	Monitor
Data validation	Check that data is not collected with no valid high.	Monitor

5.4.4 CFG checker list

The checker name	Description	Location
Read only status registers	Correct values and behave	

5.5 Code

The code that I will present here is the code of the EVC's of the RIF's environment.

5.5.1 Sequence driver

```
74
75 sequence vl_rif_sequence_s using
76   item  = vl_rif_trans_s ,
77   sequence_driver_type = vl_sequence_driver_u,
78   created_driver = vl_rif_sequence_driver_u ,
79   created_kind   = vl_rif_sequence_t ;
80
81 extend vl_rif_sequence_driver_u {
82    get_next() : vl_data_item_s @clock is {
83       result = try_next_item();
84    };
85 };
86
```

The piece of code above generates the sequence driver automatically in Specman.
Every unit's name of Valens starts with **vl** as on **line 75**.
Also each unit has a specific type or kind and a unique name.
In the above code from **line 76** to **line 79** the sequence driver and the sequences and items that it creates get their type/kind and name.

Specman is an aspect oriented language. Every unit can be extended anywhere in the code and get more parameters or methods.
Even methods can be extended with is_also statement, or completely overwrite with is_only statement.

In the above code from **line 81** to **line 84** there is an expansion to the token sequence driver that created before, and it gets a method name **get_next()** – the main and must method to get items, that returns an item – a sequence by the build in method – **try_next_item()**.
The method **try_next_item ()** try to get the next sequence from the sequence driver when the BFM finishes with the previous item.

In order to explain the code of the RIF's BFM, first see the code of the RIF's item and the RIF's transaction and the RIF's ports:

5.5.2 Item

The code above is the definitions for the RIF:
Line 28: There are two types of RIF's agents **MASTER** and **SLAVE**.
The master start a transaction by indicating a write or read with relevant address, byte enable and data in case of write and hold it. The slave answers acknowledge

and returns the data in case of read, this ends the transaction and free the buss to the next one.

```
72 struct vl_rif_trans_s like vl_data_item_s {
73 };
```

Line 30: This definition comes to differentiate between reading and writing operation.

This is the initial definition of the RIF's item.
The name of the RIF's item is **vl_rif_trans_s**.
'like' is the inheritance statement in Specman, when the new struct gets the characteristics and method of **vl_data_item_s** which is Valens basic data item that all items inherit from.

In general Valens has a generic file system with the basic settings of everything from a basic files of Valens each has basic definitions and methods and every new unit or struct inherit from them respectively.

For example: Here **vl_rif_trans_s** as just written above, inherits from the basic **vl_data_item_s** the basic verification item of Valens.
The **agent, BFM,** and the **monitor** inherit from vl_agent/bfm/monitor_u that themselves inherit from vl_unit_u the basic unit of Valens.

After the definitions let's get to the item:

```
 3 extend vl_rif_trans_s {
 4
 5    // pointer to the agent
 6    my_agt : RIF vl_agent_u;
 7    keep soft my_agt == try_enclosing_unit(RIF vl_agent_u);
 8
 9    // this set if the item is on the MASTER side
10    // or in the SLAVE side
11    rif_kind : vl_rif_device_t;
12    keep read_only(my_agt!=NULL) => rif_kind == my_agt.rif_kind;
13
```

```
13
14    // this is the time till change (new rif item in master and rif ack in slave)
15    time_till_rif : uint;
16    keep soft time_till_rif in [0..50];
17    keep read_only(rif_kind == SLAVE) => time_till_rif == 0;
18
19    %rif_type : vl_rif_trans_t;
20    keep read_only(rif_kind == SLAVE) => rif_type == WR;
21
22    time_till_ack : uint;
23    keep soft time_till_ack in [0..50];
24    keep read_only(rif_kind == MASTER) => time_till_ack == 0;
25
```

```
26    %addr : uint;
27    keep read_only(rif_kind == SLAVE) => addr == 0;
28
29    %data : uint;
30    keep read_only(rif_kind == MASTER and rif_type == RD) => data == 0;
31
32    %be : uint (bits : 4);
33    keep read_only(rif_kind == SLAVE) => be == 0;
34    keep soft read_only(rif_kind == MASTER and rif_type == RD) => be == 0xF;
35    keep read_only(rif_kind == MASTER and rif_type == RD) => be in [1,2,4,8,3,0xC,0xF];
36    keep read_only(rif_kind == MASTER and rif_type == WR) => be in [1,2,4,8,3,0xC,0xF];
37
```

```
38    keep next_item_delay == 0;
39
40    error_packet : bool;
41    keep soft error_packet == FALSE;
42
43 }; // extend vl_rif_t...
```

Due to the randomization concept Specman has randomization internal mechanism called **Generator** that generate a random value for each parameter in the environment at the beginning of the run according to its type.

For example: if **i** is of type **uint (bits: 8)** it can get values from the range: **[0...255]**. Although usually narrow down the space values with constraints, 'keep' or 'keep soft' statement, like: **keep i <= 20**.
'keep' cannot be changed during the run-time, while 'keep soft' can.

The **read_only ()** method take care that when a parameter receives its value based on the value of another parameter it will get its value just **after** the parameter it dependent on will be generated.

As shown in the code above RIF's transactions has the next fields:

- **Lines 6, 7: my_agt** - Each EVC in the agent has a pointer to its agent.
- **Lines 11, 12: rif_kind** – Sets if it is a transaction of MASTER or SLAVE device. This field permeates from the agent**.**
- **Lines 15, 17: time_till_rif** – Sets the time till transaction will start.
- **Lines 19, 20: rif_type** – The transaction type: write-wr, or reading-rd.
- **Lines 22, 24: time_till_ack** – Sets the time from transaction start to acknowledge.
- **Lines 26, 27: addr** - The register where you read from/write to.
- **Lines 29, 30: data** - The data you read/write.
- **Lines 32, 36**: **be** - The byte enable of the transaction. The defined size of a register is 32 bytes, but sometimes the valid data is not filling a whole register but only the enabled bytes. The options are {1, 2, 4, 8, 3, C, F}.

The '%' mark is to say that these parameters are going to be compared on the checking phase that will be later.

- **Line 38: next_item_delay** – This field inherit from vl_data_item_s. It sets how long the BFM will wait after done with one item until asking the sequence driver for another item.

- **Line 40, 41: error_packet** – This field is for debugging purpose, to test the **RTL** by a transaction with an error.

After introducing with the item let's see how the BFM pushing it into the design:

5.5.3 BFM - Beginning

```
3  extend RIF vl_bfm_u {
4
5     !rif_ports : rif_ports_u;
6     !rif_agt : RIF vl_agent_u ;
7
8     keep soft en == TRUE;
9
10    -- this set if the bfm is on the MASTER side or in the SLAVE side
11    !rif_kind : vl_rif_device_t;
12
13    -- this hold the current item being driven
14    !rif_item : vl_rif_trans_s;
15
16    connect_pointers() is also {
17       rif_agt = its_agt.as_a(RIF vl_agent_u);
18       rif_ports = rif_agt.rif_ports;
19       rif_kind = rif_agt.rif_kind;
20    };
21
22 }; // extend RIF vl_...
```

The RIF's BFM has an en filed which inherited from the vl_bfm_u that by default is enabled but it can be disabled when the BFM is unnecessary.

The BFM also has a pointer to the RIF's interface ports.

The '!' mark is to say that these parameters are generated with random values, but get their values only when it is explicitly written.

For example: Here in the RIF's BFM the pointers to the agent, the interface ports and the kind get their value in the **connect_pointers ()** method on **lines 16 to 20**.

5.5.4 Ports

The definition of the RIF's port:

```
 3 extend rif_ports_u  {
 4
 5    rd : inout simple_port of bit is instance;
 6    keep bind(rd , empty);
 7
 8    wr : inout simple_port of bit is instance;
 9    keep bind(wr , empty);
10
11    ack : inout simple_port of bit is instance;
12    keep bind(ack , empty);
13
14    addr : inout simple_port of uint is instance;
15    keep bind(addr , empty);
16
17    data_rd : inout simple_port of uint is instance;
18    keep bind(data_rd , empty);
19
20    data_wr : inout simple_port of uint is instance;
21    keep bind(data_wr , empty);
22
23    be : inout simple_port of uint (bits : 4) is instance;
24    keep bind(be , empty);
25 };
```

Each port is an **inout** – port that is used as in and out port of the design, **simple port** – there are several types of ports in Specman, but these ports are literally simple ports – actually ports of the design.

The **bind()** method binding the ports to the design. Here in the sample file they are binding to en empty location. The real binding will be in the environment file, and will be shown soon…

5.5.5 BFM – Continue

```
52    run() is also {
53       if( its_cfg.do_test_flow ) {
54          start test_flow();
55       };
56       if (its_cfg.do_init_signals)  {
57          start init_signals();
58       };
59    };
```

The flow of the BFM defined in the vl_bfm_u as follows:
run() is a system function that called automatically on the test phase, after the generator has done, and exists in each unit and can be extended with "is also" statement as in **line 52**.

On the BFM the **run ()** function operates the following functions: **init_signals ()**, and **test_flow ()** both are functions of vl_bfm_u.

The **test_flow ()** method calls to the **test_loop ()** method and **test_loop ()** operate the actual flow of the BFM as follows:

```
61    test_loop()@uclk_ev is also {
62       message(MEDIUM, "Starting test for agent ", agt_name);
63       while( not is_done ) {
64          if( suspended ) { sync @resume_ev; };
65          if( is_eot ) { break; };
66          cur_item = sd.get_next() ;
67          if( cur_item != NULL ) {
68             drive( cur_item );
69             emit sd.item_done;
70             cur_item.quit();
71             wait [cur_item.next_item_delay];
72             pre_item = cur_item;
73          } else {
74             wait [1];
75          };
76       };
77    };
```

The base BFM has a Boolean fields called **is_done** and **is_eot(e**nd **o**f **t**est)**,** which are set to FALSE, so the loop on **line 63** ends when the **is_done** gets FALSE value or break on **line 65** when the **is_eot** gets FALSE value.

Else the **test_loop ()** is going on and try on **line 66** to get the next item from the sequence driver.
If the return item is NULL then the BFM wait for one cycle and try again, on **line 74**.
When an actual item returns the **drive ()** function operate, **line 68**, and really drive the item's values into the DUT as will be shown soon.

When the driving into the DUT is done, the **item_done** system event of the sequence driver is emitted on **line 69**.

What is an event?
In Specman there are events. When **'emit'** an event wherever there is a reference to it there is a reaction to the emission.
For example: If there is a **wait [event_e]**, the delay will be over when somewhere will occur – **emit event_e**.

After letting the environment know that it is done with the current item, on **line 71** the BFM waits for the basic delay between items which represent in the item's field: **next_item_delay**.

The **drive ()** function is different in each mode- MASTER and SLAVE.

```
47    extend MASTER RIF vl_bfm_u {
48
49    init_signals()@sys.any is also {
50       rif_ports.rd$          = 0;
51       rif_ports.wr$          = 0;
52       rif_ports.addr$        = 0;
53       rif_ports.data_wr$     = 0;
54       rif_ports.be$          = 0;
55    }; // init_signals()@...
```

On **line 47** there is an extension of the **MASTER** kind in addition to the type – RIF - extension.
Which means that when an agent would have a **MASTER** kind its BFM will execute the code on the above extension, when an agent would have a **SLAVE** kind its BFM will execute the code on the bellow extension where there is further extension of **SLAVE** kind on **line 186**.

The method **init_signals()** in both kinds on **lines 49 to 55 and 188 to 192** gives initial values to the related ports.

```
186 extend SLAVE RIF vl_bfm_u {
187
188    init_signals()@sys.any is also {
189       rif_ports.ack$      = 0;
190       rif_ports.data_rd$ = 0;
191       rif_ports.ack_addr$ = 0;
192    }; // init_signals()@...
```

```
57    drive(di : vl_data_item_s)@clk_ev is only {
58       rif_item = di.as_a(vl_rif_trans_s);
59       wait[rif_item.time_till_rif];
60       rif_ports.addr$ = rif_item.addr;
61
62       if not rif_item.error_packet {   -- Good RIF Packets
63          rif_ports.be$   = rif_item.be;
64          if rif_item.rif_type == WR {
65             rif_ports.wr$ = 1;
66             rif_ports.data_wr$ = rif_item.data;
67          } else {
68             rif_ports.rd$ = 1;
69          };
70          var got_ack : bool;
71          var ack_ctr : uint;
72          while not got_ack and ack_ctr <= rif_agt.timer_to_ack {
73             wait [1];
74             ack_ctr +=1;
75             got_ack = rif_ports.ack$ == 1;
76          }; // while not got_a...
77          rif_item.time_till_ack = ack_ctr > rif_agt.timer_to_ack ? UNDEF : ack_ctr;
78          rif_item.data = rif_ports.data_rd$;
79          rif_ports.wr$      = 0;
80          rif_ports.rd$      = 0;
81          rif_ports.addr$    = 0;
82          rif_ports.data_wr$ = 0;
83          rif_ports.be$      = 0;
84       } else {
85          -- Code to handle Rif packets consisting of errors
86       };
87    }; // drive(di : vl_d...
88 }; // extend MASTER R...
```

The **drive ()** method of **MASTER BFM** kind:

First of all, the BFM waits the time till start of a RIF's transaction on **line 59**.
Then on **line 60** the BFM pushes the address from the item's field – **addr**, into the DUT through the port **addr** that is a port in **rif_ports** as shown already.

The '$' marks an access to a physical port.
This format is used all over the BFM to push the item's values into the DUT.

From **line 62 to 69** start the transaction if it is not a packet with error, and the right values get pushed into the concurrent ports.

From **line 70 to 76** the BFM waits for **ack** to finish the current transaction.
The variable **ack_ctr** starts from 0 (Specman initialization value of uint variable), and increased after each one cycle that **got_ack** – a Boolean variable is not TRUE. When the **ack** port has a value of 1 the statement on **line 75** returns TRUE into **got_ack**.
The RIF's agent has a timer for the ack issue.

On **line 77** When **ack_ctr** is greater than this timer the while loop ended and the **time_till_ack** field of the RIF's item gets an **UNDEF** value to alarm that ack did not come and there is a problem.

After then the data filed gets a value from the port of the data that comes back at a read transaction, and at the end - reset all the ports.

The **drive ()** method of **SLAVE BFM** kind is just the same only with the related ports.

5.5.6 Monitor

As usual – extension from the vl_monitor_u, pointers to the interface ports and to the agent.

As in the BFM the monitor also can be enabled and disabled by the **en** field.

col_item is the item that will be collected.

```
22    collect_data()@clk_ev is {
23       while (en) {
24          sync true(rif_ports.wr$ == 1 or rif_ports.rd$ == 1)@clk_ev;
25          check that not (rif_ports.wr$ == 1 and rif_ports.rd$ == 1) else
26            dut_error("wr and rd can't rise in the same cycle");
27
28          col_item = new with {
29             .my_agt     = rif_agt;
30             .time_start = sys.time;
31             .rif_kind   = rif_agt.rif_kind;
32             .rif_type   = rif_ports.wr$ == 1 ? WR : RD;
33             .addr       = rif_ports.addr$;
34             .be         = rif_ports.be$;
35          }; // col_item = new ...
36          if col_item.rif_type == WR {
37             col_item.data = rif_ports.data_wr$;
38          }; // if rif_item.rif...
```

collect_data () is an abstract method of vl_monitor_u that implemented here by the **'is'** statement on **line 22**. It collects the data from the lines.
First of all the monitor synchronize on occurrence of read or write operation on **line 24**, but check that there is no read and write operation together which is not allowed.
If there is something like that the occur **dut_error ()** that is stopped the run immediately.

On **lines 28 to 38** the monitor creates the RIF's item through the **new with** statement.
This statement creates a new struct in the memory and fills its fields with values, here - with the data that comes from the ports, or the monitor fields.

sys.time on **line 30,** is the actual time when this code is executed.

```
40          first of {
41             {
42                while TRUE {
43                   ack_ctr +=1;
44                   if ack_ctr > rif_agt.timeout_time * 2 {
45                      dut_error("no ack to transaction after ",rif_agt.timeout_time * 2);
46                   }; // if ack_ctr > TI...
47                   if (ack_ctr>= rif_agt.timer_to_ack -20) {
48                      col_item.error_packet =TRUE;
49                   };
50                   wait [1];
51                }; // while not got_a...
52             }; //
53             {
54                if col_item.rif_type == WR {
55                   sync true(rif_ports.ack$ == 1 or rif_ports.wr$ == 0
56                            or rif_ports.addr$ != col_item.addr)@clk_ev;
57                } else {
58                   sync true(rif_ports.ack$ == 1 or rif_ports.rd$ == 0
59                            or rif_ports.addr$ != col_item.addr)@clk_ev;
60                };
61                if rif_ports.ack$ == 1 {
62                   col_item.time_till_ack = ack_ctr;
63                   if col_item.rif_type == RD {
64                      col_item.data = rif_ports.data_rd$;
65                   }; // if rif_item.rif...
66                   wait [1];
67                } else {
68                   col_item.time_till_ack = UNDEF;
69                }; // ! if rif_ports....
70             }; //
71          }; // first of
```

Continuation of **collect_data ()**:

The **first of** statement is like fork of threads. The blocks in the 'first of' runs concurrently, but when the first of them ends – all the threads stopped even in the middle of the run.

Here there are two block of the 'first of' starts on **line 40:**
The first one is from **line 41 to 52**, and the second is from **line 53 to 70**.

The first block is functions as a time out.
The **ack_ctr** increase every cycle. If it is get closer to the value of the timer in the agent **line 47 to 49,** it is indicate an error packet, and the collected item will have this indication, and will let the agent know that something is wrong…

If the **ack_ctr** continue increasing till getting to the time out value two times – then the error is so severe that the whole run is stopped by **dut_error().**

The second block synchronize on a change on the lines of the ack port, or the address port or the rd/wr port depend on the type of the current operation on **line 54 to 60** to synchronize on the end of the transaction. Then check if the ack is raised and if it is raised - if it is a read transaction collects the data from its port, and set the **time_till_ack** item's field to be the **ack_ctr** counter. If not indicate the agent that something is wrong by setting the **time_till_ack** item's field to **UNDEF**.

```
73          col_item.time_end = sys.time;
74          emit col_item.cover_item;
75          emit eot_ev;
76          item_done$(col_item,0);
77       }; // while TRUE
78    }; // collect_data()@...
79 };
```

Continuation of **collect_data ()**:

At the end the item gets the collecting end time, two events that indicate the end of an item's collection emits, and the item pass on to the **Reference Model** that will be shown and explained later through **item_done** that is a **method port**, also will be introduced later.

5.5.7 Agent

```
 4 unit vl_agent_u like vl_unit_u {
 5
 6   mon    : vl_monitor_u is instance;
 7
 8   when ACTIVE vl_agent_u {
 9
10       bfm        : vl_bfm_u is instance;
11    };
12
```

Very simple the agent is a wrapper for all the EVC's inside of it.
The code above is a part of the vl_base_agent which all agents inherit from.
There is here an instance of a monitor, and when **ACTIVE** an instance of BFM.

The **'when'** statement means that when will be an instance of "ACTIVE vl_agent_u" or any other agent that inherit from it, it will have a BFM. But agent that instanced as "PASSIVE vl_agent_u" will not have a BFM.

There are a lot of connecting of ports and pointers that are not shown here to the name and kind of the agent for each EVC, connect the BFM to the sequence driver, and the monitor to the right ports, connectors of the clocks, and more to set all EVC's related to each other, and work well.

```
28 extend vl_agent_name_t : [RIF_IN , RIF_OUT];
29 extend vl_agent_kind_t : [RIF];
30
31 extend RIF vl_agent_u {
32    !rif_ports    : rif_ports_u;
33
34    -- this set if the agent is on the MASTER or SLAVE
35    rif_kind : vl_rif_device_t;
36
37    timeout_time : uint;
38    keep soft timeout_time == 10000;
39
40    timer_to_ack : uint;
41    -- this way the monitor fails the test in case on no ack return
42    -- if you're expecting no ack due to error scenario,
43    -- constrain it to be less than timeout_time
44    keep soft timer_to_ack == timeout_time+10;
45
46    when ACTIVE vl_agent_u {
47       sd : vl_rif_sequence_driver_u is instance;
48
49       connect_pointers() is also {
50          sd.its_bfm = me.bfm;
51          sd.its_agt = me;
52          bfm.sd = sd;
53       }; // connect_pointer...
54    };
55
56    connect_pointers() is also {
57       rif_ports = ports.as_a(rif_ports_u);
58    }
59 };
```

The code above is the RIF's agent code.
The fields are known from the **BFM** and the **monitor**, and when **ACTIVE** there is an instance of the RIF's **sequence driver**.